Suad Alagić

Object-Oriented Technology

Springer

Object-Oriented Technology

Suad Alagić

Object-Oriented Technology

Springer

Suad Alagić
Computer Science Professor
Portland, Maine, USA
Publications: http://dblp.uni-trier.de/pers/hd/a/Alagic:Suad
Website: https://sites.google.com/site/suadalagicme/

ISBN 978-3-319-20441-3 ISBN 978-3-319-20442-0 (eBook)
DOI 10.1007/978-3-319-20442-0

Library of Congress Control Number: 2015944088

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

For
Adrian, Evan, Lucas, and Dario

Preface

The core idea of this book is that object-oriented technology is a generic technology whose various technical aspects could be presented in a unified and consistent framework. This applies to both practical and formal aspects of object-oriented technology.

The approach in the book is based on typed technologies. The core notions fit mainstream object-oriented languages such as Java and C#. However, perhaps the most distinctive feature of this book is that it is based on object-oriented constraints (assertions), their specification and verification. Object-oriented constraints apply to specification and verification of object-oriented programs, specification of the object-oriented virtual platform, more advanced concurrent models, database integrity constraints, and object-oriented transactions, their specification and verification.

In Chap. 1, we define the key notions of typed object-oriented technology: the notions of types, objects, classes, messages, and methods. This is followed by the notions of inheritance and subtyping and their formal and pragmatic properties. Static and dynamic binding as it applies to messages and methods is then discussed. Parametric types are essential for modern typed object-oriented technologies. They are given due attention through a detailed analysis of various issues related to parametric types.

A distinctive view in this book is that types are preferably equipped with assertions. Assertions are the topic of Chap. 2. This chapter first specifies the basic reasoning rules such as those that apply to assignments, conditional statements, and loops. Loop invariants and a verification technique for proving termination are then discussed. Object-oriented assertions: method preconditions and postconditions, and class invariants are discussed in detail. Finally, we revisit the notion of subtyping in the presence of assertions and explain the all important compatibility requirements for classes called behavioral subtyping. This chapter also shows how assertions are specified in a sample application that makes use of a practical assertion specification and verification tool.

The formal basis in this book for types and constraints had an impact on the notation used to present code samples that are based on Java, C#, assertion, and

data languages. Although these code samples are close to the source languages, they are more conceptual and based on notation that is closer to the standard mathematical notation. This produces a consistent framework for both code samples and their formal aspects expressed by constraints. The correspondence with the source notation is immediate.

In Chap. 3, we describe a modern environment that supports typed object-oriented technology. The components of this environment include reflection, the actual object-oriented virtual machine, and support for persistent objects. Reflection is presented following the initial Java specifications that are more suitable for explanation of core concepts unlike the more recent revisions. Modern typed object-oriented languages such as Java or C# run on an object-oriented virtual machine. Our presentation is an abstraction of the existing very low-level specifications of object-oriented virtual machines. A distinctive feature of the presentation is the use of assertions to define the semantics of the basic operations. An object-oriented platform should support persistent objects, i.e., objects with long lifetimes. This chapter also presents persistent models and mechanisms such as orthogonal persistence and serializability.

In Chap. 1, we present the basic rules of object-oriented type systems in a mostly informal manner. However, type systems have an elaborate formal theory. In Chap. 4, we show how some of the elements of type theory apply to an object-oriented programming language. We first explain the basic typing rules for statements such as assignment, conditionals, and loops, as well as for expressions. Then we focus on the rules of object-oriented type systems that govern classes, inheritance, subtyping, and messages. These static typing rules are followed by dynamic type checking rules that mainstream object-oriented languages have, and the formal rules for parametric types. In addition, we present static and dynamic typing rules that apply to reflection. Finally, we show what kind of problems violation of type safe rules can cause. We take a well-known paradigm of type erasure that was the basis of the technique for extending Java with parametric types. This paradigm is provably incorrect as demonstrated by a variety of cases of type violation presented in this chapter.

Concurrent object-oriented models are discussed in Chap. 5. We first present the model of concurrent threads of execution. Then, we explain the basic mechanisms for synchronizing concurrent access of multiple threads to objects on the common heap. We discuss the relationship between synchronization and inheritance, and the relationship between serialization and synchronization. We demonstrate what the implications are of synchronized versus unsynchronized executions. Then, we introduce a more general concurrent and distributed model based on asynchronous messages. The basis of this model is the fact that messages may be viewed as objects. In this new framework, it is natural to define formally the rules of behavioral subtyping. Finally, we introduce a very general model of concurrent active objects, each equipped with its own virtual machine. Ambients of such objects are discussed as well.

Object-oriented database technology (the topic of Chap. 6) addresses the limitations of relational and extended relational technologies in managing complex

objects that exhibit complex behavior. Object-oriented database technology is also addressing the problem of the mismatch between the dominating software technology, which is object-oriented, and the dominating database technology that is relational, or object-relational at best. This specifically applies to the so-called "impedance mismatch" between data and programming languages. We first discuss the basic features of object-oriented definition and query languages. We look at the most important object-oriented database technologies. We explain the core features of those technologies that include database collection classes, interfacing databases from an object-oriented programming language, and object-oriented transactions. We also revisit the typing issues for persistent collections. The core features of a specific current technology and its application are presented as well. Finally, we discuss a critical limitation of all object-oriented and other database technologies: inability to specify and enforce very general database integrity constraints. We present an object-oriented database paradigm based on such general constraints.

In Chap. 7, we show how two critical formal notions in this book: types and constraints, apply to two significant practical problems. The first problem is specification of database type systems in a formal manner. The second problem is specification and verification of object-oriented transactions. Database systems are typically not based on a formally specified type system. This leads to non-trivial problems that we demonstrate in this chapter. We consider a type system required by object-oriented database systems and establish a collection of the required formal rules showing the typing problems in object-oriented database languages. This chapter also addresses another key feature of database technologies: transactions. Unlike other approaches, object-oriented schemas and transactions are in our approach equipped with integrity constraints expressed in an object-oriented assertion language. This is followed by object-oriented consistency issues. The presentation is based on a sample object-oriented schema equipped with very general integrity constraints and sample transactions. The relationship between the integrity constraints and queries is discussed, as well as inheritance for object-oriented schemas equipped with constraints. An essential issue of the role of dynamic versus static checking of constraints is also addressed in this chapter.

Portland, Maine, USA Suad Alagić

Acknowledgements

The author is grateful to two software engineers, Russell Gillen and Thomas Keschl, his former graduate students, for their numerous valuable comments that significantly improved the manuscript.

Contents

1 Typed Objects 1
1.1 Typed Objects 1
1.1.1 Objects and Classes 1
1.1.2 Inheritance 3
1.1.3 Subtyping 6
1.1.4 Static and Dynamic Binding 9
1.2 Parametric Types 12
1.2.1 Collection Types 12
1.2.2 Parametric Types 15
1.2.3 Implementing Parametric Classes 17
1.2.4 Abstract Classes 17
1.2.5 Parametric Types and Subtyping 20
1.3 Bibliographic Remarks 21
1.4 Exercises 21
References 23

2 Assertions 25
2.1 Declarative Specifications 25
2.1.1 Assertions 25
2.1.2 Basic Reasoning Rules 27
2.2 Object-Oriented Assertions 31
2.2.1 Preconditions and Postconditions 32
2.2.2 Loop Invariants 33
2.2.3 Termination 34
2.2.4 Object Invariants 35
2.2.5 Assertions for Collections 37
2.2.6 Behavioral Subtyping 40
2.3 Sample Application 42
2.4 Bibliographic Remarks 46
2.5 Exercises 47
References 48

3 Virtual Platform 49
3.1 Reflection 50
3.1.1 Reflective Classes 50
3.1.2 Class Objects 51
3.1.3 Field Objects 53
3.1.4 Method Objects 54
3.1.5 Constructor Objects 54
3.1.6 Updating Fields 55
3.1.7 Invoking Methods 55
3.1.8 Creating Class Objects 56
3.1.9 Class Files 57
3.2 Virtual Machine 58
3.2.1 The Structure of the Virtual Machine 58
3.2.2 Creating Objects 60
3.2.3 Invoking Methods 61
3.2.4 Accessing Fields 61
3.2.5 Operations 63
3.2.6 Arrays 64
3.3 Extending Virtual Platform 67
3.4 Persistent Objects 71
3.4.1 Orthogonal Persistence 71
3.4.2 Persistence Architecture 74
3.4.3 Object Serialization 75
3.5 Bibliographic Remarks 77
3.6 Exercises 78
References 79

4 Type Systems 81
4.1 Formal Type Systems 82
4.1.1 Typing Rules 82
4.1.2 Object-Oriented Type System 84
4.1.3 Dynamic Type Checking 88
4.1.4 Parametric Types 89
4.2 Reflection 91
4.3 Type Erasure 94
4.3.1 Type Erasure Idiom 94
4.3.2 Static Type Checking 95
4.3.3 Dynamic Type Checking 99
4.3.4 Persistence by Serializability 99
4.3.5 Reflection 101
4.4 Bibliographic Remarks 102
4.5 Exercises 102
References 103

5 Concurrent Models 105
5.1 Concurrent Threads 105
5.1.1 Thread Objects 105
5.1.2 Synchronized Objects 107
5.1.3 Synchronization and Inheritance 109
5.1.4 Concurrency and Serialization 110
5.1.5 Synchronized Versus Unsynchronized Executions 111
5.2 Messages as Objects 113
5.2.1 Types of Messages 113
5.2.2 Typing Rules 116
5.2.3 Behavioral Subtyping 119
5.3 Concurrent Objects 120
5.3.1 Ambients of Concurrent Objects 120
5.3.2 Reflection 124
5.3.3 Virtual Machine 127
5.4 Bibliographic Remarks 130
5.5 Exercises 130
References 131

6 Object Databases 133
6.1 Query Languages 134
6.1.1 Object Definition Language 134
6.1.2 Object Query Language 135
6.1.3 Java OQL 137
6.1.4 Language Integrated Queries (LINQ) 141
6.2 Object Databases 144
6.2.1 ODMG Java Binding 144
6.2.2 Parametric Persistent Collections 149
6.2.3 Db4 Objects 151
6.2.4 Database Application 155
6.3 Database Constraints 158
6.4 Bibliographic Remarks 162
6.5 Exercises 163
References 164

7 Types and Constraints 165
7.1 Database Type Systems 166
7.1.1 ODL and OQL Examples 166
7.1.2 Schemas 167
7.1.3 Inheritance 168
7.1.4 Structures 169
7.1.5 Interfaces 170
7.1.6 Classes 172
7.1.7 Objects and Messages 173
7.1.8 Type Checking OQL Queries 174
7.1.9 OQL Queries and Parametric Types 178

7.1.10 Queries with Order by Clause 179
7.1.11 Java OQL 181
7.1.12 Typing Ordered Collections and Indices 184
7.2 Transactions 186
7.2.1 Transactions and Constraints 186
7.2.2 Schemas and Transactions 188
7.2.3 Levels of Consistency 189
7.2.4 Constraints for Schemas 191
7.2.5 Sample Transactions 192
7.2.6 Constraints and Queries 194
7.2.7 Specification Inheritance 195
7.2.8 Abstraction 198
7.2.9 Dynamic Checking of Constraints 199
7.3 Bibliographic Remarks 200
7.4 Exercises 200
References 201

Index 203

Chapter 1
Typed Objects

In this chapter we define the key notions of typed object-oriented technology.

The very basic notions are introduced in Sect. 1.1. The notions of types, objects, classes, messages and methods are introduced and illustrated in Sect. 1.1.1. The notion of inheritance is the topic of Sect. 1.1.2. A related notion of subtyping and its formal properties are discussed in Sect. 1.1.3. Static and dynamic binding as it applies to messages and methods is the topic of Sect. 1.1.4.

Parametric types are essential for modern typed object-oriented technology. This is the topic of Sect. 1.2. Collection types are of particular importance for the topics presented in this book. Their non-parametric versions and their limitations are discussed in Sect. 1.2.1. Parametric collection types, different types of parametric polymorphism and their advantages are discussed in Sect. 1.2.2. A fully implemented parametric collection class is given in Sect. 1.2.3. Abstract parametric classes are discussed in Sect. 1.2.4. The impact of parametric types on subtyping is discussed in Sect. 1.2.5.

1.1 Typed Objects

1.1.1 Objects and Classes

An abstract data type defines its instances entirely in terms of actions that can be performed upon those instances. An example is an abstract data type *TwoDPoint* specified below as an interface. The actions are reading and setting the values of the coordinates of an instance of the *TwoDPoint* type.

```
interface TwoDPoint {
 int getX();
 int setX(int x);
```


S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0_1

```
 int getY();
 int setY(int y);
}
```

The above definition is an abstraction offered to the users of the *TwoDPoint* type. The users can only see the signatures of operations, i.e., their names, the types of their arguments, and the type of their result. All details of representation of instances of this type are hidden from the users. A representation of the *TwoDPoint* type is given in the class *TwoDPoints* given below. The components of the object state are declared as private and the methods that either read or update the object state as public. This way the object state is encapsulated and accessible only by invoking public methods.

```
class TwoDPoints implements TwoDPoint {
 private int x;
 private int y;
 public int getX(){
      return x;
 }
 public void setX(int x) {
      this.x=x;
 }
 public int getY() {
      return y;
 }
 public void setY(int y) {
      this.y=y;
 }
}
```

Instances of a class are objects. An object has three defining components:

- object identity
- object state
- methods applicable to the object

When an object is created a unique identity is assigned to the object. Details of representation of the object identity are hidden from the users of the object.

In the example below expressions x.setX(5) and x.setY(10) are called messages.

```
TwoDPoint p;
p.setX(5); p.setY(10);
```

In general, invoking a method m of a class C whose signature is B m(A1, A2,. . ., An), where B, A1, A2, . . . ,An are types, is called a message and it has the form

a.m(a1,a2,. . .,an)

where the type of a is C and the type of ai is Ai for i=1,2,. . .,n. The object a in the above message is called the receiver of the message and a1, a2, . . .,an are arguments. All an object x needs to know in order to invoke a message on an object a is the interface of the type of a as in the example above.

Object-oriented languages have a special keyword (like *self* or *this*) to denote the receiver of a message. In the above example of *TwoDPoints* class the methods *setX* and *setY* refer to the coordinates of the receiver object using the keyword *this*. This is also necessary in order to distinguish the value of the formal parameter and the value of the corresponding coordinate of the receiver object. In the methods *getX* and *getY* the values of the coordinates of the receiver object are referred to directly by their names so that *this* is implicit.

Objects are created dynamically by invoking a class constructor as in the example below:

```
TwoDPoint obj=new TwoDPoints();
obj.setX(5); obj.setY(10);
```

In the above example a no argument constructor is invoked initializing the *TwoDPoint* object to the default values for the types of components of the object state. Additional constructors are typically defined for a class that would specify a particular initialization of the object state as in the following example.

```
TwoDPoints(int xVal, int yVal) {
 x=xVal; y=yVal;
}

TwoDPoint obj= new TwoDPoints(5,100);
```

1.1.2 Inheritance

A core feature of object-oriented languages allows specification of new types by derivation from the already defined types. This is how software reuse is accomplished in object-oriented technology. An example is a type *MovingObject* defined as an extension of the type *TwoDPoint*:

```
interface MovingObject extends TwoDPoint {
 float getSpeed();
 void setSpeed(float newSpeed);
}
```

Instances of the type *MovingObject* inherit all the components and methods of the base type *TwoDPoint*. Additional components and methods of the type *MovingObject* that are specific to those objects are defined in the specification of the type *MovingObject*.

The class that implements the interface *MovingObject* specifies the additional components of the moving object state along with the associated methods:

```
class MovingObjects extends TwoDPoints {
 private float speed;
 public float getSpeed() {
      return speed;
 }
 public void setSpeed(float newSpeed) {
      speed=newSpeed;
 }
}
```

The class *MovingObjects* inherits the implementation of *TwoDPoint* objects and implements the methods whose signatures are specified in the interface *MovingObject*.

The states and the associated methods of objects of types *TwoDPoint* and *MovingObject* are represented in Fig. 1.1.

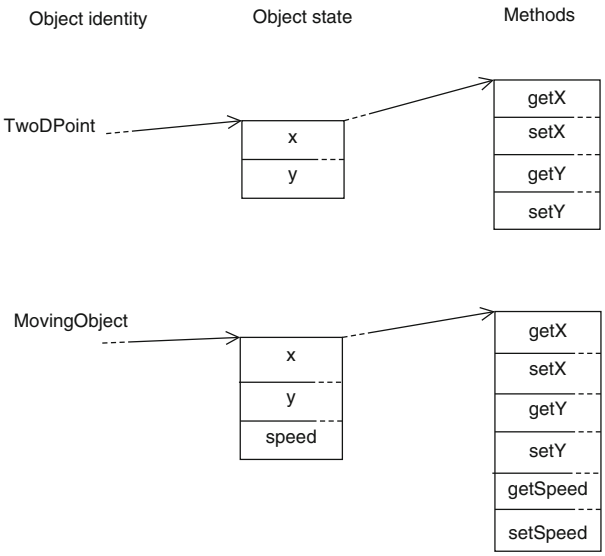


Fig. 1.1 Object states and methods

The basic features applicable to all object types are specified in the class *Object*. The details of representation of this class are not exposed to the users. A simplified specification of the class *Object* that omits signatures of other methods is given below.

```
public class Object
 public boolean equals(Object x);
 public Class getClass();
 // other methods
}
```

All classes implicitly inherit from the class *Object*. The two methods whose signatures are specified above make it possible to test whether two objects are equal and to access the class information available at run time.

The inheritance relationships in our example are represented in Fig. 1.2. This diagram illustrates the type of multiple inheritance allowed in Java and C#.

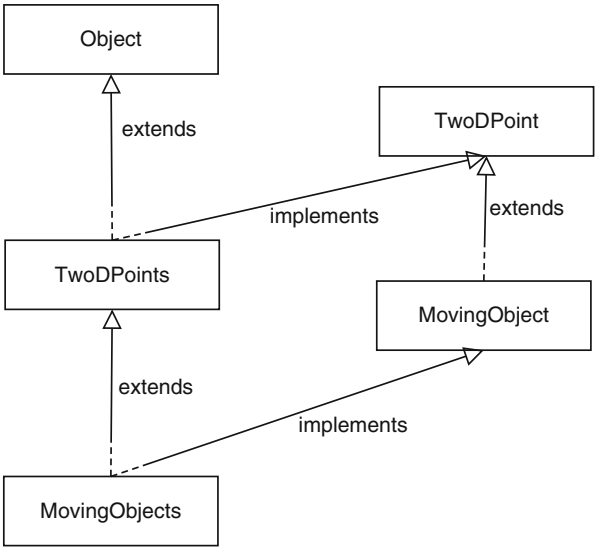


Fig. 1.2 Inheritance for interfaces and classes

A class can have a single superclass, and the root class *Object* has none. Multiple inheritance for classes is not allowed because it creates problems since a class specifies an implementation. If a class extends two different and independently developed classes, the question is which implementation is being inherited. A particularly problematic situation occurs in the case of diamond inheritance illustrated

in Fig. 1.3. All the types in this diagram cannot be classes. A class can implement multiple interfaces. An interface can extend multiple interfaces because interfaces do not contain implementation. The only conflict that may occur is with names and signatures of methods inherited from multiple superinterfaces. These issues are easily resolved with simple rules that Java and C# have. In the above example the class *MovingObjects* extends the class *TwoDPoints* and implements an interface *MovingObject*.

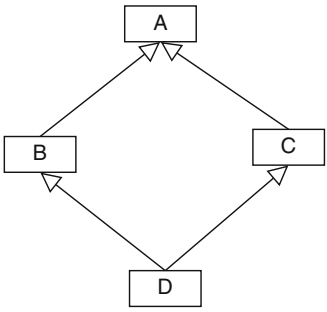


Fig. 1.3 Multiple inheritance

The immutable class object contains run-time representation of signatures of the class fields, constructors and methods. The method *getSuperClass* applied to a class object produces a reference to the superclass object. This makes the complete type hierarchy of classes available at run-time. However, this hierarchy of types can be only introspected, and not changed at run time. This is why the class *Class* contains only introspection methods and cannot be extended (it is final). Changing at run time the type information produced in the process of compilation would completely defeat the purpose of a type system.

```
public final class Class {
 // methods for accessing field signatures
 // methods for accessing constructor signatures
 // methods for accessing method signatures
 public Class getSuperClass();
}
```

The relationships between objects and their class objects is represented in Fig. 1.4.

1.1.3 Subtyping

The basic form of inheritance presented so far amounts to extension of both components of the object state, and the set of the associated methods. A subclass

inherits all of them just the way they are defined in the superclass. This makes it possible to have a particular flexibility in object-oriented languages. An instance of a subclass could be safely substituted where an instance of the superclass is expected. This is not a literal substitution of the object representation. Substitution applies to object identities that are implemented as references to object states. A reference to an object of the superclass is replaced by a reference to an object of the subclass.

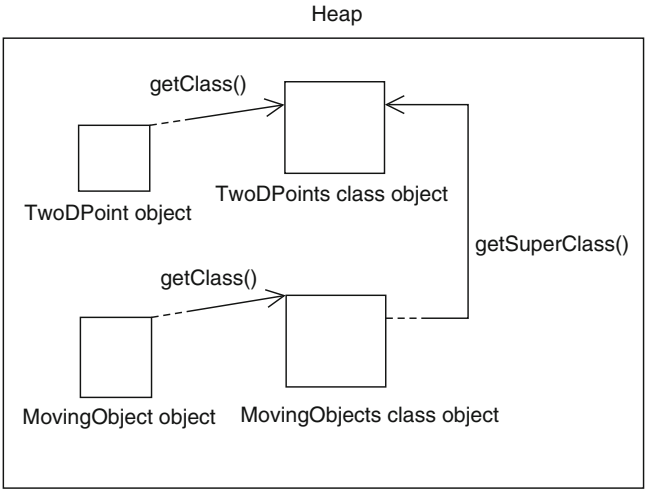


Fig. 1.4 Objects and class objects

The general term polymorphism applies to situations like this where an instance of one type is substituted where an instance of a different type is expected. The form of polymorphism that is associated with inheritance is called subtype polymorphism. Although these two notions are in general different, they are identified in mainstream typed object-oriented languages. That is, the typing rules for deriving a subclass from another class by inheritance guarantee that the subclass defines a subtype of the superclass.

The subtleties of typing rules for inherited methods come from the rule for type-safe substitution of functions. Consider a function:

f: Domain → CoDomain

Let us see under what conditions we can substitute a function

f′: Domain′: → CoDomain′:

in place of f.

Given any x ∈ Domain, we would like to substitute f′(x) in place of f(x). This means that we must have x ∈ Domain′ or else f′(x) will be undefined. In other words,

Domain ⊆ Domain′

On the other hand, f(x) ∈ Codomain for any x ∈ Domain, so that we must also have that f′ (x) ∈ Codomain. In other words, we have:

CoDomain′ ⊆ CoDomain

This situation is illustrated in Fig. 1.5.

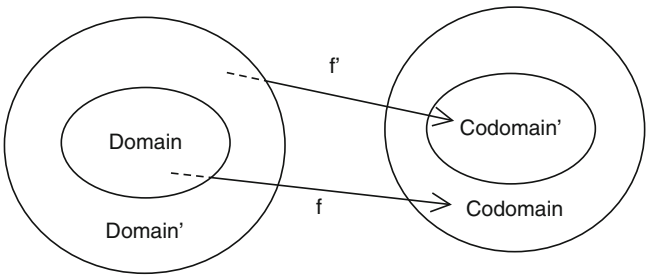


Fig. 1.5 Function subtyping

In general, a function type

T1′ → T2′

where T1 and T2 are types, is a subtype of the function type

T1 → T2

if the following requirement is satisfied:

T1 <: T1′ (contravariance) and T2′ <: T2 (covariance)

where the symbol <: is a standard symbol for subtyping. The mathematical terms covariance and contravariance simply mean the following. In the case of covariance function subtyping has the same direction as subtyping of the corresponding result types, and in the case of contravariance function subtyping and subtyping for the corresponding argument types have reversed directions.

This rule for function types applies to inheritance of methods. The most restrictive implementation of the rule for function subtyping is that an inherited method has exactly the same signature in a subclass as it does in the superclass. So the name, the types and the number of arguments and the result type will be the same. A more flexible type safe discipline is that if the method signature in the superclass is T2 f(T1), then its signature in the subclass will be T2′ f(T1) where T2′ <: T2. This is a special case of the rule for function subtyping. For example:

```
public class Object {
  public Object clone()
// other methods
}

public class TwoDPoints {
  public TwoDPoints clone()
// other methods
}
```

A field f of type T is in fact a pair of methods with signatures:

```
T get()
void set(T)
```

Consider the following class:

```
class A {
 Ta f;
 Ta getf(){return f; }
 void setf(Ta value){ f = value; }
}
```

Let us assume that the type of the field f in class A has been changed to Tb in a class B derived from A by inheritance:

```
class B extends A {
 Tb f;
 Tb getf(){return f; }
 void setf(Tb value){f = value; }
}
```

The rule for function types will require

Tb <: Ta and Ta <: Tb

This means that the type of the inherited field must remain the same in the subclass because subtyping is a partial order (reflexive, antisymmetric and transitive).

1.1.4 Static and Dynamic Binding

The flexibility introduced by subtype polymorphism makes the declared (static) type of an object in general different from its run-time (dynamic) type. The run-time type is in general a subtype of the static type. For example:

```
TwoDPoint x = new TwoDPoints();
MovingObject y = new MovingObjects();
x=y;
```

The static type of x is *TwoDPoint* and after the assignment x=y its run-time type is *MovingObject*. This has implications on selection of the most appropriate method when executing a message. Consider the class *Vehicle* given below. In the notation used in this book the symbol = is overloaded. It stands for the standard mathematical notion of equality as in the code below, as well as for the assignment as in Java and C#.

```
public class Vehicle {
 private int VIN;
 private String make;
 public boolean equals(Object x) {
      return (VIN = (Vehicle)x.VIN);
 }
 // other methods
}
```

The method *equals* in the root class *Object* is defined as the test on object identity. This is the only meaningful way of defining the equality of objects in general. That is, two objects are equal if they have the same identity. In a specific class a more suitable meaning of equality may be more appropriate. In the above class two vehicles are considered equal if their VINs (vehicle identification numbers) are equal. So the method *equals* is redefined accordingly in the class *Vehicle*. This redefinition of an inherited method is called overriding.

Note that the signatures of the inherited method arguments are required to remain the same in the subclass according to the function subtyping rule. This creates an awkward situation because in the class *Vehicle* we would like to refer to the VIN field of the argument, and *Object* does not have such a field. This is why the type cast (Vehicle)x specifies that the intent is to view x as a *Vehicle*. There is no way to verify this cast statically, hence a dynamic check is generated by the compiler. This is an instance of dynamic type checking.

In order for overriding to work correctly the method to be invoked is determined by the dynamic type of the receiver object. This is called dynamic dispatch of methods and it represents the most important case of dynamic binding in object-oriented languages. For example,

```
Object x = new Object();
Object y = new Object();
Vehicle xV = new Vehicle();
Vehicle yV = new Vehicle();
x=xV; y=yV;
. . . x.equals(y). . .
```

The method to be invoked in response to the message x.equals(y) will be the method *equals* overridden in the class *Vehicle* because the run time type of x is *Vehicle*.

There are situations in which overriding a method should not be allowed. An example is the method *getClass* of the root class *Object*. This method has a particular implementation in the underlying virtual platform, which guarantees that invocation of this method will indeed return the class object of the receiver of the method. Allowing overriding would have serious implications on the intended semantics of this method creating nontrivial problems in dynamic type checking. This is why the method *getClass* is declared as final.

```
public class Object {
 public final Class getClass();
 // other methods
}
```

The class *Class* is final, which means that it cannot be extended, and hence none of its methods can be overridden. Since the class *Class* has only introspection methods, this guarantees safety of the type system at run-time, i.e., the type information cannot be mutated at run time.

Dynamic dispatch (selection) of methods based on the type of the receiver object is the basic technique in object-oriented languages. It brings the type of flexibility that makes the whole object-oriented paradigm work. Adding new types by inheritance to an already compiled and running application requires only compilation and linking of the newly introduced types without recompiling the existing application. However, this flexibility comes with some penalty in efficiency because the decision about method selection is postponed to runtime. Modern languages have efficient techniques for dynamic dispatch of methods, but some languages like C++ and C# try to avoid the associated cost by providing a static binding (method selection) option. In C#, methods are statically bound unless they are explicitly declared as *virtual*. For example, using our notation, the method *equals* which is intended to be overridden would be declared as follows:

```
public class Object {
 public virtual boolean equals(Object x);
 // other methods
}
```

Overriding this method in C# will be indicated by an explicit keyword **override**.

```
public class Vehicle {
 private int VIN;
 private String make;
 public override boolean equals(Object x) {
      return (VIN = (Vehicle)x.VIN);
```

```
 }
 // other methods
}
```

Methods whose receiver is the class object are always bound statically. The reason is that there is only one class object for all objects of that class. Since the receiver is known at compile time, there is no need to postpone method selection to run time. These methods are thus declared as static to indicate that they belong to the class itself. An example is the method *numberOfVehicles* of the class *Vehicle* The number of vehicles is not the property of individual vehicle objects. It is the property of all objects of the class *Vehicle*, hence it belongs to the class itself.

```
public class Vehicle {
// fields;
 public static int numberOfVehicles();
 // other methods
}
```

We summarize the above discussion as follows:

- The basic mechanism for selecting a method for executing a message (method dispatch) in object-oriented languages is dynamic. It is based on the run-time type of the receiver object.
- The receiver of a static (i.e. class) method is the class object. Since there is only one class object of a given type, selection of a static method is static.
- Some languages (C++ and C#) allow a choice of static versus dynamic method dispatch. Although this is done for the reasons of efficiency, it has been shown that when both dispatch mechanisms are used in a program, that may obscure the meaning of the program.

1.2 Parametric Types

1.2.1 Collection Types

Most object-oriented languages are equipped with the root class *Object* (C++ is a notable exception). The class *Object* along with subtype polymorphism allows specification of the *Collection* type whose elements are simply objects. In fact, this was the only way of specifying a generic collection type in the initial versions of Java and C#:

```
public interface Collection{
 public boolean isMember(Object x);
 public void add(Object x);
 public void remove(Object x);
}
```

An object of type *Collection* is equipped with methods for testing whether an object belongs to the collection, inserting new objects into the collection, and deleting objects from the collection. The main problem with this specification is that objects of any type may be inserted into a collection defined this way. If we would like to define a specific collection, like a collection of employees, we would do it as follows:

```
Collection employees;
```

So the following type checks:

```
Emloyee emp = new Employee();
employees.add(emp);
```

However, so does the following

```
Department dept = new Department();
employees.add(dept);
```

The reason is that both *Employee* and *Department* are subtypes of *Object*, and so is any other object type. The other problem occurs when getting objects from a collection as in the *for* statement below. This statement introduces a control variable *emp* of type *Employee* and iterates over the collection *employees*. In the process the control variable assumes the values of the elements of the collection *employees*.

```
for (Employee emp: employees)
 emp.displaySalary();
```

The above code will not compile in Java because the control variable *emp* is declared to have the type *Employee*, and the elements of the collection *employees* are of type *Object*. Let us modify the above loop to correct this type mismatch as follows:

```
for (Object emp: employees)
 emp.displaySalary();
```

The above will not type check either because the class *Object* is not equipped with a method displaySalary(). This is why a type cast is necessary:

```
for (Object emp: employees)
 (Employee)emp.displaySalary();
```

This type cast looks redundant, but it is necessary because an object retrieved from a collection of employees may not be an employee at all. So not only is a

dynamic check necessary, but it may fail at run-time as well. There is really no good solution for this situation. In order to avoid program failure the original code must be extended with exception handling as follows. This is hardly an attractive way to specify an iteration over a collection of employee objects.

```
try {
      for (Object emp: employees)
      (Employee)emp.displaySalary();
      }
 catch (ClassCastException classEx )
      {exception handling }
```

The notion of a set of objects may be defined using subtype polymorphism as follows:

```
public interface Set extends Collection {
  public Set union(Set s);
  public Set intersection(Set s);
}
```

The difference between a collection in general and a set is that an element may belong multiple times to a collection. The notion of a set does not allow this: an element is either a member of a set or it is not. This is why an element cannot be inserted into a set if it already belongs to the set. The behavior of delete is also different for collections and sets. Deleting an object from a set means that the object does not belong to the set. Because of possible multiple occurrences of an object in a collection, that would happen only if the last occurrence is deleted. In addition, sets are equipped with operations such as *union* and *intersection* that collections in general do not have (Fig. 1.6).

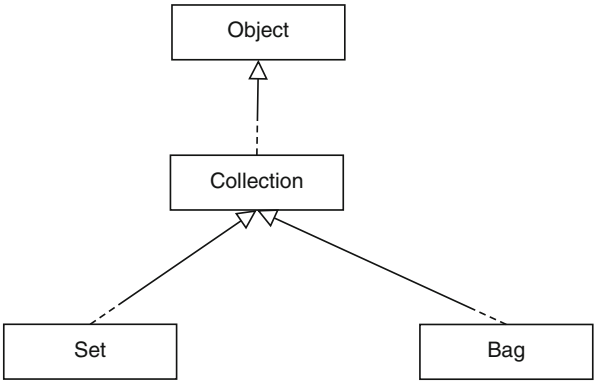


Fig. 1.6 Collection types

A bag is a collection that keeps explicit count of the number of occurrences of each element that belongs to the bag. In addition, a bag is equipped with operations such as *union* and *intersection*. The semantics of these operations are defined in such a way that they reduce to the semantics of *union* and *intersection* for sets in a particular case of a bag that is in fact a set.

The rule for union of bags is the following. If an element x belongs m times to a bag B1 and n times to a bag B2, then x will belong max(m,n) times to the union of B1 and B2. Symmetrically, x will belong min(m,n) times to the intersection of B1 and B2.

1.2.2 Parametric Types

The problems in specifying collection types using subtype polymorphism are avoided by a different form of polymorphism called parametric polymorphism. Using this form of polymorphism the notion of a collection is specified as follows.

```
public interface Collection<T> {
  public boolean isMember(T x);
  public void add(T x);
  public void remove(T x);
}
```

The interface *Collection* now has a type parameter T. T stands for any object type, so that there is an implicit quantification over all object types. Collection<T> may be viewed as a template for construction of new types by substitution. Substituting *Employee* for the type parameter T produces a collection of employees type denoted as:

Collection<Employee>

The previous code obtains the following form:

```
Collection<Employee> employees;
Employee emp = new Employee();
employees.add(emp)
```

However, the following will not type check, i.e., it will produce a compile-time error.

```
Collection<Employee> employees;
Department dept = new Department();
employees.add(dept)
```

Accessing elements of the collection of employees does not require a type cast. So a previous *for* statement will now type check:

```
for (Employee emp: employees)
 emp.displaySalary();
```

The key property of parametric polymorphism is that it allows static (i.e., compile-time) type checking. The unfortunate consequences of dynamic type checks are completely avoided.

The universal form of parametric polymorphism does not allow static typing of a variety of other abstractions such as ordered collections, ordered sets etc. The problem with the definition of the type of ordered collections as:

OrderedCollection<T>

is that it does not guarantee that the object types substituted for T will be equipped with ordering. This is why the parametric type *OrderedCollection* is defined with a type constraint for the type parameter:

OrderedCollection<T **extends** Comparable<T>>

where the interface *Comparable* is equipped with a comparison method and defined as follows:

```
public interface Comparable<T> {
  public int compareTo(T x);
// other comparison methods
}
```

Parametric interface *OrderedCollection* is now specified as follows:

```
 public interface OrderedCollection<T extends Comparable<T>>
                extends Collection<T> {
// . . .
}
```

The above specification means that only types that extend or implement the interface *Comparable* will be acceptable as the actual type parameters of OrderedCollection<T **extends** Comparable<T>>. So if we have

Employee **implements** Comparable<Employee>

OrderedCollection<Employee> will satisfy the static type check.

The form of parametric polymorphism in which there is a bound on the type parameter is called bounded. When the bound itself is parametric, like in the above cases, the form of parametric polymorphism is called F-bounded.

Array is a parametric type with special notation and special properties. T[] is an array type for any specific type T. The form of parametric polymorphism is thus universal.

1.2.3 Implementing Parametric Classes

The methods of the class *OrderedCollection* are those specified in the interface *Collection* which the class *OrderedCollection* implements. The representation of a sorted collection in the class *OrderedCollection* is a linked list, where *LinkedList* is an already defined parametric class. Methods of the class *OrderedCollection* are implemented by invoking methods of the class *LinkedList*. The main difference is in the method *add* which is overridden in such a way that insertion into the underlying linked list maintains the ordering of the sorted collection of elements. The type constraint T **extends** Comparable<T> guarantees that elements of the sorted collection are equipped with the method *compareTo*. In the code below ¬ denotes negation.

```
public class OrderedCollection<T extends Comparable<T>>
                implements Collection<T> {
 private LinkedList<T> elements;
 public OrderedCollection() {
      elements = new LinkedList<T>();
 }
 public boolean isMember(Object e) {
      return elements.contains(e);
}
 public void add(T e) {
 if (¬ elements.contains(e)) {
 for (int i = 0; i < elements.size() - 1; i++) {
      if (elements.get(i).compareTo(e) ≤ 0 ∧
      elements.get(i + 1).compareTo(e) > 0)
      elements.add(e);
}
 public void remove(Object e) {
      if (elements.contains(e))
      elements.remove(e);
}
```

1.2.4 Abstract Classes

The class *AbstractBag* is an example of an abstract class. An abstract class is a partially implemented class. It is different from an effective class in that it has at least one abstract method whose implementation is deferred to the classes that extend the abstract class.

In the extreme case when an abstract class has only abstract methods, it looks just like an interface. However, there are major differences between an abstract class and

an interface. An interface has only public methods. An abstract class may have fields and implemented methods. Standard access rights (private, protected, public) may be specified for members of an abstract class whereas methods of an interface are necessarily public. An abstract class is a class, so that in languages like Java and C#, single inheritance applies to it as for classes in general, whereas multiple inheritance applies to interfaces.

Creating an instance of an abstract class makes no sense and it is not possible. Such an instance would in general have an incomplete specification of the object state and unimplemented methods. The run-time consequences are obvious.

```
public abstract class AbstractBag<T> {
 public abstract int size();
 public abstract int occurrences(T x);
 protected class Member<T> {. . . }
 // code for union, intersection and copy
}
```

One subtlety in the partial implementation of *AbstractBag* is the internal representation of bag elements. It differs from the type of the actual parameter of *AbstractBag*. The internal representation is specified in the protected class *Member* whose objects contain the actual data of type T along with the associated count of the number of occurrences. The access right for this class is declared as protected to make it available to subclasses derived from the abstract class *AbstractBag*.

```
protected class Member<T> {
 T value;
 int count;
 @Override
 public boolean equals(Object other) {
      Member mem = (Member) other;
      return this.value.equals(mem.value);
 }
 // other methods
}
```

Type signatures of the methods *copy*, *union*, and *intersection* are given below. These are examples of parametric methods because they introduce their own type parameter B.

```
 public abstract < B extends AbstractBag<T> > B copy(); {
// code for this method
}
 public < B extends AbstractBag<T>> B union(B otherBag) {
 // code for this method
}
 public <B extends AbstractBag<T>> B intersection(B otherBag){
 //code for this method
}
```

The result type of the methods *copy*, *union*, and *intersection* is specified using F-bounded parametric polymorphism. To see why this is appropriate assume the following:

OrderedBag< T **extends** Comparable<T> > **extends** AbstractBag<T>
Employee **implements** Comparable<Employee>

The results of the inherited methods *union* and *intersection* in the class OrderedBag<Employee> will be exactly what we want:

OrderedBag<Employee>.

The reason for providing the intermediate level of abstraction in terms of abstract classes is illustrated by the implementation of the methods *union* and *intersection* of the class *AbstractBag*. In spite of the fact that the implementation is left to the classes derived from *AbstractBag*, it is possible to specify the algorithms for computing the union and the intersection. These algorithms make use of the method for copying a bag and they follow the definitions of these two operations as defined in the algebra of bags.

The algorithms for *union* and *intersection* make use of the following parametric interface:

```
public interface Iterator<T> {
 boolean hasNext();
 T next();
 void remove();
}
```

The algorithm for the union copies the larger bag to the result bag, and it then iterates over the other bag to add new elements that belong to the other bag or set correctly the number of occurrences for elements belonging to both. The algorithm for intersection will look alike, except that it will copy the smaller bag to the result first.

```
 public < B extends AbstractBag<T>> B union(B otherBag) {
      B result, other;
 if (this.size() < otherBag.size()) {
      result = otherBag.copy();
      other = this;
 }
 else {
      result = this.copy();
      other = otherBag;
      }
 Iterator<Member> itr = other.iterator();
      Member m, n;
      while (itr.hasNext()) {
      m = itr.next();
```

```
    n = other.getMember(m);
    if (n ≠ null)
    set count of n to max(m,n);
    else result.addMember(m);
    }
return result;
}
```

This discussion shows that abstract classes along with more sophisticated forms of parametric polymorphism provide a powerful abstraction mechanism. Combination of these two techniques allows specification of generic libraries that can be tailored to particular applications contributing significantly to the core feature of the object-oriented paradigm: software reuse.

1.2.5 Parametric Types and Subtyping

Assume that we have

class Employee **extends** Person {. . .} .

We know that this implies

Employee <: Person

following the rules for the signatures of inherited fields and methods. The question is now whether this implies (Fig. 1.7)

Collection<Employee> <: Collection<Person> ? ? ?

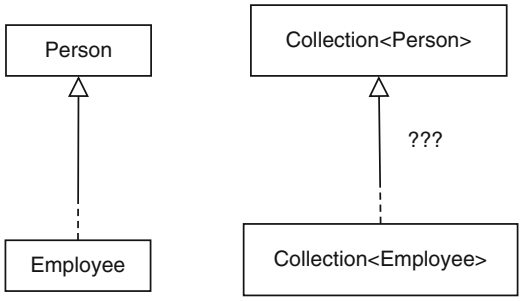


Fig. 1.7 Parametric types and subtyping

The answer is no. It is easy to see why. The signature of the method *add* in the class Collection<Person> is *void add(Person x)* and the signature of this method in the class Collection<Employee> is *void add(Employee x)*. Assuming that Collection<Employee> <: Collection<Person> violates the typing rules for inherited methods. The argument signatures of an inherited method must remain the same as in the superclass.

Following the same argument we would have for array types:

Employee[] $\not<:$ Person[]

However, languages like Java and C# still allow substitution of objects of type Employee[] in place of an object of type Person[]. This flexibility has pragmatic reasons. For example, an algorithm that sorts an array of persons would not be otherwise applicable to an array of employees. However, this relaxation of the static typing rules requires dynamic checks in order to prevent violation of subtyping at run-time, which both Java and C# have.

1.3 Bibliographic Remarks

The core object-oriented notions (objects, classes, inheritance, subtyping, dynamic binding) are presented following the Java programming language [1, 6]. For a formal view of subtyping see [3]. Problems caused by violation of type safe rules in object-oriented languages are discussed in [5]. Parametric types for object-oriented languages and their more sophisticated forms are formalized in [4]. The example of an abstract parametric type is due to Ben Gruba. Parametric types for Java generated a major controversy discussed in [8]. The reason is a wrong idiom for incorporating parametric types in Java advocated in [2, 7] and officially accepted for Java with nontrivial negative implications.

1.4 Exercises

1. Specify and implement a parametric class *Queue* using a parametric class *List* to represented a queue.
2. Specify classes *Set* and *OrderedSet* using two different idioms:
   - The type of elements is specified as *Object*
   - *Collection* and *OrderedCollection* are parametric types where the proper form of parametric polymorphism is used.

   Demonstrate the problems of the first representation and the advantages of the second in a simple application program on sets and ordered sets.
3. Specify a parametric class *Dictionary* with two type parameters. The first parameter stands for the type of entries in the dictionary. The second type parameter stands for the type of the key. Use proper form of parametric polymorphism.
4. Specify a class *SearchTree* following the pattern for the class *Dictionary*. In addition, a search tree is sorted and it has methods for sequential processing and search based on the given key. A suggested implementation is a balanced binary search tree.

5. Consider a class *Person* and its subclass *Employee*. Assume that *PersonalApplication* is a class with a method *process* whose signature is

   void process(Collection<Person>).

   – Is the method *process* applicable to the argument of type Collection<Employee>?
   – Specify *process* as a parametric method using F-bounded parametric polymorphism so that it will be applicable to Collection<Emplyee>.
6. Specify a parametric method *max* that returns the larger of its two arguments. Use F-bounded parametric polymorphism.
7. Specify a class *Airport* whose components are a list of runways, a list of traffic patters (queues of incoming aircraft), a set of ground structures, and a location. Use the appropriate parametric types (list, queue, set).
8. Specify class *Runway* whose components are heading, length, width, elevation, orientation and an aircraft queue.
9. Specify a hierarchy of aircraft types. The following two top levels might look like in Fig. 1.8.

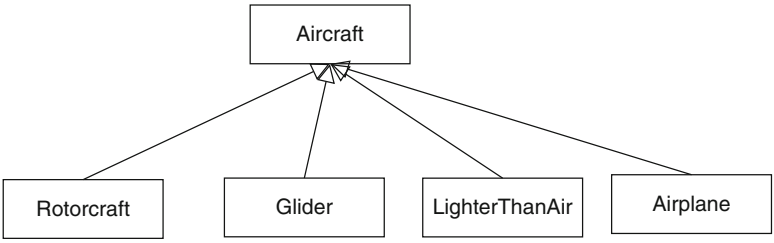


Fig. 1.8 Aircraft hierarchy

10. Specify the methods of the class *Aircraft* that reflect general behavior of an aircraft. Override those methods so that they reflect behavior of airplanes, and then subtypes of the type *Airplane* such as *PassengerPlane*.
11. Specify the hierarchy of military planes that has two subtypes, strategic and tactical military planes. Further decomposition should include multiple inheritance.
12. Specify classes and interfaces that implement the model with multiple inheritance in Fig. 1.9.

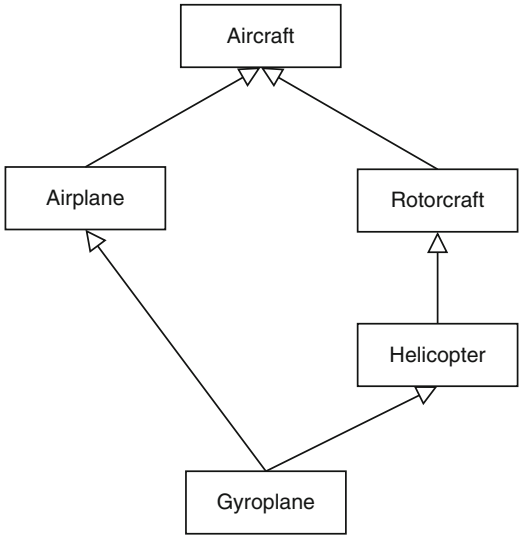


Fig. 1.9 Multiple inheritance

References

1. K. Arnold, J. Gosling, D. Holmes, *The Java Programming Language*, 4th edn. (Addison-Wesley, Boston, 2005)
2. G. Bracha, M. Odersky, D. Stoutmire, P. Wadler, Making the future safe for the past: adding genericity to the Java programming language, in *Proceedings of OOPSLA 1998* (ACM, New York, 1998)
3. K. Bruce, *Foundations of Object-Oriented Languages* (MIT Press, Massachusetts, 2002)
4. P. Canning, W. Cook, W. Hill, W. Olthoff, J. Mitchell, F-bounded polymorphism for object-oriented programming languages, in *Proceedings of Functional Programming Languages and Computer Architecture* (ACM, New York, 1989), pp. 273–280
5. W.R. Cook, A proposal for making Eiffel type safe. Comput. J. **32**, 305–311 (1989)
6. J. Gosling, B. Joy, G. Steel, G. Bracha, *The Java Language Specification*, 3rd edn. (Prentice Hall, New Jersey, 2005)
7. M. Odersky, P. Wadler, Pizza into Java: translating theory into practice, in *Proceedings of POPL 1997* (ACM, New York, 1997), pp. 146–159
8. J. Solorzano, S. Alagić, Parametric polymorphism for Java^{TM}: a reflective solution, in *Proceedings of OOPSLA '98*, (ACM, New York, 1998), pp. 216–225

Chapter 2
Assertions

A distinctive feature of this book is that types are preferably equipped with assertions. Assertions are the topic of Sect. 2.1. The basic notions related to assertions are introduced in Sect. 2.1.1. This section also specifies the basic reasoning rules such as those that apply to assignments, conditional statements, and loops. Methods equipped with preconditions and postconditions are discussed in Sect. 2.2.1. Loop invariants are the topic of Sect. 2.2.2 and a verification technique for proving loop termination in Sect. 2.2.3. Class assertions called object invariants are the topic of Sect. 2.2.4. The implications of the developed apparatus of assertions on parametric types are discussed in Sect. 2.2.5. Finally, in Sect. 2.2.6 we revisit the notion of subtyping in the presence of assertions and explain the all important compatibility requirement for classes called behavioral subtyping.

In Sect. 2.3 we show how assertions are specified in a sample application that makes use of a practical tool called Code Contracts.

2.1 Declarative Specifications

2.1.1 Assertions

Assertions are declarative, logic-based specifications of programs. For a simple or complex program statement S such a specification in the Hoare style has the form P{S}Q. In this notation P is a precondition for S and Q is a postcondition for S. A graphical representation is given in Fig. 2.1.

The meaning of the notation P{S}Q is that if the assertion P holds before the statement S is executed, upon completion of execution of S the assertion Q will hold. This statement does not say anything about whether the execution of S will terminate if the precondition P holds before execution of S. This is why this style of specification is called partial correctness.

S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0_2

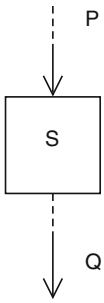


Fig. 2.1 Partial correctness

A sample specification given below is summation of elements of an integer array a between given indices i and j. The precondition requires that the indices i and j are within the range of indices of the array a. The postcondition specifies that the variable s contains the sum of elements of the array a in the specified segment.

$(0 \leq i) \wedge (i \leq j) \wedge (j < a.Length);$
{int s = 0; int n=i;
while (n ≤ j)
s += a[n]; n++;
}
s = sum{int k ∈ (i..j)); a[k] };

In the above example (i..j) denotes an interval $i \leq k \leq j$. sum{int k ∈ (i..j)); a[k]} stands for the sum of all a[k] for k in the specified interval. This notation comes from Spec#. s + = a[n] stands for s=s+a[n] and likewise n++ stands for n=n+1 where = stands for the assignment. Recall that the equality symbol = is overloaded. It is used with its standard mathematical meaning in assertions, as well as the symbol for the assignment statement as in Java and C#.

In order to reason about programs, we have to establish the reasoning rules for simple statements S, and then for the statement composition rules that may be applied to construct complex (composite) S. The most basic such rule applies to the assignment statement and it has the following form:

$$P[e/x]\{x=e\}P$$

where x is a variable, e is an expression, P is an assertion, and P[e/x] denotes P in which all free occurrences of x are replaced by e. Free occurrences are those that are not quantified. In order to evaluate a formula, free occurrences must be assigned specific values.

For example,

$x \geq 0$ {x=x+1} $x > 0$

Here P is $x > 0$, P[x+1/x] is $x+1 > 0$, i.e. $x \geq 0$.

2.1.2 Basic Reasoning Rules

Consider now a conditional statement represented in Fig. 2.2 in which B is a boolean expression, S1 and S2 are statements, and P and Q are assertions. If P holds before execution of this conditional statement and B evaluates to true, $P \wedge B$ will hold before execution of S1. In order for Q to hold after execution of S1, we must have $P \wedge B$ {S1}Q. Following the same logic we get the condition $P \wedge \neg B$ {S2}Q for the other branch. This leads to the following reasoning rule in the form of a Horn clause. The horizontal bar stands for implication. The premises are written above the implication line separated by commas (denoting conjunction) and the conclusions below the line. The notation for the standard logical connectives is $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication) and $\neg$ (negation).

$$\frac{P \wedge B \ \{S1\}\ Q,\ P \wedge \neg B \ \{S2\}Q}{P \ \{\textbf{if}\ (B)\ S1\ \textbf{else}\ S2\ \}\ Q}$$

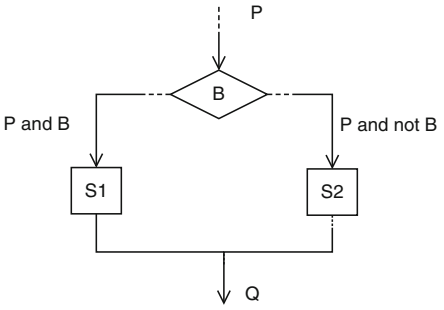


Fig. 2.2 Conditional statement

For example,

$(x \neq 0)$ {**if** $(x > 0)$ y=x **else** y= -x } $(y > 0)$

In this case we have:

P: $(x \neq 0)$
Q: $(y > 0)$

$P \wedge B$: $(x \neq 0) \wedge (x > 0)$
$P \wedge \neg B$: $(x \neq 0) \wedge (x \leq 0)$
$(x \neq 0) \wedge (x > 0) \Rightarrow (x > 0)$
$(x \neq 0) \wedge (x \leq 0) \Rightarrow (x < 0)$

The two premises in the reasoning rule for the conditional statement are:

$(x > 0)\{y{=}x\}\ (y > 0)$
$(x < 0)\ \{y{=}\ {-}x\}\ (y > 0)$

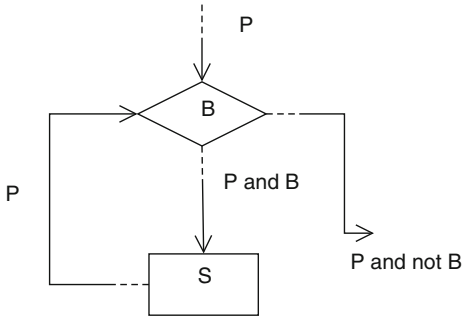
The above two statements are easily established by applying the rule for the assignment statement. Applying the rule for the conditional statement we obtain the desired result:

$(x \neq 0)\{$**if** $(x > 0)\ y{=}x$ **else** $y{=}\ {-}x\ \}\ (y > 0)$

Reasoning rules for loops reveal that the essence of any loop is an assertion called loop invariant. This was no small discovery when it was first observed. This observation says that while values of variables are changing in a loop, it is the relationships among those variables that remain the same that explains what the task of the loop is. So a dynamic behavior of a loop is captured by static relationships that hold in the loop regardless of changing values of program variables.

Consider the while loop in Fig. 2.3 in which B is a boolean expression, S is a statement, and P is an assertion. Assume that P holds when the loop is entered and that one execution of the loop body does not affect the truth value of P, i.e., $P \wedge B\ \{S\}\ P$. This implies that no matter how many times the loop body S executes, P will hold (remains invariant) at the points to which it is attached in Fig. 2.3.

Fig. 2.3 While loop



When the loop terminates (if it does) we will have $P \wedge \neg B$, hence the rule for the while statement:

$$\frac{P \wedge B\ \{S\}\ P}{P\ \{\textbf{while}\ (B)\ S\ \}\ P \wedge \neg B}$$

The following example illustrates this reasoning rule:

// precondition: $x \geq 0$;
{r=0;
while $((r+1)*(r+1) \leq x)$
// invariant: $r*r \leq x$;
{ r=r+1; }
}
// postcondition: $r*r \leq x \wedge x < (r+1)*(r+1)$

Note that the loop invariant $r*r \leq x$ holds when the loop is entered since at that point $x \geq 0$ and r=0. For one execution of the loop body we have

$(r * r \leq x) \wedge (x \geq (r+1) * (r+1))$
{ r=r+1; }
$(r * r) \leq x$

The rule for the assignment statement requires to establish that $((r * r) \leq x)[r+1/r]$ holds before execution of the assignment r=r+1. This amounts to $(r+1) * (r+1) \leq x$ which is exactly what we have.

The loop terminates when $x < (r+1) * (r+1)$, hence upon termination we have:

$(r * r) \leq x < (r+1) * (r+1)$

Following a similar reasoning we obtain the reasoning rule for the *Do* loop in Fig. 2.4:

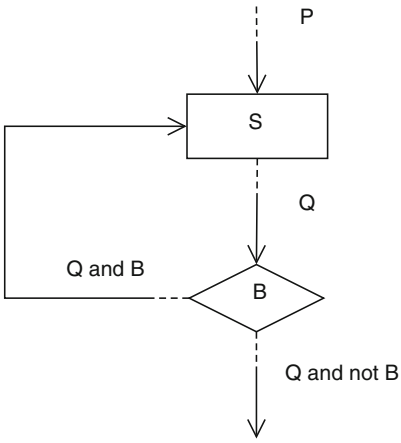
$$\frac{P\ \{S\}\ Q,\ Q \wedge B \Rightarrow P}{P\ \{\textbf{do}\ S\ \textbf{while}\ (B)\}\ Q \wedge \neg B}$$

Assume that P holds upon entrance to the *Do* loop as the above rule requires. Likewise, let P{S}Q holds. If B evaluates to true upon execution of S, $Q \wedge B$ will hold and the loop will continue. In order for P to hold before S is again executed, we must have $Q \wedge B \Rightarrow P$. When the loop terminates (if it does) we will have $Q \wedge \neg B$.

The following example illustrates the above reasoning rule. In this example a stands for an integer array and n:a.Length stands for the interval $0 \leq n < a.Length$ (range of indices of the array a).

// precondition: a.Length ≥ 0
{int n = a.Length;
do
// invariant: $(0 \leq n) \wedge (n \leq a.Length)$;

Fig. 2.4 Do loop




```
// invariant: ∀ { int i (n ≤ i) ∧ (i < a.Length); a[i] ≠ key};
 { n=n-1;
      if (n < 0) {
      break;
 }
 }  while (a[n] ≠ key);
}
// postcondition (n ≥ 0) ⇒ (a[n]=key)
```

When the loop is first entered the invariant $(0 \leq n) \wedge (n \leq a.Length)$ clearly holds because n=a.Length which is a nonnegative integer. This invariant also holds during the loop execution because as soon as $n < 0$ the loop is exited.

The other invariant $\forall$ { int i $(n \leq i) \wedge (i < a.Length)$; a[i] $\neq$ key} also holds because the interval $(n \leq i) \wedge (i < a.Length)$ is initially empty since n=a.Length. This invariant and a[n]$\neq$ key implies that the invariant holds after the assignment n=n-1. The loop continues while the interval satisfies the condition a[n] $\neq$ key. In every execution of the loop body this interval is extended by decreasing n.

There are two possibilities for loop termination. Either $n < 0$ and hence there is no element a[n] such that a[n]=key, or a[n]=key for some n in the range of indices of the array a. In the latter case the desired element is located as indicated by the postcondition.

2.2 Object-Oriented Assertions

The specification style P{S}}Q leads naturally to specification of methods in terms of preconditions and postconditions. Method preconditions and postconditions are illustrated below for a parametric interface *Collection<T>*. The *requires* clause specifies the method precondition, and the *ensures* clause specifies the method postcondition.

```
public interface Collection<T>
{ public boolean contains(T obj);
  public void add(T obj)
  ensures this.contains(obj);
  public void remove(T obj)
  requires this.contains(obj);
// ...
}
```

Class invariants are constraints that must be satisfied by all object states outside of method execution. If the proper information hiding discipline is exercised this means that class invariants must hold for all object states visible to the clients. Class invariants are illustrated below for a parametric interface *Set<T>*. The interface *Set<T>* extends the previously given interface *Collection<T>* by two additional methods *union* and *intersection* along with the associated constraints, that are in fact class invariants.

```
public interface Set<T>
                        extends Collection<T>
{ public Set<T> union(Set<T> S);
  public Set<T> intersection(Set<T> S);
  // ...
invariant
(∀ Set<T> this, S; ∀ T: obj)
(this.union(S).contains(obj) ⇐ this.contains(obj);
this.union(S).contains(obj) ⇐ S.contains(obj);
this.intersection(S).contains(obj) ⇐ this.contains(obj) ∧ S.contains(obj));
// ...
}
```

Note that the methods and the constraints of the interface *Collection<T>* are inherited in *Set<T>*. The logic paradigm used in the above example is Horn clause logic, which is the basis for logic programming languages. The symbol ⇐ denotes implication and comma denotes conjunction. This logic was chosen because of its well-known nice properties: well-defined formal semantics (in which the monotonicity plays a critical role), and the existence of an execution model tied to the formal semantics.

However, Horn clause logic has its limitations. It cannot express negative information. This is why it would not be possible to specify the precondition assertion in the interface *Set<T>* which requires that the element to be inserted does not already belong to the receiver set. Likewise, the postcondition assertion which guarantees that the removed element does not belong to the receiver set, cannot be expressed in Horn clause logic.

The availability of assertions allows specification of semantics (or at least a part of it) in a declarative and formal manner. This way the behavioral properties of objects of a class are communicated to the clients with no need to investigate procedurally decomposed methods. This also allows addressing the issue of behavioral compatibility in the inheritance hierarchies.

The rules of the type system guarantee that when an object of type *Set<T>* is substituted where an object of the type *Collection<T>* is expected, type safety will be guaranteed. This holds for any particular type *T*. This means that an object will never get a message for which it does not have a method with a correct signature. But an object-oriented type system does not address the issue of method semantics.

In fact, in a typical, full-fledged object-oriented language (such as Java or C#), it is possible to redefine the semantics of an inherited method in any way, as long as its signature satisfies the rules of the type system. This leads to all kinds of semantic incompatibilities. For example, suppose that in the interface *Set<T>* we specify the precondition for the method *add* which requires that the element to be added does not already belong to the receiver set. This creates an immediate behavioral incompatibility with the super interface *Collection<T>*. Indeed, collections include bags, and there is no such requirement for adding an element to a bag. The formal rules for behavioral compatibility are expressed by the notion of behavioral subtyping to be discussed further in the sections that follow.

2.2.1 Preconditions and Postconditions

Examples of assertions that follow are based on Spec#. The first three examples are from [11]. Method preconditions and method postconditions are further illustrated by the method *exchangeElements* that exchanges elements of an integer array at two given indices. The precondition is a conjunction of two **requires** clauses that specify that the given indices must be in the range of indices for the given array. The postcondition is a conjunction of two **ensures** clauses that specify the outcome of the method invocation. The **ensures** clauses refer to the state before method execution using the operator **old**. This makes it possible to specify that in the state after execution of the method *exchangeElements*, the values of the array at the given indices were indeed swapped.

```
void exchangeElements(int[] a, int i, int j)
requires (0 ≤ i) ∧ (i < a.Length);
```

```
requires (0 ≤ j) ∧ (j < a.Length);
modifies a[i], a[j];
ensures a[i] = old(a[j]);
ensures a[j] = old(a[i]);
{int temp;
 temp = a[i];
 a[i] = a[j];
 a[j] = temp;
}
```

The **modifies** clause that appears in this specification is called the frame constraint. It specifies that only array elements a[i] and a[j] will be affected by the method invocation. All other objects in the frame of this method remain the same.

A distinctive feature of the Spec# technology and its spin-offs (such as Code Contracts) is static verification of assertions. Code equipped with assertions is compiled by a verifying compiler. In addition to parsing and type checking, the compiler verifies that the code actually satisfies the assertions. This represents a huge progress in comparison with the techniques that preceded Spec# such as Eiffel and Java Modeling Language (JML) that are based on dynamic checking of assertions. In those technologies violation of assertions will be detected only at run-time, requiring run-time actions (exception handling) to deal with assertion violation. This has nontrivial implications on run-time efficiency and reliability of object-oriented programs.

2.2.2 Loop Invariants

In the specification of the method *ArraySum* that follows the postcondition refers to the result of the method invocation using the keyword **result**. This example contains specification of the loop invariant of the **for** statement in the body of the method *ArraySum*. The first invariant requires that the control variable n is always in the required range. Most importantly, the second invariant specifies that the variable s contains the sum of values of the array a at indices up to n. In this invariant the Spec# notation for the half open interval is used. i ∈ (0: a.Length) stands for the interval $0 \leq i < a.Length$.

The invariant $n \leq a.Length$ follows from the fact that the loop body will be executed if $n < a.Length$. After the assignment n=n+1 this amounts to $n \leq a.Length$ before the next iteration of the loop body.

The invariant s = sum {int i ∈ (0: n); a[i] } holds initially because the interval (0:n) is empty. This invariant and $n < a.Length$ before the assignments s = s + a[n]; n=n+1 imply that the invariant will hold after these assignments. When the loop terminates, n will be equal to the length of the array a, hence the summation is completed and the result is returned.

The method *ArraySum* makes use of non-null object types. If T is an object type, then T! denotes a type that cannot contain a null reference. This is a distinctive feature of the Spec# type system. So the argument of *ArraySum* is required to be a non null reference to an array object. Violation of this requirement will be detected statically by the Spec# verifying compiler.

```
public int ArraySum(int[]! a)
ensures result = sum {int i ∈ (0: a.Length); a[i]};
{int s = 0;
 for (int n = 0; n < a.Length; n++)
      invariant n ≤ a.Length;
      invariant s = sum {int i ∈ (0: n); a[i] };
{s += a[n];}
 return s;
}
```

2.2.3 Termination

The core idea for proving termination is to come up with an integer expression (variant function) that remains non-negative throughout the loop execution and its value is strictly decreased with every execution of the loop body. That can happen only a finite number of times, hence the loop must terminate. This is illustrated for the while loop in Fig. 2.5 where vf stand for the variant function.

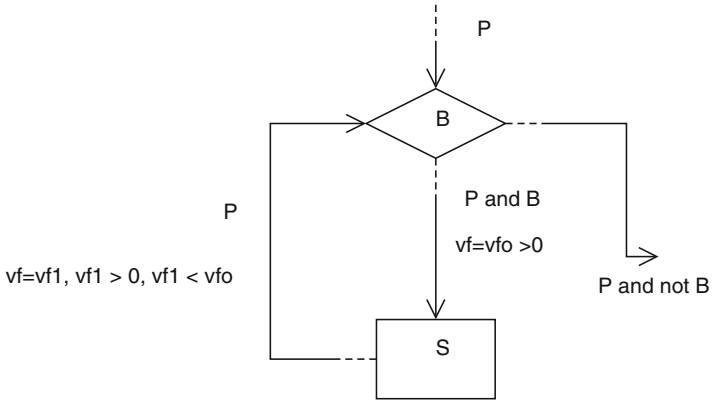


Fig. 2.5 Proving termination

Verifying termination is illustrated in the method *ArrayRangeSum* given below. With the assertions specified in the method *ArrayRangeSum* the static verifier of

Spec# will actually be able to verify that the above two conditions are satisfied for variant defined as *variant=j-n*.

Consider first the invariant (i ≤ n) ∧ (n ≤ j). This invariant holds initially because n=i so that the invariant amounts to (i ≤ i) ∧ (i ≤ j) which is implied by the precondition (i ≤ j) of the method *ArrayRangeSum*. Before execution of the loop body we have (i ≤ n) ∧ (n ≤ j) ∧ (n < j). This implies (i ≤ n+1) ∧ (n+1 ≤ j). This assertion before the assignment n=n+1 implies that (i ≤ n) ∧ (n ≤ j) holds after this assignment.

The invariant s = sum {int k ∈ (i: n); a[k]} holds initially because the interval (i: n) is empty since n=i and s=0. This invariant before the assignments s=s+a[n]; n=n+1 implies that it will hold after these assignments.

The third invariant that is verified statically ensures that the variant function is nonnegative. $j - n \geq 0$ holds initially because n=i so that this invariant amounts to $j - i \geq 0$, i.e., i ≤ j which is a precondition of the method *ArrayRangeSum*. Before the assignment n=n+1 we have $j - n \geq 0$ and j > n, i.e., j ≥ n and j > n which applies j ≥ n+1, i.e., $j - (n+1) \geq 0$. This assertion before the assignment n=n+1 implies $j - n \geq 0$ after this assignment.

The **assert** statement that is also statically verified (and dynamically enforced) ensures that the variant function is strictly decreased by an execution of the loop body. Since n is increased in every execution of the loop body, $j - n$ is strictly decreased as stated in the **assert** statement. This proves that the loop will terminate.

```
public int ArrayRangeSum(int[]! a, int i, int j)
requires (0 ≤ i) ∧ (i ≤ j) ∧ (j ≤ a.Length);
ensures result = sum {int k ∈ (i: j); a[k] };
{int s = 0; int n=i;
 while (n < j)
 invariant (i ≤ n) ∧ (n ≤ j);
 invariant s = sum {int k ∈ (i: n); a[k]};
 invariant 0 ≤ j - n;
 { int variant = j - n;
 s += a[n]; n++;
 assert j - n < variant;
 }
return s;
}
```

2.2.4 Object Invariants

In addition to method preconditions and postconditions, object-oriented assertion languages specify consistent object states by object invariants. An object invariant is an assertion that an object is required to satisfy in all visible object states.

A visible object state is an object state outside of a method execution. During method execution object invariants are typically violated. But an object invariant must hold after a constructor, before method execution, and after method execution. A simple example is given below in the class *Accumulator*.

```
class Accumulator {
 int count;
 invariant count ≥ 0;
 public Accumulator()
 { count = 0; }
 public void Inc(int increase)
 modifies count;
 ensures count = old( count)+increase;
 { expose (this) {
       count = count + increase;
       }
 }
}
```

The object invariant specifies that the accumulator value must be non-negative. Note that the object invariant is first established by the constructor of this class. The object invariant may be temporarily violated in the method body, but it must be reestablished upon termination of the method. The **expose** block (as in Spec#) indicates this situation so that the verifier will not enforce the object invariant in this block. Upon exit of the expose block the loop invariant will be statically enforced. This is why the accumulator class will not be verified because the actual parameter *increase* of the method *Inc* may be negative. The verifier will indicate that there is no guarantee that the invariant will hold upon completion of the **expose** block. This problem is solved by specifying a precondition for the method *Inc* as follows:

```
 public void Inc(int increase)
 modifies count;
 requires increase > 0;
 ensures count = old( count)+increase;
 { expose (this) {
       count = count + increase;
       }
 }
```

With the above precondition, the **expose** block in the above method is not necessary any more. This example also shows the tension that exists between two requirements: components of the object state should be private, but specifications should be public. However, those specifications often refer to the object states, such as the object invariant in the *Accumulator* class. A technique for resolving this controversy in assertion languages is explained in the next section.

2.2.5 Assertions for Collections

A subtlety related to object invariants is that they should be public, i.e., visible to the clients of a class. However, object invariants refer to the components of the object state that should be private. This controversy is resolved by making components of the object state public only for specification purposes as in the class *CollectionSpec* given below. The notation is based on Spec#.

The state of a collection object is represented as a list. Methods such as *contains*, *add* and *remove* are implemented invoking the appropriate methods of the class *List*.

The method *contains* is an example of a pure method. A pure method is a pure function. As such it does not have any side effects on the object state. This is why pure methods can be used in specifications. The corresponding method *Contains* of the class *List* is also declared as pure. The postcondition of the method *contains* specifies that the result of this method will be true if the underlying list contains the argument object of this method.

```
public abstract class CollectionSpec<T>{

 [SpecPublic] protected List<T!>!
      elements = new List<T!>();
 [SpecPublic] protected int size = 0;

 invariant elements.Count = size;
 invariant size ≥ 0;

 [Pure] public bool contains(T! x)
 ensures result = elements.Contains(x);
 { code }

 public virtual void add(T! x)
 modifies elements;
 ensures this.contains(x);
 { code }

 public virtual void remove(T! x)
 modifies elements;
 requires this.contains(x);
 { code }
}
```

The postcondition of the method *add* is that the underlying list contains the argument object of this method. The frame constraint in the **modifies** clause of the method *add* specifies that this method modifies the underlying list of the collection elements.

The precondition of the method *remove* is that the argument object belongs to the collection. The postcondition of the method *remove* does not specify that the removed element does not belong to the collection. That would be true if the collection is a set, but not necessarily true is the collection is a bag.

The list of collection elements is required to be non-null, and so are the elements. The argument of the methods *contains*, *add* and *remove* are required to be non-null as well.

In the specification of the class *BagSpec* a pure method *occurrences* specifies the number of occurrences of an element of a bag. Its postcondition makes use of a predefined combinator *Count* of Spec# that returns the number of elements that are equal to the argument of the method *occurrences*.

The methods *add* and *remove* are overridden. The postcondition of the method *add* ensures that the number of occurrences of the added element has been increased by one. The precondition of the method *remove* is inherited from the class *CollectionSpec* and it requires that the element to be removed belongs to the bag. The postcondition of the method *remove* ensures that the number of occurrences of the element removed from the bag is decreased by one.

The methods *union* and *intersection* are defined as pure methods that construct new objects representing the results of these operations. The postconditions of these methods follow the definitions of the union and intersection of bags as defined in the algebra of bags. Note the C# notation for inheritance in which the symbol : corresponds to the Java keyword **extends**.

The first *ensures* clause of the method *union* guarantees that the resulting bag consists of elements that belong to one or to both of the bags whose union is constructed. For elements that belong to both bags, the second *ensures* clause specifies the number of occurrences of those elements in the resulting bag. Likewise, the first *ensures* clause of the *intersection* method guarantees that the resulting bag consists of elements that belong to both of the input bags. The second *ensures* clause specifies the number of occurrences of those elements.

```
public class BagSpec<T> : CollectionSpec<T>
{
 [Pure] public int occurrences(T! x)
 ensures result = Count{int i ∈ (0: elements.Count);
      x.equals(elements[i])};
 { code }

 public override void add(T! x)
 modifies elements;
 ensures occurrences(x) = old(occurrences(x)) + 1;
 { code }

 public override void remove(T! x)
 modifies elements;
 ensures occurrences(x) = old(occurrences(x)) - 1;
```

```
 { code }

[Pure] public BagSpec<T!>! union(BagSpec<T!>! other)
ensures result.contains(x) ⇔
      this. contains(x) ∨ other.contains(x);
// type of x inferred as T
ensures this.contains(x) ∧ other.contains(x) ⇒
result.occurrences(x)=
      Max{this.occurrences(x), other.occurrences(x)};
{code }

[Pure] public BagSpec<T!>! intersection(BagSpec<T!>! other)
ensures result.contains(x) ⇔
      this.contains(x) ∧ other.contains(x);
ensures result.occurrences(x)=
      Min{this.occurrences(x), other.occurrences(x)};
{code }
```

In the above specifications of the methods *union* and *intersection* x is assumed to be universally quantified and its type is inferred to be T. Spec# limits universal and existential quantification to finite integer intervals in order to accomplish static and automatic verification. That requirement can be satisfied as follows:

```
[Pure] public BagSpec<T!>! union(BagSpec<T!>! other)
ensures ∀ { T x ∈ result.elements;
      this. contains(x) ∨ other.contains(x)};
ensures ∀ { T x ∈ result.elements;
 this.contains(x) ∧ other.contains(x) ⇒
result.occurrences(x)=
      Max{this.occurrences(x), other.occurrences(x)}
      };
{code }
```

Finally, a detailed specification that explicitly operates on sequences of elements would look like this:

```
[Pure] public BagSpec<T!>! union(BagSpec<T!>! other)
ensures ∀ {int i ∈ (0: result.elements.Count);
 result.elements[i] ∈ elements ∨
 result.elements[i] ∈ other.elements; }
ensures ∀ {int i ∈ (0: result.elements.Count);
      int j ∈ (0: elements.Count);
      int k ∈ (0: other.elements.Count);
      result.elements[i].equals(elements[j]) ∧
      result.elements[i].equals(other.elements[k]) ⇒
      result.occurrences(elements[i])=
```

```
      Max{occurrences(elements[j]), other.occurrences(elements[k])}
      };
 {code }

[Pure] public BagSpec<T!>! intersection(BagSpec<T!>! other)
ensures ∀ {int i ∈ (0: result.elements.Count);
 result.elements[i] ∈ elements ∧
 result.elements[i] ∈ other.elements};
ensures ∀ {int i ∈ (0: result.elements.Count);
 int j ∈ (0: elements.Count);
 int k ∈ (0: other.elements.Count);
      result.elements[i].equals(elements[j]) ∧
      result.elements[i].equals(other.elements[k]) ⇒
      result.occurrences(elements[i])=
      Min{occurrences(elements[j]), other.occurrences(elements[k])}
      };
{code }
```

The class *OrderedBagSpec* makes use of F-bounded polymorphism that guarantees that the actual type parameter substituted for T will be equipped with the method *CompareTo*. F-bounded polymorphism is expressed in the C# notation. The invariant guarantees that the bag is ordered.

```
public class OrderedBagSpec<T> : BagSpec<T>
      where T : IComparable<T>
{
invariant ∀ {int i ∈ (0: elements.Count),
                int j ∈ (0: elements.Count);
      (i ≤ j ) ⇒ elements[i].CompareTo(elements[j]) ≤ 0 }
// . . .
};
```

2.2.6 Behavioral Subtyping

Object-oriented assertion languages support a particular form of behavioral subtyping that guarantees behavioral compatibility of objects of a subclass when viewed as objects of the superclass. This discipline, illustrated by the example that follows, does not allow changes of the inherited preconditions. At the same time the postconditions and object invariants are allowed to be strengthened by addition of new assertions.

An example is given below. The class *Stock* has a method *setValue* that has no precondition. Its postcondition ensures that the stock value has been assigned correctly.

```
public class Stock {
 private String code;
 public String getCode()
 { return code; }
 invariant this.getCode() ≠ null;
 [SpecPublic] protected int value;
 public int getValue(){ return value; }
 public virtual void setValue(int v)
 ensures value=v;
 { value=v; }
}
```

The class *BackupStock* overrides the method *setValue*. The overridden method *setValue* inherits the postcondition from the class *Stock* and strengthens it by conjunction with an additional postcondition. The method precondition (trivially true) remains the same. The invariant is inherited from the class *Stock*.

```
public class BackupStock: Stock {
 [SpecPublic] protected int backup;
 public override void setValue (int v)
 ensures backup =old(value);
 { backup=value; value=v; }
}
```

If an instance of *BackupStock* is substituted where a *Stock* instance is expected, a user of a *Stock* object will see no unexpected behavior of a *Stock* object. Invoking the method *setValue* requires no precondition. The postcondition that holds after execution of the overridden method *setValue* is

$$(\text{value}=\text{v}) \wedge (\text{backup} = \textbf{old}(\text{value}))$$

which implies (value=v). This means that a user of a *Stock* object will see nothing unusual if the receiver object happens to be a *BackupStock* object.

Let us now attempt to override the method *setValue* as in the class *GrowingStock* given below. The keyword *base* in C# corresponds to the keyword *super* in Java and represents a reference to the superclass object. The precondition of the overridden method is strengthen. This means that correct invocation of the overridden method requires a precondition (value $\leq$ v). If an instance of *GrowingStock* is substituted in place of an object of type *Stock*, a user of a *Stock* object will invoke the method *setValue* with no precondition. This will create an exception inexplicable for a user of a *Stock* object. This is why preconditions of overridden methods must remain the same.

```
public class GrowingStock: Stock {
 public override void setValue(int v)
 requires value ≤ v; //not allowed
 { base.setValue(v);}
}
```

2.3 Sample Application

Promoting assertion languages comes with several challenges. One of them is that a full-fledged assertion language of the kind presented in the preceding sections requires significant changes of the underlying object-oriented programming language and a complex implementation technique if static verification of assertions is supported. In this section we look at an application (tournament management) in which Code Contracts is used to specify assertions with no changes to the underlying C# language.

All assertions of Code Contracts appear as invocations of methods of the class *Contracts*. The actual assertions are specified as boolean expressions, arguments of these methods. This approach leads to dynamic checking of assertions as methods of the class *Contracts* are executed. Code Contracts also has static checking capabilities that are being extended as the system develops.

The tournament management application has several types of users, two of which we specify. A tournament has a manager and a list of players. This is represented in Fig. 2.6. The notation in this figure indicates that a tournament has a unique manager and it is associated with multiple players.

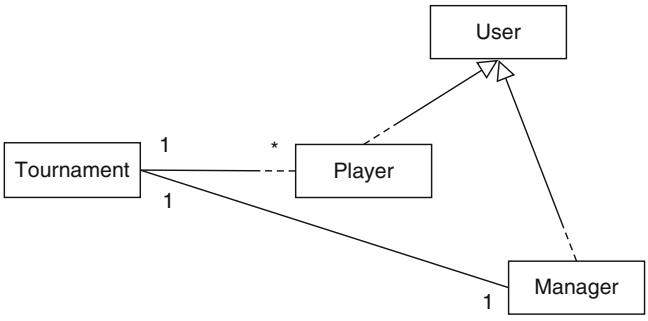


Fig. 2.6 Tournament management application

The class *Tournament* given below specifies the features of tournament objects. It contains self explanatory fields, properties and a constructor.

```
class Tournament {
 String name;
 Manager manager;
 List<Player> players = new List<Player>();
 // other fields
 // constructor
 public String Name
 {    get { return name; }
```

```
        set { name = value; }
  }
  // other poperties
}
```

Generic properties of a tournament user are specified below in a class *User* in the C# style. A property in C# is a pair of methods. The method *get* returns the value of a property and the method *set* assigns a value to a property. Note that **private** is the default accessibility for members of a class in C#. So in the above example the underlying (backing) fields are private and the properties are public.

This class shows how Code Contracts specifies object invariants. An object invariant is specified in a distinguished method marked with a special attribute [ContractInvariantMethod]. This method contains calls of the method *Invariant* of the class *Contracts*. Code Contracts enforces object invariants after execution of public methods. In the class *User* the invariants are that the user name cannot be null and the user ID number cannot be null.

```
abstract class User {
  String IDNum;
  String name;
  String role;
  [ContractInvariantMethod]
  void ObjectInvariant() {
        Contract.Invariant(this.UserName ≠ null);
        Contract.Invariant(this.ID ≠ null);
  }
  public String ID
  {     get { return IDNum; }
        set { IDNum = value; }
  }
  public String UserName
  {     get { return name; }
        set { user = value; }
  }
  // Role property
}
```

There are two subtypes of the type *User* that we define: players and tournament managers. The type *Player* introduces an additional invariant requiring that the number of wins of a player must be nonnegative. Other object invariants are inherited from the class *User*.

```
class Player : User {
 int winCount;
 public int WinCount
 {    get { return winCount; }
      set { winCount = value; }
 }
 //other properties
 [ContractInvariantMethod]
 void ObjectInvariant() {
      Contract.Invariant(this.WinCount ≥ 0);
 }
 // constructor and other methods
}
```

The class *Manager* strengthens the object invariant. In addition to the invariants inherited from the class *User*, the class *Manager* requires that the role of a tournament manager contains the word "MANAGER".

```
class Manager : User {
 // fields
 [ContractInvariantMethod]
 void ObjectInvariant() {
{ Contract.Invariant(Role.ToUpper().Contains("MANAGER"));
 }
// methods
}
```

Consider now a method for adding a new player to a tournament. This would be a method of the overall application class *TournamentManagement* that we do not show. The method *addPlayer* requires a pure method *playerRegistered* that checks whether the player to be added is already in the list of players of the given tournament. A pure method is marked with a special C# attribute [Pure].

```
[Pure]
public boolean playerRegistered(Player newPlayer, Tournament tournament) {
 foreach (Player player  in tournament.players)
 { if (newPlayer.UserName.ToUpper().Equals(player.UserName.ToUpper()))
      return true;
 }
 return false;
}
```

The preconditions of the method *addPlayer* require that the given player and the tournament must be non null.

```
Contract.Requires(newPlayer ≠ null);
Contract.Requires(tournament ≠ null);
```

In addition, the third precondition requires that the player does not already participate in the tournament.

```
Contract.Requires(¬ playerRegistered(newPlayer, tournament));
```

The postconditions are that the player participates in the tournament (i.e., it has been added to the list of players).

```
Contract.Ensures(playerRegistered(newPlayer, tournament));
```

In addition, the postconditions require that the number of players has been increased by one.

```
Contract.Ensures((tournament.Players.Count) =
    (Contract.OldValue(tournament.Players.Count) + 1));
```

A reference to the number of players before the method execution is specified by invocation of the method *OldValue* of the class *Contracts*. *Players* is a property whose underlying field is *players*.

```
public void addPlayer(Player newPlayer, Tournament tournament) {
Contract.Requires(newPlayer ≠ null);
Contract.Requires(tournament ≠ null);
Contract.Requires(¬ playerRegistered(newPlayer, tournament));
Contract.Ensures(playerRegistered(newPlayer, tournament));
Contract.Ensures((tournament.Players.Count) =
 (Contract.OldValue(tournament.Players.Count) + 1));
tournament.Players.Add(newPlayer);
}
```

Specification of assertions that require universal and existential quantification is accomplished by using methods *ForAll* and *Exists* of the class *Contracts*.

For example, universal quantification is used in order to specify an assertion that the list of players contains no null entries.

```
Contract.Invariant(Contract.ForAll(Players, p ⇒ p ≠ null));
```

Existential quantification is used to specify an assertion that the list of players contains at least one player whose total number of wins is greater than zero.

```
Contract.Invariant(Contract.Exists(Players, p ⇒ p.WinCount > 0) );
```

These examples contain C# lambda expressions that are in fact unnamed functions. For example, $p \Rightarrow p \neq$ **null** is a function that takes a player as its argument and returns true if the player is not null. Likewise for function $p \Rightarrow p.WinCount > 0$.

Interfaces in Java and C# suffer from a major contradiction. The only way to understand the specific meaning of the methods of an interface is to look into the method code in the implementing class, contrary of the intent for introducing interfaces. This is why specifying assertions in interfaces is so very important. However, assertions such as object invariants often require the knowledge of the object state which is not available in interfaces. Code Contracts resolves this

situation by specifying assertions of an interface in a special class associated with the interface. Methods of the special contract class are required only to satisfy the compiler. This class is used only to specify the assertions and it is never executed like other classes. But these assertions will be enforced in any class that implements the interface.

The simplified view of the tournament management application can be generalized so that a tournament has a collection of matches. A match has a number of players, and a player participates in a number of matches. This extended view is presented in Fig. 2.7. The associated constraints are elaborated in the exercises.

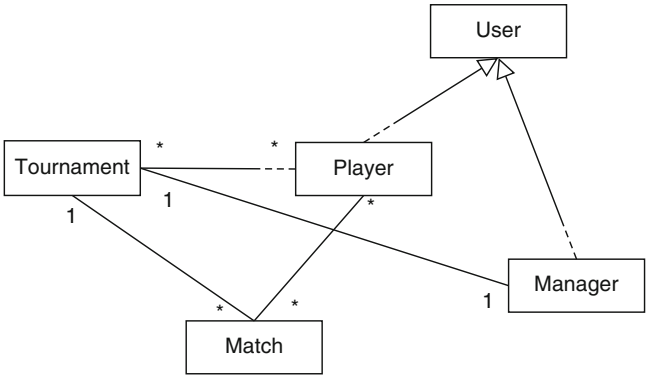


Fig. 2.7 Tournaments with matches

2.4 Bibliographic Remarks

The first widely know object-oriented language with assertions was Eiffel [9]. Eiffel is entirely based on dynamic checking of assertions. The assertion language for Java is JML (Java Modeling Language) [6]. JML is based on dynamic checking of assertions. Our presentation of an object-oriented assertion language is based on Spec# because of its generality and static verification of assertions [10]. Some presented samples of Spec# code are from the tutorial [11] or [7]. Examples of object-oriented assertions for collection types are from [1]. Spec# assertions for collection types are based on the specifications due to Tom Keschl. The notion of behavioral subtyping was introduced in a seminal paper [8]. A spinoff of Spec# is Code Contracts [5] which, although much more limited, is an actual software product. Examples in the section on Code Contracts are based on an application developed by Christopher Hunter.

Object-oriented languages require more sophisticated logics such as temporal or separation logic. Even the logic underlying major object-oriented languages is

implicitly temporal because postconditions make use of the operator that refers to the state before method execution. Foundations of an object-oriented paradigm based on temporal logic are developed in [2] and specific verification techniques for Java-like classes in [4]. Even more general formal theory of classes equipped with constraints in a general logic is published in [3].

2.5 Exercises

1. Following the approach for the proof rules for **while** and **do** loops, specify the formal reasoning rule for the **for** statement as defined in Java and C#. Likewise, specify the formal rule for the C# **foreach** statement.
2. Specify a method that computes square by addition. Specify the precondition, the postcondition, and the loop invariant.
3. Specify a method that operates on a non-null array and performs an operation of your choice on each of its elements. Specify the precondition, the postcondition, and the loop invariant.
4. Specify a method that produces the count of those elements of an array that have a specific property expressed by a predicate. Specify the precondition, the postcondition, and the loop invariant.
5. Specify a method that sorts an integer array using an algorithm of your choice. Specify the precondition, the postcondition and the loop invariant.
6. Specify the method that computes the greatest common divisor of two integers. This method requires the precondition, the postcondition, and the loop invariant. Use a version of the Euclidean algorithm of your choice. Verify termination.
7. Specify the formal reasoning rule for method declaration and method invocation.
8. The rules of behavioral subtyping specify that a subclass may weaken the precondition of a method of the superclass, strengthen its postcondition, and strengthen the invariant of the superclass. Typically, in object-oriented assertion languages the precondition remains the same. Specify formally these rules.
9. Specify a class *Car* and its subclass *SportsCar*. Specify all the required assertions in accordance with the rules of behavioral subtyping.
10. Specify a method that takes an integer array and an integer as parameters. The method performs a liner search of the array and returns true if the array contains the given integer. Verify termination.
11. Specify a method that performs a binary search of an integer array for a given integer value. Specify the precondition, the postcondition and the loop invariant.
12. Complete specification of the class *Tournament* by invariants such as those that determine the maximum number of players, require that the name of the tournament must be non-null, specify the start and end date and their relationship, etc.
13. Specify the method *removePlayer* from a tournament with its precondition and its postcondition.

14. Modify the tournament management application in such a way that a tournament consists of a collection of matches. Specify the assertions required by this extended model.
15. A player can naturally participate in multiple matches of a tournament. Specify the required constraints such as that a player cannot participate in multiple overlapping matches.

References

1. S. Alagić, A family of the ODMG object models, invited paper, in *Proceedings of ADBIS '98*. Lecture Notes in Computer Science, vol. 1691 (1999)
2. S. Alagić, Semantics of temporal classes. Inf. Comput. **163**, 60–102 (2000)
3. S. Alagić, S. Kouznetsova, Behavioral compatibility of self-typed theories, in *Proceedings of ECOOP 2002*. Lecture Notes in Computer Science, vol. 2374 (Springer, New York, 2002), pp. 585–608
4. S. Alagić, M. Royer, D. Crews, Temporal verification theories for Java-like classes, in *Proceedings of the ECOOP FTfJP06 Workshop: Formal Techniques for Java-Like Programs* (2006), http://www.disi.unige.it/person/AnconaD/FTfJP06/
5. Code Contracts, Microsoft Research, http://research.microsoft.com/en-us/projects/contracts/
6. Java Modeling Language, http://www.eecs.ucf.edu/leavens/JML/
7. K.R. Leino, P. Muller, Using Spec# language, methodology, and tools to write bug-free programs, Microsoft Research (2010), http://research.microsoft.com/en-us/projects/specsharp/
8. B. Liskov, J.M. Wing, A behavioral notion of subtyping. ACM TOPLAS **16**, 1811–1841 (1994)
9. B. Meyer, *Object-Oriented Software Construction*, 2nd edn. (Prentice Hall, New Jersey, 1997)
10. Microsoft Corp., Spec#, http://research.microsoft.com/specsharp/
11. R. Monahan, R. Leino, Program verification using the Spec# programming systems, ECOOP 2009 Tutorial, http://www.rosemarymonahan.com/specsharp/papers/ECOOPTutorial.pdf

Chapter 3
Virtual Platform

In this chapter we describe a modern environment that supports a typed object-oriented technology. The components of this environment include reflection, the actual object-oriented virtual machine, and support for persistent objects.

Reflection is the topic of Sect. 3.1. We first describe in Sect. 3.1.2 the core of type—safe object-oriented reflection: the Java class *Class* (C# has the corresponding type *Type*). Classes *Field*, *Method*, and *Constructor* are described in respective Sects. 3.1.3, 3.1.4, and 3.1.5. In Sect. 3.1.6 we show how access and updating of field values is carried out using reflection. In Sect. 3.1.7 we show how methods are invoked using reflection. Creating class objects using an extensible class loader is the topic of Sect. 3.1.8. Finally, in Sect. 3.1.9 we give a general specification of the structure of the Java class files.

Modern typed object-oriented languages such as Java or C# run on an object-oriented virtual machine. A virtual machine is an interpreter providing machine independence and portability of the compiled source code. This is the topic of Sect. 3.2. We first describe the structure of an object-oriented virtual machine in Sect. 3.2.1. Our presentation is an abstraction of the existing very low-level specification of the Java Virtual Machine (JVM). We use assertions to define the semantics of the basic operations of the object-oriented virtual machine. The operation of creating an object is specified in Sect. 3.2.2. The operation of invoking a method is specified in Sect. 3.2.3. The operations of accessing and updating field values are specified in Sect. 3.2.4. Operations that are used to evaluate expressions are defined in Sect. 3.2.5. Operations on arrays (creating, accessing via an index, and updating the element with a given index) are defined in Sect. 3.2.6.

The current generation of object-oriented virtual platforms has no ability to adequately represent and access constraints that are specified for classes. In accordance with a core idea in this book, we show in Sect. 3.3 how an object-oriented virtual platform can be extended to represent and report information about constraints associated with object-oriented types.

S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0_3

An object-oriented platform should support persistent objects, i.e., objects with long lifetimes. Many, if not most, applications feature such objects. Persistent models and mechanisms are discussed in Sect. 3.4. We first define a very general and most attractive model of orthogonal persistence in Sect. 3.4.1 and show its implications using an implemented system and its application. We then describe in Sect. 3.4.2 what kind of a persistent architecture an object-oriented virtual machine should preferably support. This is followed by a description of a much more limited model of persistence based on object serialization like in Java or C#.

3.1 Reflection

3.1.1 Reflective Classes

Reflection allows run-time access to all type information that is the result of compilation. The type information can be introspected through a collection of methods, and run-time actions taken based on the type information discovered at that time. This makes it possible to write very general programs that do not have type information available at the time those programs are compiled.

A simplified Java type hierarchy that includes reflective classes such as *Class*, *Method*, and *Field*, is represented in Fig. 3.1. For the reasons of simplicity and clarity of concepts our presentation follows the original specification of Java Core Reflection. More recent versions include unnecessary complexity that is in part a consequence of a faulty solution for parametric types in Java.

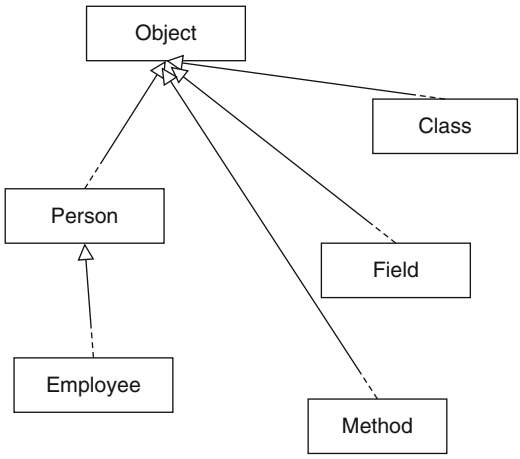


Fig. 3.1 Object, Class, Method, Field

The core class of Java Core Reflection (JCR) is the class *Class*. Objects of the class *Class* contain type information compiled from the classes in the source code. The class *Class* is final, and so are its methods. These methods only introspect and return type information. There are no mutator methods in the class *Class*, i.e., run-time type information cannot be changed. If that was possible, the purpose of the static type system would be completely defeated. The state of class objects is very complex and it is not exposed to the users. Furthermore, class objects are not created by the users. They are created by the class loader based on the compiled information contained in the class files produced by the compiler from the source code. Given an object, its class object is accessed by invoking the method *getClass* of the root class *Object*. From the class object, fields, methods, and constructor objects of that class are accessed using the methods of the class *Class*.

3.1.2 Class Objects

The basic methods of the class *Class* are given in the code below.

```
public final class Class extends Object {
 public Object newInstance()
      throws InstantiationException, IllegalAccessException;
 public boolean isInstance(Object obj);
 public Class getSuperclass();
 public Class[] getClasses();
 public Field[] getFields();
 public Method[] getMethods();
 public Constructor[] getConstructors();
// other methods
}
```

The method *newInstance* creates an object of the given class represented by its class object. The method *isInstance* tests whether the argument object of this method is an instance of the class represented by the receiver class object.

The method *getSuperClass* returns the class object of the superclass. This way the whole hierarchy of superclasses can be traversed all the way to the root class *Object*.

Methods *getFields*, *getMethods*, and *getContructors* return arrays of the field, method, and constructor objects of the class. Specific class members are discovered given specific criteria. The method *getField* returns the field object of a class given the field name. The method *getMethod* returns the method object of a class given its type signature. This means the name of the method and the types of arguments must be supplied. The types of arguments are supplied as an array of the corresponding class objects. The method *getConstructor* requires the same information except for the name which is always the class name.

```
public Field getField(String name)
    throws NoSuchFieldException
public Method getMethod(String name, Class[] parameterTypes)
    throws NoSuchMethodException
public Constructor getConstructor(Class[] parameterTypes)
    throws NoSuchMethodException.
```

Interfaces are also represented by class objects. The method *isInterface* tests whether a class object represents an interface. The interfaces that a class implements are returned as an array of their class objects using the method *getInterfaces*.

```
public boolean isInterface()
public Class[] getInterfaces()
```

Arrays are objects, hence their type information is also represented by class objects. The method *isArray* tests whether a class object represents an array. The method *getComponentType* returns the class object of the components of the array.

```
public boolean isArray()
public Class getComponentType()
```

Subtype relationships are tested at run-time using the method *isAssignableFrom*. This method tests whether the class represented by the argument class object is derived directly or indirectly from the class represented by the receiver class object of this method.

```
public boolean isAssignableFrom(Class fromClass)
```

There are methods that produce a string representation of a class, return the name of a class, and return the access and other modifiers of a class encoded as an integer.

```
public String toString()
public String getName()
public int getModifiers()
```

Class objects are constructed by the class loader. One way of doing this is by invoking the method *forName*. This method takes the full name of the class, locates the class file, and constructs the class object based on the information in the class file. The method *getClassLoader* returns the loader object of a class.

```
public static Class forName(String className)
    throws ClassNotFoundException
public ClassLoader getClassLoader()
```

Simple types, in Java, are also represented by special class objects. The method *isPrimitive* tests whether a class object represents a simple type.

```
public boolean isPrimitive()
```

JCR has specific methods that handle simple types that we will not specify. We will use only object types in our presentation, in part because Java has a wrapper class for each simple type. JCR has automatic conversions from objects of these wrapper classes to the corresponding values of simple types, and the other way around.

Recent editions of Java have obscure reflective capabilities for parametric classes which we do not present. We just point out that a minimal support for parametric classes should include a method *isParametric* that tests whether the class object represents an instantiated parametric class. In addition, a method *getTypeParameters* is needed to discover the actual type parameters of an instantiated parametric class.

```
public boolean isParametric()
public Class[] getTypeParameters()
```

Additional apparatus is needed in order to reflect on the actual parametric class definition. But parametric class definitions should not have class objects, only their instantiations should. This is why we do not discuss this apparatus although both Java and C# have it. The Java view of parametric classes using reflection is particularly confusing.

3.1.3 Field Objects

Given a field object, the class object of that field is accessed by invoking the method *getDeclaringClass*. The name of the field is discovered by the method *getName*. The type of a field is represented by the class object of that type returned by the method *getType*.

Methods of the class *Field* allow getting the value of a field and even updating the value of a field. The method *get* applies to a receiver field object and takes as the argument the object whose field value will be returned by the method *get*. The field of the argument object is accessed based on the information in the field object and its value is returned as an object.

Assigning to a field is performed by the method *set*. Its receiver is the field object, and the argument is an object whose field will be updated. The new field value is supplied as an object (the second argument of the method set). All of this is done with dynamic type checking enforced. In other words, type violations such as assigning the value to a field of a wrong type will create a run-time exception.

```
public final class Field extends Object {
 public Class getDeclaringClass();
 public String getName();
 public Class getType();
 public Object get(Object obj);
      throws NullPointerException, IllegalArgumentException,
      IllegalAccessException;
```

```
 public void set(Object obj, Object value)
      throws NullPointerException, IllegalArgumentException,
      IllegalAccessException;
}
```

3.1.4 Method Objects

The class *Method* has the basic methods *getDeclaringClass*, *getName*, and *getModifiers* like the class *Field*.

Type introspection methods are *getParameterTypes* and *getReturnType*. The parameter types are returned as an array of their class objects. The return type is represented by its class object. The same applies to the exception types that the method can throw.

A method of particular importance is *invoke*. This method allows invoking a method discovered by reflection. The receiver of this method is a method object. The first argument is the object that is the actual receiver of the method to be invoked. The actual parameters are supplied in the second argument as an array of objects. The result is returned as an object. Any type violation such as passing actual parameters of wrong types will be detected by the run-time type system and exceptions will be raised.

```
public final class Method extends Object {
 public Class getDeclaringClass();
 public String getName();
 public int getModifiers();
 public Class getReturnType();
 public Class[] getParameterTypes();
 public Class[] getExceptionTypes();
 public Object invoke(Object obj, Object[] args)
      throws NullPointerException, IllegalArgumentException,
      IllegalAccessException, InvocationTargetException;
}
```

3.1.5 Constructor Objects

Constructors are class methods. As such, they are represented at run-time as objects just like other methods. The most important method of the class *Constructor* is *newInstance*. This method allows creation of an object using the underlying receiver constructor object. The arguments are supplied as an array of objects. Any mismatch of the formal type parameters as produced by the method *getParameterTypes* and

the actual parameters supplied to the method *newInstance* will be detected by the run-time type system and exceptions raised.

```
public final class Constructor extends Object {
 public String getName();
 public int getModifiers();
 public Class[] getParameterTypes();
 public Class[] getExceptionTypes();
 public Object newInstance(Object initargs[])
      throws InstantiationException, IllegalArgumentException,
      IllegalAccessException, InvocationTargetException;
}
```

3.1.6 Updating Fields

Updating a field value is illustrated by the method *updateField*. This method takes as arguments the object whose field should be updated and the name of the field. Then the class of the object given as the first parameter of the method *updateField* is accessed using the method *getClass*. The field object is accessed using the given field name. The type of this field is discovered using the method *getType*. The method *getObjectOfType* is invoked to construct an object of the discovered type from the user provided input. Then the method *set* is invoked on the field object to update the value of the field.

```
void updateField(Object o, String fieldName) {
 Object v;
 try {
      Field f = o.getClass().getField(fieldName);
      do {
      v = getObjectOfType(f.getType());
      } while (v = null);
      if (v ≠ null)
      f.set(o, v);
      } catch (Exception e)
      { exception handling }
 }
```

3.1.7 Invoking Methods

Invoking methods using reflection requires a receiver object and a method object as illustrated by the method *invokeMethod* given below. The first step for correct invocation is discovering the parameter types. The method *getParameterTypes*

returns an array of class objects of the parameter types. The argument objects of the discovered types are then constructed from the user input invoking the method *getObjectOfType*. Finally, the method *invoke* of the class *Method* is called on the given method object with the receiver object (the first parameter of *invokeMethod*) and the constructed array of argument objects. The result of invocation of the method *invoke* is returned as the result of the method *invokeMethod*.

```
Object invokeMethod(Object o, Method m) {
 Class[] params = m.getParameterTypes();
 Object[] args = new Object[params.length];
 Object newObject = null;
 for(int i = 0; i < params.length; i++) {
 Class cls = params[i];
 args[i] = getObjectOfType(cls);
 try {
      newObject = m.invoke(o, args);
 } catch (Exception e) { exception handling }
 return newObject;
}
```

The above code, as well as other code samples that we present in this chapter, follow Java except for some complexities related to parametric types that obscure the core ideas and are thus omitted.

3.1.8 Creating Class Objects

Class objects are constructed by the class loader. Java has a class *ClassLoader* that can be extended. Its simplified code is presented below. An extensible class loader is a distinctive feature introduced by the Java technology. The method *loadClass* takes the full class name as a parameter, it constructs the class object from the class file, and returns it as the result. The method *defineClass* takes a sequence of bytes as its input. If this sequence satisfies the class format requirements, it constructs a class object out of those bytes. The method *resolveClass* links the loaded class.

```
public abstract class ClassLoader {
 protected ClassLoader();
 protected Class loadClass(String Name)
      throws ClassNotFoundException;
 protected Class defineClass(byte[] bytes)
      throws ClassFormatError;
 protected void resolveClass(Class c);
 // other methods
}
```

An example of extending the class *ClassLoader* to produce a specialized loader is given below. The method *loadClass* is overridden. It attempts to find a file with a given name that is meant to contain the bytes required to construct a class object. This is done using an auxiliary method *getClassBytes* that reads these bytes into a buffer. Then the method *defineClass* of the class *ClassLoader* is invoked to construct a class object on the heap.

```
class SpecialLoader extends ClassLoader {

 public Class loadClass(String name)
      throws ClassNotFoundException
 {
      try {
      byte[]buffer = getClassBytes(fileName);
      return defineClass(buffer);
      }
      catch (IOException e)
      {exception handling}
// other methods: getClassBytes etc.
}
```

3.1.9 Class Files

Java classes are compiled into Java class files. The class loader locates the class file, reads the bytes in the class file, and constructs a class object on the heap. The structure of Java class files is described below in the Java notation.

The information collected in the process of compilation of a Java class is stored in a table of the generated class file. This table is called the constant pool. The fields *thisClass* and *superclass* of a class file refer to the constant pool information about the class itself and about its superclass. A class file also contains an array of references to the constant pool items that contain information about the interfaces implemented by this class.

```
public class ClassFile {
 // file type and version data
 CPInfo        constantPool[];
 short         accessFlags;
 CPInfo        thisClass;
 CPInfo        superClass;
 CPInfo        interfaces[];
 FieldInfo     fields[];
 MethodInfo    methods[];
 AttributeInfo attributes[];
// other info
}
```

The information about fields and methods of a class is collected into the arrays *fields* and *methods* of the class file. The structure of the corresponding array elements for methods is given below in the Java notation. The structure for fields follows the same pattern.

```
public class MethodInfo {
 short          accessFlags;
 CPInfo         name;
 CPInfo         signature;
 AttributeInfo attributes[];
// other info
}
```

The name of a method is given as a reference to a constant pool item which contains the actual string. The signature of a method consists of its parameter types and the result type. This type descriptor is represented as a specially formatted string.

In the Java class file structure an array of attributes is associated with each class, each field, and each method. These attributes represent a flexibility of the Java class file structure that has been used for features that were not in the original language design, like parametric types.

3.2 Virtual Machine

3.2.1 The Structure of the Virtual Machine

Modern object-oriented languages such as Java and C# run on virtual machines: JVM (Java Virtual Machine) and CLR (Common Language Runtime). A virtual machine is an efficient interpreter that provides an abstraction over the underlying machine architecture. Object-oriented languages are compiled into the code executable by the underlying virtual machine. This makes compiled code portable as there are multiple implementations of virtual machines on different machine architectures.

In this section we present the basic structure and the functionalities of an object-oriented virtual machine. This is an abstract view that follows Java Virtual Machine. The main memory components of an object-oriented virtual machine are presented in Fig. 3.2. The three main components are the run-time stack, the heap, and the file system.

The stack component implements the run-time model for procedural languages. The stack consists of frames, one for each invocation of a method. A stack frame contains space for values of local variables and parameters of the corresponding method, in addition to the control information. We will view the run time stack as an array whose elements are of the most general type *Object*. There is a pointer to

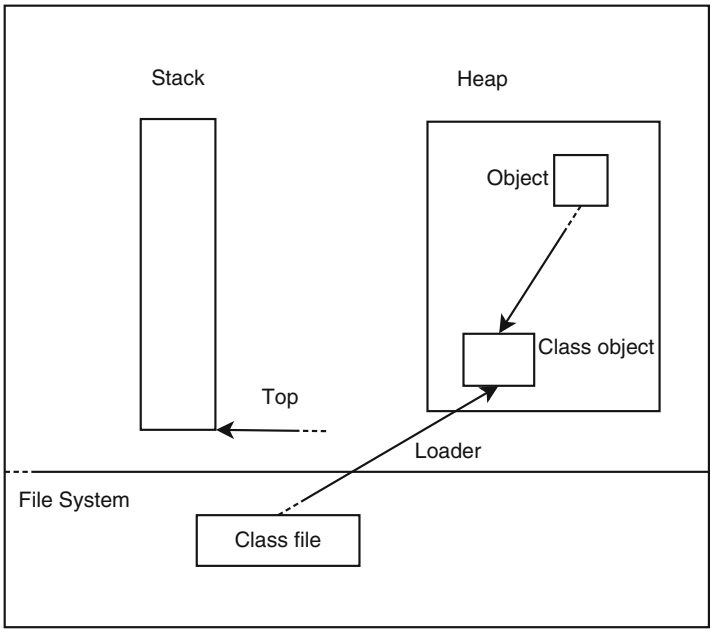


Fig. 3.2 Structure of the virtual machine

the element of the array *Stack* denoting the top of the stack. In this simplified view simple types could be viewed as objects of their corresponding wrapper classes that Java has.

$Stack : Object[]$

$Top : int$

Objects are allocated on the heap. Since this happens ad hoc, there is no particular ordering of elements on the heap. This is why we view the heap as a set of objects.

$Heap : Set < Object >$

Compiling a class produces a class file in the file system. A class file is a sequence of bytes of a particular format. The class loader reads a class file, and produces a class object on the heap.

$classFile : Sequence < byte >$

$loader : classFile \rightarrow Class$

The code is also allocated in a special area of the heap. Each instruction of the Java Virtual Machine is a sequence of bytes where the first byte is the operation code and the remaining bytes specify the arguments that are not available on the top of the stack.

$VMop : Sequence < byte >$
$code : Sequence < VMop >$

3.2.2 Creating Objects

The virtual platform operation that creates a new object will be viewed as a function that takes a class object as its argument and returns an object of that class that is allocated on the heap.

$newInstance : Class \rightarrow Object$

The precondition for execution of the operation newInstance(c) is that a class object is loaded on the heap:

$(\exists\ c : Class) \wedge (c \in Heap)$

Upon execution of the operation newInstance(c), a new object of the type specified by the class object is allocated on the heap, the top of the stack is increased by one, and a reference to the created object is placed on the top of the run-time stack.

$(\exists\ obj : Object)(obj \in Heap) \wedge$
$(newInstance(c)=obj) \wedge (obj.getClass()=c) \wedge$
$(Top=\mathbf{old}(Top)+1) \wedge$
$(Stack[Top]=obj)$

The effect of the virtual platform operation *newInstance* is illustrated in Fig. 3.3.

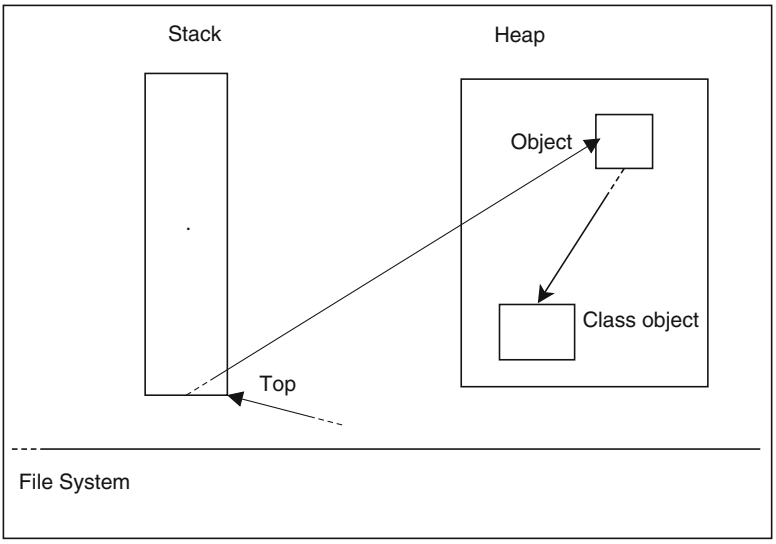


Fig. 3.3 Creating objects

There are two simplifications in our specifications. The first one is that we do not make explicit use of reference types. In Chap. 5 we specify a more general virtual machine that has explicit reference types. The second simplification is that we

omit frame constraints from the specifications in this section. In this case there are obvious frame constraints: except for the top of the stack, other elements of the stack are unaffected, and other objects on the heap are not affected either. The frame constraints are elaborated in the exercises.

3.2.3 Invoking Methods

The virtual platform operation *invokeMethod* invokes a method specified by its *Method* object. Arguments are supplied as an array of objects. The result of the method invocation is an object.

$invokeMethod : Object, Method, Object[] \rightarrow Object$

The precondition of the operation invokeMethod(r,m,a) is that the method object is available on the heap, that the receiver object r is on the top of the stack followed by a sequence of argument objects.

$(\exists\, m : Method)(m \in Heap) \wedge$
$(\exists\, r : Object)(Stack[Top]{=}r) \wedge (r \in Heap) \wedge$
$(\exists\, a : Object[])(a \in Heap) \wedge$
$(\forall i : int)(i \geq 0) \wedge (i < length(a))(Stack[Top - 1 - i]{=}a[i])$

The postcondition of the operation invokeMethod(r,m,a) is that a reference to the result of the method invocation is placed on the top the stack after the receiver and the arguments are removed from the top of the stack.

$(\exists\, obj : Object)(obj \in Heap) \wedge (invokeMethod(r, m, a){=}obj) \wedge$
$(Top{=}\mathbf{old}(Top) - length(a)) \wedge (Stack[Top]{=}obj)$

The effects of the operation *invokeMethod* are illustrated in Fig. 3.4.

There are frame constrains not given in the above specification that are left for the exercises.

3.2.4 Accessing Fields

The virtual platform operation *getField* performs access to a field of an object. We will view it as a function that takes a reference to a field object and an object whose field is to be accessed and returns the value of that field.

$getField : Field, Object \rightarrow Object$

The precondition of the operation getField(f,obj) is that the field object and the object whose field will be accessed are both allocated on the heap.

$(\exists\, f : Field)(f \in Heap) \wedge$
$(\exists\, obj : Object)(obj \in Heap)$

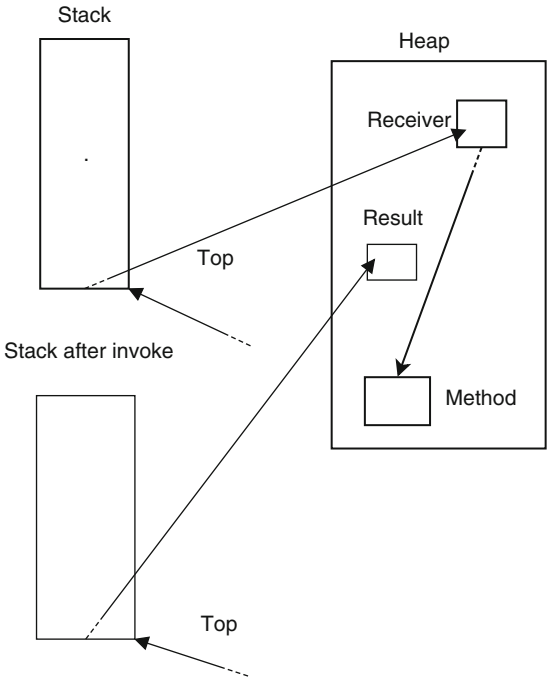


Fig. 3.4 Invoking methods

Upon execution of the operation getField(f,obj), the value of the field (in fact its reference) which is the result of the operation *getField* is pushed on the top of the stack.

$(\exists\ val : Object)(val \in Heap) \wedge$
$(getField(f, obj){=}val) \wedge (Stack[Top]{=}val) \wedge$
$(\text{Top} = \textbf{old}(\text{Top}) + 1)$

Updating the value of a field is performed by the virtual platform operation *putField*. This operation is viewed as a function that takes a field object, an object whose field will be updated, and the new value of the field. *putField* returns an updated field.

$putField : Field, Object, Object \rightarrow Field$

The precondition of the operation putField(f,obj,val) is that the field object and the object itself are available on the heap. A reference to the object and a reference to the new value are placed on the top of the stack.

$(\exists\ f : Field)(f \in Heap) \wedge$
$(\exists\ obj : Object)(obj \in Heap) \wedge$
$(\exists\ val : Object) \wedge$
$(Stack[Top]{=}val) \wedge$
$(Stack[Top - 1]{=}obj)$

The postcondition is that performing the operation *getField* with the same field of the same object will produce exactly the updated value of the object as produced by the operation *putField*.

$(Top{=}\mathbf{old}(Top) - 1) \wedge$
$getField(putField(f, obj, val), obj){=}val \wedge$
$Stack[Top]{=}f$

The effect of the operation putField is presented in Fig. 3.5.

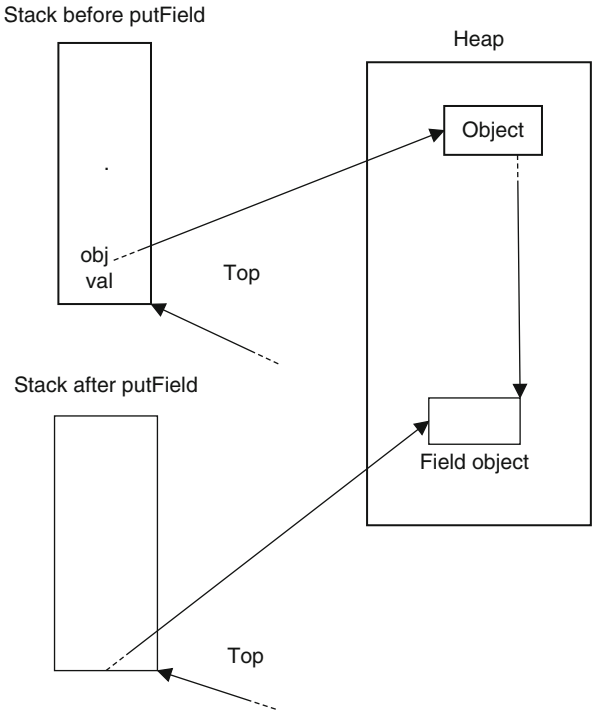


Fig. 3.5 Updating a field of an object

3.2.5 Operations

A binary operation will be viewed as a function:

$binaryOp : Object, Object \rightarrow Object$

Arguments and results of simple types may be viewed as instances of their corresponding wrapper classes. The precondition of the binary operation binaryOp(obj1,obj2) is that the two argument objects are available on the heap and that their references are placed on the top of the stack.

$(\exists\ obj1 : Object)(obj1 \in Heap) \wedge$
$(\exists\ obj2 : Object)(obj2 \in Heap) \wedge$
$(Stack[Top]{=}obj1)\ \wedge (Stack[Top - 1]{=}obj2)$

The postcondition of the operation binaryOp(obj1,obj2) is that the top of the stack contains a reference to the result of the operation after the top of the stack has been adjusted to remove the arguments of *binaryOp*.

$Top{=}\mathbf{old}(Top) - 1$
$Stack[Top]{=}binaryOp(obj1, obj2)$

The effect of invoking a binary operation on the run-time stack is presented in Fig. 3.6.

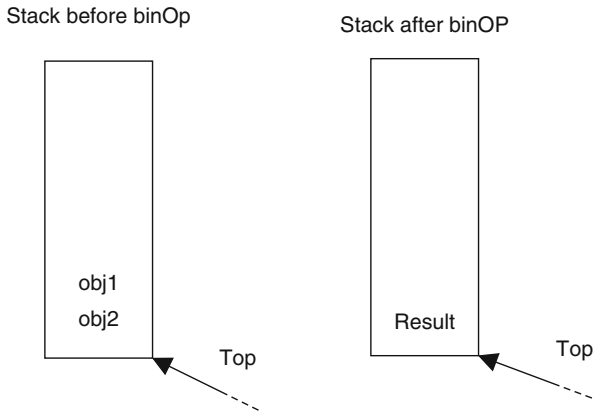


Fig. 3.6 Binary operations

3.2.6 Arrays

Object-oriented platforms contain special operations on arrays. Creating an array object is performed by the virtual platform operation *newArray*. This operation may be viewed as a function that takes a class object representing the type of array elements, and the size of the array to be created, and returns an array object.

$newArray : Class, Int \rightarrow Object[]$

The precondition for the operation newArray(c,n) is that the class object for elements of the array is available on the heap and that the top of the stack contains the size of the array to be created.

$(\exists\ c : Class)(c \in Heap) \wedge$
$(\exists\ n : Int)(Stack[Top]{=}n)$

Upon execution of the operation newArray(c,n) the array object that is the result of this operation is created on the heap. Its component type obtained by the function *getComponentType* is precisely the first argument of the operation *newArray*. The size of the created array object is equal to the second argument of this operation. A reference to the newly created array object is placed on the top of the stack.

$(\exists\ obj : Object[])(obj \in Heap) \wedge$
$(newArray(c, n){=}obj) \wedge (isArray(getClass(obj)) \wedge$
$(getComponentType(getClass(obj){=}c)) \wedge$
$(Stack[Top]{=}obj) \wedge (length(obj){=}n))$

The effect of the operation *newArray* is presented in Fig. 3.7.

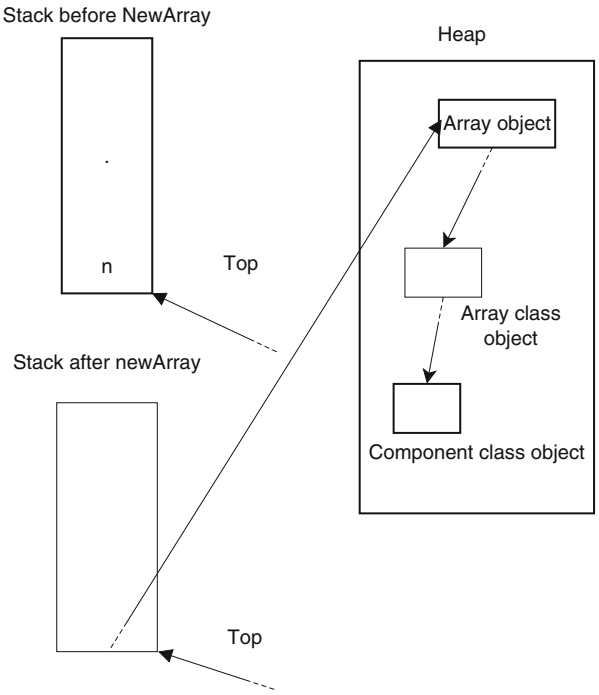


Fig. 3.7 Creating array objects

The virtual platform operation *arrayLoad* loads the element of an array at a given index on the top of the stack. It may be viewed as a function that takes an array object and the index value, and returns an element of the array at that index.

$arrayLoad : Object[], Int \rightarrow Object$

The precondition of the operation arrayLoad(a,n) is that the array object is available on the heap and that the index value and a reference to the array object are on the top of the stack.

$(\exists\ a : Object[])(a \in Heap) \wedge$
$(\exists\ n : Int)(Stack[Top]{=}a) \wedge (Stack[Top - 1]{=}n)$

The postcondition of the operation arrayLoad(a,n) is that a reference to the element at the given index of the given array is placed on the top of the stack. This is precisely the result of the function *arrayLoad*.

$(Top{=}\mathbf{old}(Top) - 1) \wedge$
$(\exists\ obj : Object)(arrayLoad(a, n){=}obj) \wedge$
$(Stack[Top]{=}obj)$

The effect of the operation *arrayLoad* is presented in Fig. 3.8.

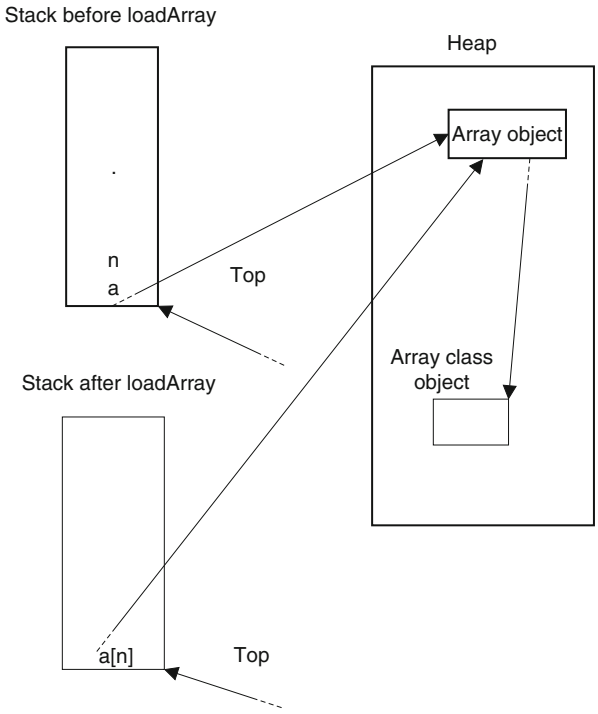


Fig. 3.8 Loading an array element

The virtual platform operation *arrayStore* stores a new element value at a given index of an array. This operation may be viewed as function that takes an array object, the index of the element to be updated, and the new value for the element at that index, and returns the updated array.

$arrayStore : Object[], Int, Object \rightarrow Object[]$

The precondition of the operation arrayStore(a,n,val) is that the array object is available on the heap. In addition, a reference to the array object, the index value and the new element value are on the top of the stack.

$(\exists\ a : Object[])(a \in Heap) \wedge$
$(\exists\ n : Int)\ \wedge (\exists\ val : Object) \wedge$
$(Stack[Top - 2]{=}a)\ \wedge (Stack[Top - 1]{=}n) \wedge (Stack[Top]{=}val)$

The postcondition of the operation arrayStore(a,n,val) is that the top of the stack has been adjusted to remove the objects needed by this operation. In addition, invoking the operation *arrayLoad* on the given array with a given index will produce the updated element given as the third argument of the operation *arrayStore*.

$(Top{=}\mathbf{old}(Top) - 3) \wedge$
$(arrayLoad(arrayStore(a, n, val), n){=}val) \wedge$
$(\forall i : Int)(i \neq n)(a[i]{=}\mathbf{old}(a[i]))$

Specification of the frame constraints will indicate that the other elements on the stack are not affected by this operation. The frame constraints are elaborated in the exercises.

3.3 Extending Virtual Platform

In the current object oriented technology, accessing classes that were not available at compile time via reflection produces type signatures for classes, methods, and fields. But the programmer is in the dark as to what exactly those methods are doing, or what the properties of objects of the discovered classes are. Currently, if the source code is not available, the semantic information cannot be introspected by reflection, hence correctly using classes discovered by reflection is a very problematic matter. Availability of constraints by reflection provides the basic semantics of classes and methods. This is a prerequisite for their correct use.

To overcome the problems outlined above, a system called XVP (Extended Virtual Platform) manages constraints in their declarative form which is associated with the type information available both statically and dynamically. The typing environment is extended at compile-time and at run-time with semantic information expressed by logic-based constraints.

The general architecture of the XVP extended virtual platform is represented in Fig. 3.9. The main components of this architecture are:

- Class files that allow representation of logic-based constraints.
- Class objects that contain type signatures along with constraints.
- A loader that assembles class objects from class files and properly manages type signatures and constraints.
- Reflective capabilities that allow introspection of types and constraints.

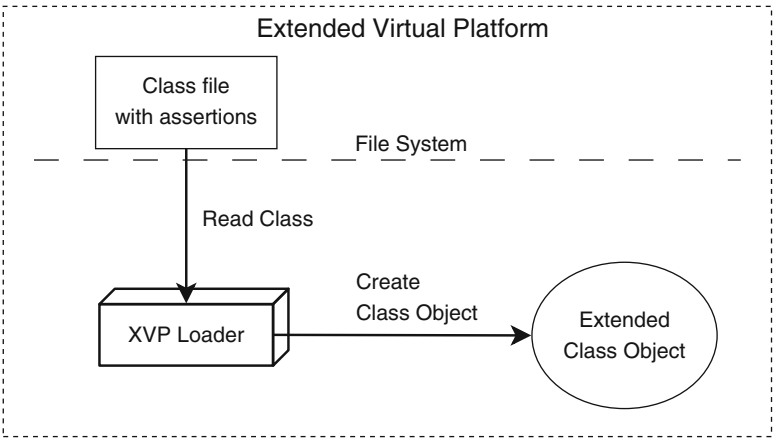


Fig. 3.9 Platform architecture

The Java Core Reflection (JCR) classes that have been extended are *Class*, *Constructor*, and *Method*. These extensions are based on new types such as *Invariant*, *PreCondition*, and *PostCondition*. With these new types it becomes possible to add method preconditions and postconditions to the class *Method*, and the class invariant to the class *Class*. These assertions require further types that make it possible to create objects that represent logical formulas for constraints. In order to achieve independence of a particular constraint language and its logic basis, the types representing logical formulas are specified as abstract classes. These classes must be extended for a particular assertion language.

The structure of constraints is determined by their underlying logic basis. This is why the class *Sentence*, given below, is abstract, and its methods only report universally and existentially quantified variables. A sentence is a logical formula with no free variables, i.e., all its variables are quantified. An example of a sentence is an invariant.

```
public abstract class Sentence {
 public Variable[] getVariables();
 public Variable[] getExistentialVariables();
 public Variable[] getUniversalVariables();
 public abstract boolean evaluate(Object[] variables);
}
```

Methods of the class *Variable* report names and types of variables. A logical formula in general contains free variables. An example of a formula is *this.count(o)* $\geq$ *0*. This is reflected in the definition of the abstract class *Formula*. If the values of free variables are bound to values invoking the method *bindVariables*, a formula may be evaluated. But a specific logic still must be chosen in order to perform the evaluation of a formula.

```
public abstract class Formula extends Sentence {
 public Variable[] getFreeVariables();
 public void bindFreeVariables(Object[] vars);
}
```

The basic building blocks for constructing expressions are terms. An example of a term is *this.count(o)*. The class *Term*, specified below, is abstract to allow for a variety of possible forms of terms. A term has a type and a collection of free variables. Given values of these variables a term may be evaluated. However, the method *evaluate* in the class *Term* is abstract since the specific evaluation rule depends upon the form of a term.

```
public abstract class Term {
 public Class getType();
 public Variable[] getVariables();
 public abstract Object evaluate(Object[] variables);
}
```

A message term consists of the receiver term, a method, and an array of arguments which are also terms. The specific evaluation rule amounts to substitution of arguments and invocation of the underlying method.

```
public abstract class MessageTerm extends Term {
 public Term getReceiverTerm();
 public Method getMethod();
 public Term[] getArguments();
 public Object evaluate(Object[] variables);
}
```

Formulas are constructed recursively starting with atoms and applying the rules of a particular logic.

Additions of the recompiled class *Class* allow access to the declared and the inherited invariant.

```
public final class Class { ...
 public Invariant getInvariant();
}
```

The extensions of the class *Method* allow access to (declared and inherited) preconditions and postconditions. The class *Constructor* is similarly extended.

```
public final class Method { ...
 public PostCondition getPostCondition();
 public PreCondition getPreCondition();
}
```

An application illustrating the advantages offered by the XVP extended virtual platform is from the database area where constraints are absolutely critical. In spite

of that, practically all object-oriented database technologies are lacking general, logic-based constraint capabilities. Extended reflection is critical because database users expect to see declarative constraints when introspecting classes in a database schema. Procedural representation of constraints as in Eiffel or JML is completely unacceptable for introspection.

In this experiment, Versant's FastObjects, a legacy ODMG object-oriented database management system was used. The XVP extended platform was used with FastObjects rather than the standard JVM. Just like ODMG, FastObjects does not know anything about constraints. Since FastObjects does not have such a capability, a transaction example below shows how a particular transaction can access assertions using extended reflection, evaluate them, and proceed depending upon the result, i.e., abort or commit.

First, the program initializes access to the database. The employee object "John Doe" is looked up in the database and returned. At this point components of the XVP are utilized. The invariant and the precondition are accessed from the employee class and evaluated. If the invariant and the precondition evaluate to true, then the program binds variables in postconditions, employs an evaluation strategy that takes into account the semantics of the *old* operator, and continues to execute the update on the employee. After the update has taken place, the postcondition and the invariant are evaluated. If each assertion evaluates to true, the transaction is completed and committed to the database.

```
public void update() {

 Transaction txn = new Transaction();
 txn.begin();
 Employee employee = null;
 try { employee = (Employee)database.lookup("John Doe");
      } catch (ODMGException e) { txn.abort(); /* other actions */ }

 Invariant inv = employee.getClass().getInvariant();

 Method m = null;
 try { m = employee.getClass().getMethod("updateSalary", Float.class);
      } catch (Exception e) { txn.abort(); /* other actions */ }

 PreCondition pre = m.getPreCondition();
 Object[] params = new Object[]{1000f};
 if (¬ (pre.evaluate(employee, params) ∧ inv.evaluate(employee)) )
      txn.abort();

 PostCondition post = m.getPostCondition();
 post.bindPreMethodVars(employee, params);
 employee.updateSalary(1000f);
 if (¬ (post.evaluate(employee, params) ∧ inv.evaluate(employee)))
```

```
      txn.abort();
 txn.commit();
}
```

3.4 Persistent Objects

3.4.1 Orthogonal Persistence

A persistent object is an object whose lifetime extends beyond execution of the program that created that object. Persistent objects are thus objects with possibly very long lifetimes. This concept is implemented by providing some form of a persistent store containing objects that are promoted to persistence.

Among a variety of models of persistence the model of *orthogonal persistence* deserves special attention. This model has the following properties:

- *Orthogonality*
  Persistence is independent of types, i.e., an object (or a value) of any type may be persistent.
- *Transitivity (reachability)*
  If an object is promoted to persistence, so are all of its components, direct or indirect.
- *Transparency*
  The details of the persistent supporting architecture are completely hidden from the users.

Widely used technologies typically support only some of the above properties. For example, in relational systems only objects of type relation are persistent. Even tuples can persist only as long as they appear in relations, not by themselves. In Java, only objects of classes that implement a special interface *Serializable* can persist. In relational systems transparency is supported. In Java, a user must deal with opening and closing files, writing objects to files and reading objects from files. Basic relational systems do not have complex objects per se. Java supports transitivity, with some issues to be explained in this section.

A very small number of systems implemented orthogonal persistence, such as PJama and Gemstone. PJama is in fact a persistent Java system. This system provides one interface *PJStore* and its implementing class to support orthogonal persistence without introducing any changes to the Java language. The interface *PJStore* is given below:

```
public interface PJstore {
 public static PJstore getStore();
 public void newProot(String name, Object obj);
 public void setProot(String name, Object obj);
 public Object getProot(String name);
}
```

The method *getStore* gets a reference to the underlying persistent store. The method *newProot* promotes the object supplied as the second argument to persistence and binds the name supplied as the first argument to the newly established persistent object. Subsequently, this object may be accessed using its name. This is why this object is called a root of persistence. By transitivity, all components of a persistent root (direct or indirect) are also promoted to persistence. Since the second argument of the method is of type *Object*, that implies that objects of any type may be promoted to persistence (orthogonality).

The PJama model of persistence will be illustrated using the following classes.

```
public class Aircraft
{ private String model;
 private Pilot pilot;
 public Aircraft(String aModel)
 { model = aModel; pilot = null; }
 public assignPilot(Pilot p)
 { pilot=p;}
 // other methods
}

public class Pilot
{ private String name;
 private int points;
 public Pilot(String pName, int pPoints)
 { name = pName; points = pPoints; }
 // other methods
}
```

Promoting a complex object to persistence is illustrated below in the class *StoreAircraft*. The aircraft pilot object, as a component of the aircraft object, is also promoted to persistence when the method *newProot* is invoked.

```
public class StoreAircraft {
 Aircraft airObj = new Aircraft("Boeing777");
 Pilot pilot = new Pilot("Mark Sellinger");
 airObj.assignPilot(pilot);
 try {
      PJstore pjs = PJstore.getStore();
      pjs.newProot("MarksPlane", airObj);
 }
 catch (PJexception e) { exception handling }
}
```

The method *getProot* looks up a persistent object by its name and returns it as the result of invocation of this method. The result type is necessarily of type

Object because *getProot* must be applicable to objects of any type. This means that performing specific methods on the retrieved object requires a type cast. This is illustrated by the class *ChangePilot* in which the pilot of an aircraft object is changed. When the program terminates, this change is promoted to the persistent store automatically (transitivity and transparency). This means that the program acts like a limited form of a transaction.

```
public class ChangePilot {
 public static void main(String[] args) {
 try {
      PJstore pjs = PJstore.getStore();
      Pilot p= new Pilot("Mark Royer");
      Aircraft airObj = (Aircraft)pjs.getProot("MarksPlane");
      airObj.assignPilot(p);
      }
      catch (PJexception) { exception handling }
 }
}
```

The class *DisplayAircraft* shows how access to a persistent root is performed. The persistent root is accessed by the name that it is bound to, and then the method *display* is invoked on that object. This requires a type cast of the returned object to the type *Aircraft*, as explained above.

```
public class DisplayAircraft {
 public static main(String[] args) {
 try {
      PJstore pjs = PJstore.getStore();
      Aircraft airObj = (Aircraft) pjs.getProot("MarksPlane");
      airObj.display();
      }
      catch (PJexception e) { exception handling }
 }
}
```

The list of various PJama methods that apply to persistent objects is given below. They allow checking whether a persistent object with a given name exists in the persistent store, getting an enumeration of all names of persistent objects, getting an enumeration of all persistent objects, and discarding a persistent object with a given name.

```
public boolean existProot(String name)
public Enumeration getAllProotNames()
public Enumeration getAllProots()
```

```
public Object getProot(String name)
public void newProot(String name, Object obj)
public void discardProot(String name)
```

In the above method signatures *Enumeration* is the Java legacy interface given below because Java did not have parametric types when PJama was designed and implemented.

```
public interface Enumeration {
 boolean hasMoreElements();
 Object nextElement();
}
```

3.4.2 Persistence Architecture

The transparency property hides non-trivial complexity of the underlying persistence architecture. This is illustrated in Fig. 3.10.

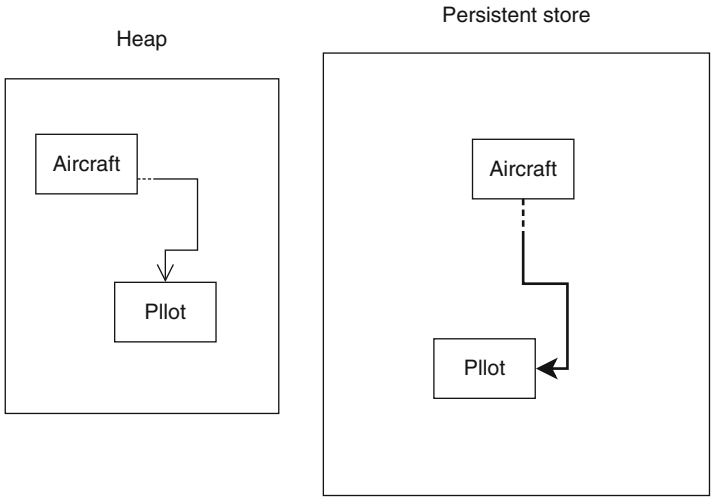


Fig. 3.10 Persistent complex objects

The structure of the aircraft object is represented on the heap using pointers that are in fact heap addresses. So an aircraft object will contain a pointer to a pilot object on the heap. When a complex object is promoted to persistence, its structure must be maintained. However, the pointers in the persistent store (typically disc

addresses) are different from the heap pointers. This means that the whole complex object structure of an object must be preserved in the persistent store, and pointers to component objects must be implemented as disc addresses. This procedure is called swizzelling out. The reverse operation is swizzelling in. It restores the complete persistent object structure on the heap. In a transparent model of persistence these procedures are automatic and completely hidden from the users.

One implication of transitivity is that if an object is promoted to persistence, its class object should also be promoted to persistence. The reason is that an object contains a reference to its class object. In addition, in order to perform actions on complex objects, one needs to know what types of objects are in the persistent store. PJama has the following methods that operate on persistent class objects:

```
public Enumeration getAllPclassNames()
public Enumeration getAllPclasses()
public boolean existsPclass(String name)
public Class getPclass(String name)
```

The class objects related methods allows getting an enumeration of the names of all persistent class objects, getting an enumeration of all class objects, checking whether a class object with a given name exists in the persistent store, and getting a class object with a given name.

Transitivity also implies that the superclass of an object should also persist as a class object contains a reference to the superclass object. This implies that the whole relevant part of the inheritance hierarchy all the way to the root class object will be promoted to persistence.

3.4.3 Object Serialization

The Java model of persistence offers transitivity, but not transparency nor orthogonality. Only objects of types that implement a special interface *Serializable* could be made persistent. This leads to a paradox: it is not possible to define a persistent collection whose elements are of the type *Object*, because *Object* does not implement *Serializable*. That would have to be the case in order for the Java model of persistence to be orthogonal.

The Java model does not satisfy the transparency requirement because it is based on the file system, so users have to open and close files, read and write objects from and to files, etc. However, transitivity is supported to the extent that it is possible to write a complete complex object to a file with a single statement, and read a complex object from a file in a single statement. This is accomplished through interfaces *ObjectInput* and *ObjectOutput* and their implementing classes. These interfaces also have methods for reading and writing values of simple types that are specified in the Java interfaces *DataInput* and *DataOutput*.

The method *writeObject* takes an object of any type and writes it to the output file stream. The underlying algorithm represents the structure of a complex object as a sequence of bytes, hence the term serialization. This is illustrated in Fig. 3.11.

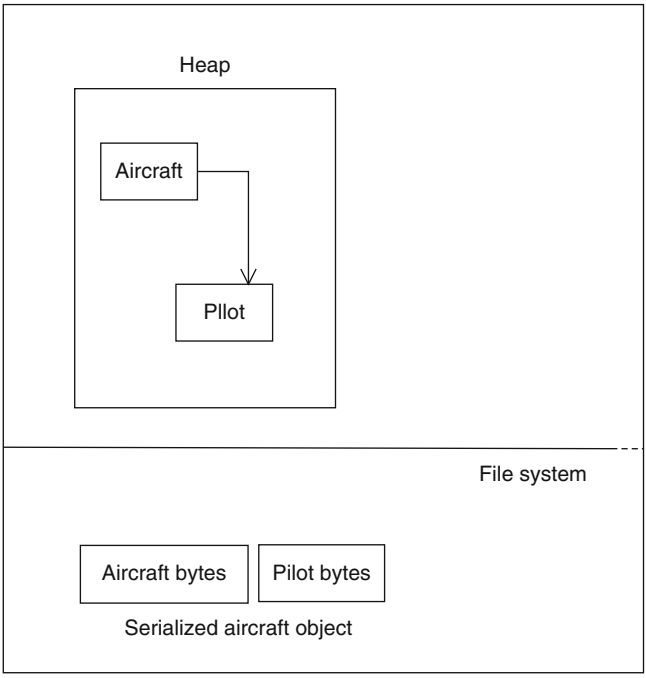


Fig. 3.11 Serialized complex object

```
public interface ObjectOutput extends DataOutput
{
 void writeObject(Object obj);
      throws IOexception;
 // other methods
}
```

The method *readObject* reads an object from a file input stream. The complex object structure is restored on the heap based on its serialized representation.

```
public interface ObjectInput extends DataInput
{
 Object readObject()
      throws ClassNotFoundException, IOexception;
 // other methods
 }
```

The argument of the method *writeObject* is necessarily of type *Object* and the result of the method *readObject* is necessarily of type *Object* as well. This reflects the requirement that objects of any type may be read or written. This means that a type cast is necessary when reading an object in order to perform specific actions on the object read. This dynamic check is unavoidable. In general, an object is written by one program and read by a different program. The type cast verifies that the type assumption made by the second program is in fact correct. This is illustrated in the following code:

```
public class Aircraft extends Serializable {
// . . .
}
```

```
FileOutputStream fileOut = new FileOutputStream("AircraftFile");
ObjectOutput out = new ObjectOutputStream(fileOut);
aircraftObj = new Aircraft("Boeing777"); out.writeObject(aircraftObj);
out.flush();
out.close();
```

```
FileInputStream fileIn = new FileInputStream("AircraftFile");
ObjectInput in = new ObjectInputStream(fileIn);
Aircraft aircraftObj= (Aircraft) in.readObject();
in.close();
```

In the Java model of persistence, class objects are not written to the file streams. Strictly speaking, this means that the model does not fully support transitivity. Java uses a shortcut to store the class type information using a hashed value that is subsequently used in a type cast to verify its type correctness.

3.5 Bibliographic Remarks

The presentation of reflection follows the early specification of Java Core Reflection (JCR) [9]. The reason is that this provides a conceptually clean view of object-oriented reflection unlike the recent versions of JCR that are unnecessarily complicated because of parametric types in Java. This early JCR view is also much more elegant than the C# reflective capabilities [1]. Two examples of using JCR are due to Daniel Lawrence. Presentation of the loader follows [5].

The presentation of an object-oriented virtual machine is an abstraction of otherwise very low level specification [11]. A distinctive feature of this abstraction is assertions that specify formally the semantics of various operations of the virtual machine.

An extension of the Java Virtual Machine (JVM) that provides a correct implementation of parametric types in Java is presented in [3]. An extension of the JVM that allows representation and management of assertions is presented in [12]. A technique of using reflection to implement a database language Java OQL in a type safe manner is presented in [4]. A very general form of reflection that handles ad hoc polymorphism in a type safe manner is linguistic reflection [10].

A basic exposition of the features of PJama, an orthogonal model for persistence for Java, is [7]. The implementation of this model requires an extension of the JVM. The architectures of persistent object systems are elaborated in [6]. See also [2].

The Java model of persistence based on serializability is explained in [5].

3.6 Exercises

1. Write a Java class whose main method takes the full name of a class as its input. It then attempts to load the class object with that name using the method *forName* of the class *Class*. The type signatures of fields and methods are then printed as the output.
2. Write a Java class whose main method walks the type hierarchy of a class whose name is given in the input. The program first attempts to load the class using the method *forName*. It then recursively displays all the type information for the interfaces that the given class implements, the class that the given class extends, all the way to the root class *Object*.
3. Specify the frame constraints for the virtual machine operation *newInstance*. Keep in mind that a frame constraint in general refers to both the runtime stack and the heap.
4. The virtual machine operation *invokeMethod* does not just produce the result but it also affects the state of the receiver object and other objects on the heap. Extend the specification of the operation *invokeMethod* to capture these side effects.
5. Specify the frame constraints for the operation *invokeMethod*.
6. Specify the frame constraints for the operations *getField* and *putField*.
7. Specify the frame constraints for the operation *binaryOp*. Keep in mind that this operation affects only the stack and has no impact on the heap.
8. Specify the frame constraint for the operations:
   - *newArray*
   - *arrayLoad*
   - *arrayStore*
9. Specify other subtypes of the type *Term* as defined in Java or C#.
10. Given types *Sentence* and *Formula*, specify types *Invariant*, *Precondition* and *PostCondition*.

11. Write a Java program that does the following:
    - Writes a complex object Collection<Aircraft> to a file.
    - Reads this object and performs a type cast to Collection<Aircraft> and a type cast to Collection<Pilot>.
    - Analyze the outcome of these type casts.
    - Iterate over the retrieved collection when viewed in accordance with the above two type casts.
    - Analyze the outcome.
12. Java has the following methods for customized serialization:

```
void writeObject(objectOutputStream out)
        throws IOException
void readObject(objectInputStream in)
        throws IOException, ClassNotFoundException
```

    - Specify the class *Collection* with your choice of the representation for the collection of elements.
    - Specify private methods *writeObject* and *readObject* of the class *Collection* that perform customized serialization and deserialization.

References

1. J. Alabahari, B. Albahari, *C# 5.0 in a Nutshell* (O'Reilly, Beijing, 2012)
2. S. Alagić, T. Ngyen, Parametric polymorphism and orthogonal persistence, in *Proceedings of the ECOOP 2000 Symposium on Objects and Databases*. Lecture Notes in Computer Science, vol. 1813 (Springer, Heidelberg, 2001), pp. 32–46
3. S. Alagić, M. Royer, Genericity in Java: persistent and database systems implications. VLDB J. **17**(4), 847–878 (2007)
4. S. Alagić, J. Solorzano, Java and OQL: a reflective solution for the impedance mismatch. L'objet **6**, 3 (2000)
5. K. Arnold, J. Gosling, D. Holmes, *The Java Programming Language*, 4th edn. (Addison-Wesley, Boston, 2005)
6. M. Atkinson, R. Morrison, Orthogonally persistent object systems. VLDB J. **4**, 319–401 (1995)
7. M. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, S. Spence, An orthogonally persistent Java^{TM}. ACM SIGMOD Rec. **25**, 68–75 (1996)
8. J. Gosling, B. Joy, G. Steel, G. Bracha, *The Java Language Specification*, 3rd edn. (Addison Wesley, Boston, 2005)
9. Java Core Reflection, JDK 1.1. Sun Microsystems (1997)
10. G. Kirby, R. Morrison, D. Stemple, Linguistic reflection in Java. Softw. Pract. Exp. **28**(10), 1045–1077 (1998)
11. T. Lindholm, F. Yellin, *The Java^{TM} Virtual Machine Specification* (Addison-Wesley, Boston, 1996)
12. M. Royer, S. Alagić, D. Dillon, Reflective constraint management for languages on virtual platforms. J. Object Tech. **6**(10), 59–79 (2007). http://www.jot.fm/issues/issue_2007_11/article1/index.html

Chapter 4
Type Systems

In Chap. 1 we presented the basic rules of object-oriented type systems in a mostly informal manner. However, type systems have an elaborate formal theory. In this section we show how some of the elements of type theory apply to an object-oriented programming language.

Section 4.1 is devoted to the formal specification of the basic rules of object-oriented type systems. In Sect. 4.1.1 we explain the basic typing rules for statements such as assignments, conditionals, and loops, as well as for expressions. In Sect. 4.1.2 we focus on the rules of an object-oriented type system that govern classes, inheritance, subtyping, and messages. These static typing rules are in Sect. 4.1.3 followed by dynamic type checking rules that mainstream object-oriented languages have. Formal rules for parametric types are given in Sect. 4.1.4.

Section 4.2 presents static and dynamic typing rules that apply to reflection. This section includes static and dynamic rules for field access using reflection, field update, and method introspection and invocation.

In Sect. 4.3 we show what kind of problems violation of type safe rules can cause. We take a well-known paradigm of type erasure that was the basis of the technique for extending Java with parametric types. This paradigm is provably incorrect as demonstrated by a variety of cases of type violation presented in this section. In Sect. 4.3.1 we consider legacy collection classes and their parametric versions and show how type erasure violates type safe rules of inheritance and subtyping. This is followed in Sect. 4.3.2 by a demonstration of the problems that type erasure causes in static type checking of assignments and messages. Even more serious typing problems are caused in dynamic type checking as shown in Sect. 4.3.3. In Sect. 4.3.4 we show what kind of typing problems type erasure causes in the model of persistence based on serializability. Finally, in Sect. 4.3.5 we show severe problems caused by type erasure when using reflection.

S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0_4

4.1 Formal Type Systems

4.1.1 Typing Rules

A typing environment, just like a symbol table built by a compiler, contains information about types of identifiers declared in a program. Based on that information, a typing environment also makes it possible to deduce the types of expressions that appear in a program. In doing so, the type checker (a compiler component) follows a collection of formal rules that specify the type system of the underlying programming language.

The typing environment will be denoted as $\mathcal{T}$. The fact that an identifier x in the typing environment $\mathcal{T}$ is associated with a type T will be denoted as $\mathcal{T} \vdash x : T$. For simplicity, we will also assume that subtype ordering deduced when processing subclasses is also recorded in the typing environment $\mathcal{T}$ rather than in a separate type constraint system as in some other approaches.

The standard typing rule for the assignment statement is given below. This rule says that if the type checker can conclude from the typing environment $\mathcal{T}$ that x is a variable of type A, and e is an expression of type A, then x = e is a type correct assignment statement. The typing rules have the form of Horn clauses.

Rule 1 (Assignment Statement).

$$\frac{\mathcal{T} \vdash x : A\ var, \mathcal{T} \vdash e : A}{\mathcal{T} \vdash x = e : Statement}$$

For example:

$$\frac{\mathcal{T} \vdash x : int\ var, \mathcal{T} \vdash x + 1 : int}{\mathcal{T} \vdash x = x + 1 : Statement}$$

In $\mathcal{T} \vdash x : A\ var$ we indicate that the typing environment contains an indication that x is a variable as in any compiler symbol table.

In order to deduce that **if** (e) S1 **else** S2 is a correctly typed conditional statement, the type checker must verify that e is a boolean expression and that S1 and S2 are correctly typed statements. This is specified in the rule for type checking of a conditional statement:

Rule 2 (Conditional Statement).

$$\frac{\mathcal{T} \vdash e : boolean, \mathcal{T} \vdash S1 : Statement, \mathcal{T} \vdash S2 : Statement}{\mathcal{T} \vdash \textbf{if}\ (e)\ S1\ \textbf{else}\ S2 : Statement}$$

For example:

$$\mathcal{T} \vdash x > 0 : \mathit{boolean}, \mathcal{T} \vdash y = x : \ \mathit{Statement}, \mathcal{T} \vdash y = -x : \ \mathit{Statement}$$

$$\mathcal{T} \vdash \textbf{if } (x > 0)\ y = x \textbf{ else } y = -x : \ \mathit{Statement}$$

The loop **while** (e) S is type correct if e is a boolean expression and S is a correctly typed statement, as in the rule below:

Rule 3 (While Loop).

$$\mathcal{T} \vdash e : \ \mathit{boolean}, \mathcal{T} \vdash S : \ \mathit{Statement}$$

$$\mathcal{T} \vdash \textbf{while } (e)\ S : \ \mathit{Statement}$$

For example:

$$\mathcal{T} \vdash x < \mathit{highValue} : \ \mathit{boolean}, \mathcal{T} \vdash x = x + 1 : \ \mathit{Statement}$$

$$\mathcal{T} \vdash \textbf{while } (x < \mathit{highValue})\ x = x + 1 : \ \mathit{Statement}$$

Likewise, the loop **do** S **while** (e) is correctly typed if from the typing environment the type checker can deduce that e is a boolean expression and S is a correctly typed statement.

Rule 4 (Do Loop).

$$\mathcal{T} \vdash e : \ \mathit{boolean}, \mathcal{T} \vdash S : \ \mathit{Statement}$$

$$\mathcal{T} \vdash \textbf{do } S \textbf{ while } (e) : \ \mathit{Statement}$$

Types of variables are recorded in the symbol table when the compiler is processing declarations. The types of constants can be inferred from their denotation. The types of the basic operators are predefined in the language definition and the typing environment is initialized accordingly. The types of expressions are not declared and must be inferred by the type checker from the types of constants, variables and operations that appear in expressions. Two sample rules are given below. In order to infer that e1 **op** e2 is an expression of type integer, the type checker must infer that e1 and e2 are expressions of type integer, and that **op** is a binary operator whose arguments and the result are of type integer.

Rule 5 (Expressions).

$$\mathcal{T} \vdash e1 : \ \mathit{int}, \mathcal{T} \vdash e2 : \ \mathit{int},\\ \mathcal{T} \vdash \textbf{op } \in \{+, -, *, /, \%\}$$

$$\mathcal{T} \vdash \ e1 \textbf{ op } e2 : \ \mathit{int}$$

For example:

$$\mathcal{T} \vdash x : \ \mathit{int}, \mathcal{T} \vdash y : \ \mathit{int}$$

$$\mathcal{T} \vdash \ x + y : \ \mathit{int}$$

Following the same logic, the typing rule for Boolean expressions will have the following form:

Rule 6 (Expressions).

$$\frac{\mathcal{T} \vdash e1 : \ boolean, \mathcal{T} \vdash e2 : \ boolean, \quad \mathcal{T} \vdash \mathbf{op} \ \in \{\&, |, \&\&, ||\}}{\mathcal{T} \vdash \ e1 \ \mathbf{op} \ e2 : \ boolean}$$

The above two rules contain operators available in Java.

4.1.2 Object-Oriented Type System

Consider the following class:

```
class Collection {
 private int size;
 private Object[] elements;
 public Collection() {...}
 public boolean belongs(Object x){...}
 public int cardinality(){...}
 public void add(Object x){...}
 public void remove(Object x) {...}
}
```

The type signature of a class C denoted Sig(C) consists of the type signatures for fields Sig(fields(C)), methods Sig(methods(C)) and constructors Sig(constructors(C)). For the above class we have:

$$\begin{aligned} &fields(C) =< int\ size;\ Object[]\ elements > \\ &constructors(C) =< Collection() > \\ &methods(C) =< boolean\ belongs(Object);\ int\ cardinality(); \\ &\qquad\qquad void\ add(Object);\ void\ remove(Object) > \end{aligned}$$

In general, type information for a class A with k^A fields and m^A methods will be represented using the following type signature expressions:

- Type signatures:
  $Sig(A) = \{fields(A); constructors(A); methods(A)\}$ where
- Field signatures:
  $fields(A) =< F_i\ f_i >_{i=1}^{k^A}$ (a sequence of fields) with F_i the type of field f_i, and $f_i \neq f_j$ for $i \neq j$, where $j = 1, 2, \ldots, k^A$.
- Constructor signatures:
  $constructors(A) = \{A(C_{c_{i_1}}, C_{c_{i_2}}, \ldots, C_{c_{i_n}})\}_{i=1}^{c^A}$

(a collection of constructors) where $C_{m_{i_j}}$ are types for $j = 1, 2, \ldots, n$.

– Method signatures:
$methods(A) = \{C_{m_i}\ m_i(C_{m_{i_1}}, C_{m_{i_2}}, \ldots, C_{m_{i_n}})\}_{i=1}^{m^A}$
(a collection of methods) where m_i is the method name, and C_{m_i}, $C_{m_{i_j}}$ are types where $j = 1, 2, \ldots, n$.

We will denote extending a typing environment $\mathcal{T}$ with a binding for a class C as $\mathcal{T} \cup \{\textbf{class}\ C\ \textbf{extends}\ B\}$. Producing the resulting environment $\mathcal{T}'$ will be denoted by the symbol $\Rightarrow$. Using this notation we obtain the following rule for the relationship between inheritance and subtyping in Java:

Rule 7 (Inheritance and Subtyping).

$$\frac{\mathcal{T} \cup \{\textbf{class}\ D\ \textbf{extends}\ B\} \Rightarrow \mathcal{T}'}{\mathcal{T}' \vdash D\ <:\ B}$$

The above rule says that if a typing environment $\mathcal{T}'$ contains the definition of a class D which extends a class B, then from $\mathcal{T}'$ we can deduce that D is a subtype of B. The standard symbol for subtyping $<:$ is used in the rule above. This rule says that the inheritance rules satisfy the subtyping rules as discussed in Chap. 1. Subtyping is a partial order (reflexive, antisymmetric and transitive). Here is the rule for transitivity:

Rule 8 (Subtyping).

$$\frac{\mathcal{T} \vdash\ T1\ <:\ T2,\ \mathcal{T} \vdash\ T2\ <:\ T3}{\mathcal{T} \vdash T1\ <:\ T3}$$

The object-oriented rule for the assignment statement can now be specified in a more flexible manner:

Rule 9 (Assignment Statement).

$$\frac{\mathcal{T} \vdash x :\ A\ var,\ \mathcal{T} \vdash e :\ B,\quad \mathcal{T} \vdash\ B\ <:\ A}{\mathcal{T} \vdash x = e : Statement}$$

For example:

$$\frac{\mathcal{T} \vdash x :\ Person\ var,\ \mathcal{T} \vdash y :\ Employee,\quad \mathcal{T} \vdash\ Employee\ <:\ Person}{\mathcal{T} \vdash x = y : Statement}$$

The above rule says that if the type of a variable x in the environment $\mathcal{T}$ is A, the type of the expression e is B, and B is a subtype of A, then the assignment statement $x = e$ is type correct. This rule can be simplified by requiring that the types of x and e are in fact the same as long as the type system contains the following subsumption rule:

Rule 10 (Subsumption).

$$\frac{\mathcal{T} \vdash\ e : B, \quad \mathcal{T} \vdash\ B <: A,}{\mathcal{T} \vdash e : A}$$

The following rule for inheritance of fields shows that all the fields from the superclass are inherited in the subclass in such a way that their types are invariant.

Rule 11 (Inheritance).

$$\frac{\begin{array}{c}\mathcal{T} \vdash D\ <:\ B, \\ \mathcal{T} \vdash \mathit{fields}(B) =< F_i^B\ f_i^B >_{i=1}^{k^B}, \\ \mathcal{T} \vdash \mathit{fields}(D) =< F_i^D\ f_i^D >_{i=1}^{k^D}, \\ k^B \leq k^D, \\ j \in \{1, 2, \ldots, k^B\}\end{array}}{\mathcal{T} \vdash < F_j^B\ f_j^B > =< F_j^D\ f_j^D >}$$

The premises in the above rule are that from a given typing environment $\mathcal{T}$ we can deduce that D is a subtype of B, that B has a sequence of fields f_i^B where $i = 1, 2, \ldots, k^B$ with respective types F_i^B , and that D has a sequence of fields f_i^D with respective types F_i^D for $i = 1, 2, \ldots, k^D$. With the above assumptions we can deduce that in the typing environment $\mathcal{T}$ the fields f_j^D for $j = 1, 2, \ldots, k^B$ have the same names and types as the corresponding fields f_j^B of B. In the above rule $< F_j^B\ f_j^B > =< F_j^D\ f_j^D >$ means $F_j^B = F_j^D$ and $f_j^B = f_j^D$.

For example:

$$\begin{array}{c}\mathcal{T} \vdash \mathit{OrderedCollection}\ <:\ \mathit{Collection}, \\ \mathcal{T} \vdash \mathit{fields}(\mathit{Collection})) =< \mathit{int\ size};\ \mathit{Object}[]\ \mathit{elements} >, \\ \mathcal{T} \vdash \mathit{fields}(\mathit{OrderedCollection})) =< \mathit{int\ size}; \mathit{Object}[]\ \mathit{elements};\ \mathit{other\ fields} >\end{array}$$

Inheritance of methods is governed by the following rule, which shows that even if an inherited method is overridden, the number and the types of arguments remain invariant:

Rule 12 (Overriding).

$$\frac{\mathcal{T} \vdash D <: B, \quad \mathcal{T} \vdash C_m\ m(C_{m_1}, C_{m_2}, \ldots, C_{m_n}) \in \mathit{methods}(B)}{\mathcal{T} \vdash C_m\ m(C_{m_1}, C_{m_2}, \ldots, C_{m_n}) \in \mathit{methods}(D)}$$

For example:

$$\mathcal{T} \vdash Collection <: Object,$$
$$\mathcal{T} \vdash boolean\ equals(Object) \in methods(Object)$$

$$\mathcal{T} \vdash boolean\ equals(Object) \in methods(Collection)$$

In recent editions of Java the above rule is relaxed allowing the result type of the method m in the class D to be a subtype of C_m. This is a type safe rule because it follows the rule for function subtyping.

For example:

$$\mathcal{T} \vdash Collection <: Object,$$
$$\mathcal{T} \vdash Object\ clone() \in methods(Object)$$

The following is fine:

$$\mathcal{T} \vdash Collection\ clone() \in methods(Collection)$$

The Java overloading rules prior to Java 5.0 imposed the following restriction on method signatures.

Rule 13 (Overloading).

$$\mathcal{T} \vdash C_m\ m(C_{m_1}, C_{m_2}, \ldots, C_{m_n}) \in methods(A),$$
$$\mathcal{T} \vdash C'_m\ m(C'_{m_1}, C'_{m_2}, \ldots, C'm_n) \in methods(A)$$

$$\mathcal{T} \vdash (\exists i)((i \in \{1, 2, \ldots, n\}), (C_{m_i} \neq C'_{m_i}))$$

In other words, if a class has two methods with the same name, their argument type signatures must be different, i.e., if they have the same number of arguments, at least one pair of the corresponding arguments must have different types. The same rule applies to overloaded constructors.

For example, the following type checks:

$\mathcal{T} \vdash boolean\ equals(Object) \in methods(Collection)$
$\mathcal{T} \vdash boolean\ equals(Collection) \in methods(Collection)$

Consider the rule that specifies when a message $x.m(a_1, a_2, \ldots, a_n)$ is type correct in an environment $\mathcal{T}$.

Rule 14 (Type Checking Messages).

$$\mathcal{T} \vdash x : A,$$
$$\mathcal{T} \vdash C_m\ m(C_{m_1}, C_{m_2}, \ldots, C_{m_n}) \in methods(A),$$
$$\mathcal{T} \vdash a_i : A_i, i \in \{1, 2, \ldots, n\},$$
$$\mathcal{T} \vdash (\forall i)(i \in \{1, 2, \ldots, n\} \Rightarrow A_i <: C_{m_i})$$

$$\mathcal{T} \vdash x.m(a_1, a_2, \ldots, a_n) : C_m$$

The requirements are that the type of the receiver x must be equipped with a method m with n arguments. If the type of a formal parameter of m is C_{m_i}, and the

type of the corresponding actual type parameter is A_i, then A_i must be a subtype of C_{m_i} for $i = 1, 2, \ldots, n$. This rule is based on the subtyping rule for functions.

For example:

$$\frac{\begin{array}{c}\mathcal{T} \vdash \mathit{employees} : \ \mathit{Collection}\ \mathit{var}, \\ \mathcal{T} \vdash \mathit{boolean}\ \mathit{add}(\mathit{Object}) \in \mathit{methods}(\mathit{Collection}), \\ \mathcal{T} \vdash x : \mathit{Employee}, \\ \mathcal{T} \vdash \mathit{Employee} <: \mathit{Object}\end{array}}{\mathcal{T} \vdash \mathit{employees.add}(x) : \ \mathit{boolean}}$$

4.1.3 Dynamic Type Checking

Dynamic type checking in Java is required by type casts and instance of tests. The following static rules govern type casts:

Rule 15 (Static Rule for Down Casts).

$$\frac{\mathcal{T} \vdash e : \ A, \mathcal{T} \vdash D \ <: \ A}{\mathcal{T} \vdash (D)e : \ D}$$

This rule says that if the type of an expression e in the typing environment $\mathcal{T}$ is A and we can deduce from $\mathcal{T}$ that D is a subtype of A, then the expression $(D)e$ is well-typed and its type is D. But since there is no way for the type checker to verify statically that the dynamic type of e is indeed D, a dynamic check is generated.

For example:

$$\frac{\mathcal{T} \vdash e : \ \mathit{Person}, \mathcal{T} \vdash \mathit{Employee} \ <: \ \mathit{Person}}{\mathcal{T} \vdash (\mathit{Employee})e : \ \mathit{Employee}}$$

Rule 16 (Static Rule for Up Casts).

$$\frac{\mathcal{T} \vdash e : \ A, \mathcal{T} \vdash A \ <: \ D}{\mathcal{T} \vdash (D)e : \ D}$$

This rule says that if the type of an expression e in the typing environment $\mathcal{T}$ is A and we can deduce from $\mathcal{T}$ that A is a subtype of D, then the expression $(D)e$ is well-typed and its type is D. This follows from the subsumption rule which says that if an expression has type A then it also has type D for any supertype D of A.

Note that the expression $(D)e$ in which the type of e is A will fail type checking unless $D <: A$ or $A <: D$.

The run time value of an expression e is denoted as $eval(e)$. Because of the flexibility of the assignment statement, the run time type of e is in general a subtype

of its static type. This fact is expressed in the following rule. Recall that *forName* is a method of the class *Class* which is in fact a class loader that produces a class object given the class name.

Rule 17 (Static and Dynamic Type).

$$\frac{\mathcal{T} \vdash e : A}{\mathcal{T} \vdash eval(e.getClass()) <: eval(Class.forName(''A''))}$$

A dynamic check that applies to an expression $(D)e$ where e has a statically determined type A is specified in the rule given below. The check is required to verify that the run-time type of the expression e is indeed a subtype of D. If that is not the case, the result of evaluation of the expression $(D)e$ is *ClassCastException*.

Rule 18 (Dynamic Rule for Down Casts).

$$\frac{\mathcal{T} \vdash e : A, \mathcal{T} \vdash D <: A, \quad \mathcal{T} \vdash eval(e.getClass()) \not<: eval(Class.forName(''D''))}{\mathcal{T} \vdash eval((D)e) : ClassCastException}$$

4.1.4 Parametric Types

The following rule specifies instantiation of a parametric class in which $e[D/T]$ stands for substitution of all free occurrences of T by D in the expression e. This rule says: if from the assumption that in the typing environment $\mathcal{T}$ T stands for a valid class we can conclude that the parametric class $C < T >$ is a valid class, and D is a valid class in $\mathcal{T}$, then $C < D >$ is a valid, type correct class the typing environment $\mathcal{T}$. For simplicity, we omit constructors.

Rule 19 (Parametric Types).

$$\frac{\mathcal{T} \vdash T <: Object \Rightarrow \quad \mathcal{T} \vdash \textbf{class } C < T > \{fields(C);\ methods(C)\} : Class, \quad \mathcal{T} \vdash D <: Object}{\mathcal{T} \vdash C < D > <: Object}$$

The rule that follows specifies the type signature of the class $C < D >$ obtained from the type signature for $C < T >$ by substituting D for T.

Rule 20 (Universal Instantiation).

$$\frac{\mathcal{T} \vdash T <: Object \Rightarrow \quad \mathcal{T} \vdash \textbf{class } C < T > \{fields(C);\ methods(C)\} : Class, \quad \mathcal{T} \vdash D <: Object}{\mathcal{T} \vdash Sig(C < D >) = \{fields(C)[D/T];\ methods(C)[D/T]\}}$$

For example, assume that we have

Sig(Collection<T>) = < T[] elements; boolean belongs(T);
void add(T); void remove(T) >.

Then

Sig(Collection<Employee>) = < Employee[] elements;
boolean belongs (Employee);
void add(Employee); void remove(Employee) >.

A more general case is instantiation of a bounded parametric class **class** $C < T$ **extends** $B >$.

Rule 21 (Bounded Parametric Types).

$$\frac{\mathcal{T} \vdash B <: \mathit{Object},\quad \mathcal{T} \vdash T <: B \Rightarrow \mathcal{T} \vdash \textbf{class } C < T \textbf{ extends } B > \{\mathit{fields}(C);\ \mathit{methods}(C)\} :\ \mathit{Class},\quad \mathcal{T} \vdash D <: B[D/T]}{\mathcal{T} \vdash C < D > <: \mathit{Object}}$$

The first premise in the above rule specifies that the bound B is a valid class in the typing environment $\mathcal{T}$. The second premise is that assuming that T stands for a class derived from B, $C < T$ **extends** $B >$ is a valid class in the typing environment $\mathcal{T}$. The last premise says that from the typing environment $\mathcal{T}$ we can deduce that D is a subtype of B, hence it qualifies as an actual type parameter for C. Under the above assumptions we can deduce from the typing environment $\mathcal{T}$ that $C < D >$ is a valid class in the typing environment $\mathcal{T}$. In the above rule B may be parametric, so that the type constraint may in fact be F-bounded, which is reflected in the condition $D <: B[D/T]$.

The type signatures of fields and methods of the instantiated parametric class $C < D >$ are determined by the same rule as for universal parametric polymorphism. They are obtained from the type signatures of C by substituting the actual type parameter D for the formal type parameter T in C. For example, assume that we have

Sig(OrderedCollection<T **extends** Comparable< T >>) = < T[] elements;
boolean belongs(T); void add(T); void remove(T) >.

Then if

Employee <: Comparable<Employee>

we will have

Sig(OrderedCollection<Employee>) = < Employee[] elements;
boolean belongs(Employee); void add(Employee); void remove(Employee) >.

4.2 Reflection

Formal type systems as presented so far deal primarily with the rules for static type checking. However, languages such as Java and C# have sophisticated dynamic type checking capabilities, as we showed in the rules for type casts. Furthermore, reflection necessarily requires dynamic type checks because full type information is available only at run time. In this section we present some of the formal rules for largely dynamic type checking required by reflection.

The static type checking rules when reflection is used are necessarily very limited because the specific type information is not available at compile time.

Consider first the field access. The static rule is:

Rule 22 (Static Rule for Field Access).

$$\frac{\mathcal{T} \vdash f : \mathit{Field}, \mathcal{T} \vdash \mathit{obj} : \mathit{Object}}{\mathcal{T} \vdash f.\mathit{get}(\mathit{obj}) : \mathit{Object}}$$

So if from a static typing environment $\mathcal{T}$ we can deduce that an expression f is of type *Field*, and the expression *obj* is of type *Object*, then in the same environment $\mathcal{T}$ the message $f.\mathit{get}(\mathit{obj})$ is well typed and its type is *Object*.

The dynamic rules are much more specific. A dynamic type checking rule given below specifies one of the dynamic checks related to accessing the value of a field. This check ensures that an object *obj* is indeed equipped with a field f. This check amounts to accessing the fields of the class of the object *obj* by evaluating the expression $\mathit{obj}.\mathit{getClass}().\mathit{getFields}()$ and checking whether the result of evaluation of f belongs to this array of fields.

Rule 23 (Dynamic Rule for Field Access).

$$\frac{\mathcal{T} \vdash f : \mathit{Field}, \mathcal{T} \vdash \mathit{obj} : \mathit{Object}, \quad \mathcal{T} \vdash \mathit{eval}(f) \notin \mathit{eval}(\mathit{obj}.\mathit{getClass}().\mathit{getFields}())}{\mathcal{T} \vdash \mathit{eval}(f.\mathit{get}(\mathit{obj})) : \mathit{IllegalArgumentException}}$$

Java reflective capabilities allow even a type safe update of the value of a field whose type was not known at compile time. But the static checks are then necessarily limited as illustrated by the rule for field update given below:

Rule 24 (Static Rule for Field Update).

$$\frac{\mathcal{T} \vdash f : \mathit{Field}, \mathcal{T} \vdash \mathit{obj} : \mathit{Object}, \mathcal{T} \vdash \mathit{val} : \mathit{Object},}{\mathcal{T} \vdash f.\mathit{set}(\mathit{obj}, \mathit{val}) : \mathit{void}}$$

The premises in the above rule are that the type of an expression f in the typing environment $\mathcal{T}$ is *Field*, the type of the expression *obj* is *Object*, and the type of the expression *val* is also *Object*. If so, then the message $f.\mathit{set}(\mathit{obj}, \mathit{val})$ setting the value of the receiver field f of an object *obj* to the value *val* will be well typed, and its type will be *void*, i.e., $f.\mathit{set}(\mathit{obj}, \mathit{val})$ is a correctly typed statement.

More specific type checking is possible only at run time. The dynamic type checking rule given below specifies a dynamic check similar to the check in the rule for field access. This rule verifies that an object referred to by *obj* indeed has a field referred to by f.

Rule 25 (Dynamic Rule for Field Update).

$$\frac{\mathcal{T} \vdash f : \mathit{Field}, \mathcal{T} \vdash \mathit{obj} : \mathit{Object}, \mathcal{T} \vdash \mathit{val} : \mathit{Object}, \quad \mathcal{T} \vdash \mathit{eval}(f) \notin \mathit{eval}(\mathit{obj}.\mathit{getClass}().\mathit{getFields}())}{\mathcal{T} \vdash \mathit{eval}(f.\mathit{set}(\mathit{obj}, \mathit{val})) : \mathit{IllegalArgumentException}}$$

The rule that follows specifies a dynamic check which ensures that assigning the value referred to by *val* to the field referred to by f of an object referred to by *obj* does not violate the typing rule for assignment. This test amounts to checking that the run time type of *val* obtained by evaluating the expression *val.getClass*() is a subtype of the run time type of the field f, where the latter is obtained by evaluating the expression f.*getType*(). If this condition is not satisfied, *IllegalArgumentException* will be thrown.

Rule 26 (Dynamic Rule for Field Update).

$$\frac{\mathcal{T} \vdash f : \mathit{Field}, \mathcal{T} \vdash \mathit{obj} : \mathit{Object}, \mathcal{T} \vdash \mathit{val} : \mathit{Object}, \quad \mathcal{T} \vdash \mathit{eval}(f) \in \mathit{eval}(\mathit{obj}.\mathit{getClass}().\mathit{getFields}()), \quad \mathcal{T} \vdash \mathit{eval}(\mathit{val}.\mathit{getClass}()) \not<: \mathit{eval}(f.\mathit{getType}())}{\mathcal{T} \vdash \mathit{eval}(f.\mathit{set}(\mathit{obj}, \mathit{val})) : \mathit{IllegalArgumentException}}$$

Given a class object c, the name of one of its methods *name*, and the array of types of its parameters *parameterTypes*, the expression

c.*getMethod*(*name*, *parameterTypes*) evaluates to a reference to a method and hence has type *Method*. However, static type checking of the expression

c.*getMethod*(*name*, *parameterTypes*) is very limited. The type checker can verify that c is a class. The name of the method is passed at rune-time as a string. This may be a complex string expression. The type checker can infer that the type of this expression is String, but it is still in general not possible to verify at compile time whether this string is a valid name of a method of class c. As for *parameterTypes*, all that the type checker can verify is that it is an array of classes. If these checks are successful, the checker will conclude that the type of the expression c.*getMethod*(*name*, *parameterTypes*) is *Method*.

Rule 27 (Static Rule for Method Introspection).

$$\frac{\mathcal{T} \vdash c : \mathit{Class}, \mathcal{T} \vdash \mathit{name} : \mathit{String}, \quad \mathcal{T} \vdash \mathit{parameterTypes} : \mathit{Class}[\,]}{\mathcal{T} \vdash c.\mathit{getMethod}(\mathit{name}, \mathit{parameterTypes}) : \mathit{Method}}$$

Because the specific type information is not available at compile time, several dynamic checks are required. Two dynamic rules for method introspection are given below. They show what kind of type violations are detected and the exceptions thrown. The first rule verifies that the specified method with the name specified in $c.getMethod(name, parameterTypes)$ does indeed exist. If not, *NoSuchMethodException* will be thrown when the expression $c.getMethod(name, parameterTypes)$ is evaluated.

Rule 28 (Dynamic Rule for Method Introspection).

$$\frac{\begin{array}{c}\mathcal{T} \vdash c : Class, \mathcal{T} \vdash name : String, \mathcal{T} \vdash parameterTypes : Class[\,], \\ \mathcal{T} \vdash (\forall i)(i \in \{i = 0, 1, 2, \ldots, c.getMethods().length - 1\} \Rightarrow \\ eval(name) \neq eval(c.getMethods()[i].getName())\end{array}}{\mathcal{T} \vdash eval(c.getMethod(name, parameterTypes)) : NoSuchMethodException}$$

The second rule given below specifies a dynamic check that the corresponding type parameters in $c.getMethod(name, parameterTypes)$ match the types of parameters of the method with the name *name* of the class c. Otherwise, *NoSuchMethodException* will be thrown.

Rule 29 (Dynamic Rule for Method Introspection).

$$\frac{\begin{array}{c}\mathcal{T} \vdash c : Class, \mathcal{T} \vdash name : String, \mathcal{T} \vdash parameterTypes : Class[\,], \\ \mathcal{T} \vdash (\exists i)((i \in \{0, 1, 2, \ldots, c.getMethods().length - 1\}) \\ (eval(name) = eval(c.getMethods()[i].getName())), \\ (\exists j)((j \in \{0, 1, 2, \ldots, c.getMethods()[i].getParameterTypes().length - 1\}) \\ (eval(parameters[j].getClass()) \not<: \\ eval(c.getMethods()[i].getParameterTypes()[j]))))\end{array}}{\mathcal{T} \vdash eval(c.getMethod(name, parameterTypes)) : NoSuchMethodException}$$

Java reflective capabilities allow type safe invocation of a method discovered at run time. The static rule checks that given a typing environment $\mathcal{T}$, in the expression $m.invoke(obj, args)$, m is of type *Method*, *obj* is of type *Object*, and *args* is an array of objects representing the actual arguments.

The static rule for method invocation using Java Core Reflection is:

Rule 30 (Static Rule for Method Invocation).

$$\frac{\mathcal{T} \vdash m : Method, \mathcal{T} \vdash obj : Object, \mathcal{T} \vdash args : Object[\,]}{\mathcal{T} \vdash m.invoke(obj, args) : Object}$$

Dynamic rules include the rules given below. The first rule specifies a dynamic check which verifies that *obj* refers to an object whose class is equipped with a method referred to by m. If not, the result of evaluation of $m.invoke(obj, args)$ will be *IllegalArgumentException*.

Rule 31 (Dynamic Rule for Method Invocation).

$$\frac{\mathcal{T} \vdash m : Method,\ \mathcal{T} \vdash obj : Object,\ \mathcal{T} \vdash args : Object[\,],\ \mathcal{T} \vdash eval(m) \notin eval(obj.getClass().getMethods())}{\mathcal{T} \vdash eval(m.invoke(obj, args)) : IllegalArgumentException}$$

The second rule given below specifies a dynamic check which ensures that the run time types of the actual arguments *args* are subtypes of the corresponding formal parameters types of the method referred by *m*. If not, the result of evaluation of $m.invoke(obj, args)$ is *IllegalArgumentException*.

Rule 32 (Dynamic Rule for Method Invocation).

$$\frac{\begin{array}{c}\mathcal{T} \vdash m : Method,\ \mathcal{T} \vdash obj : Object,\ \mathcal{T} \vdash args : Object[\,],\\ \mathcal{T} \vdash eval(m) \in eval(obj.getClass().getMethods()),\\ \mathcal{T} \vdash (\exists i)(i \in \{0, 1, \ldots, args.length - 1\} \Rightarrow\\ (eval(args(i).getClass()) \not<: eval(m.getParameterTypes()[i]))\end{array}}{\mathcal{T} \vdash eval(m.invoke(obj, args)) : IllegalArgumentException}$$

4.3 Type Erasure

4.3.1 Type Erasure Idiom

We consider primarily collection types that naturally require parametric polymorphism. Consider the legacy *Collection* class specified as follows:

```
class Collection {
 private Object[] elements;
 public boolean belongs(Object x){...}
 public int size(){...}
 public void add(Object x){...}
 public void remove(Object x) {...}
}
```

The class *Collection* given below is an example of universal parametric polymorphism. The type parameter *T* is assumed to be preceded with quantification over all types.

```
class Collection<T> {
 private T[] elements;
 public boolean belongs(T x){...}
```

```
 public int size(){...}
 public void add(T x){...}
 public void remove(T x) {...}
}
```

Looking at the type signature for *Collection* and *Collection<Employee>* it is easy to see the following:

– Collection<Employee> is not a subtype of Collection.
– Collection is not a subtype of Collection<Employee>.

The type signatures of the methods belongs, add, and remove in the class *Collection* are

```
boolean belongs(Object x)
void add(Object x)
void remove(Object x)
```

The type signatures of these methods in the class Collection<Employee> are

```
boolean belongs(Employee x)
void add(Employee x)
void remove(Employee x)
```

According to the typing rules that we introduced in this chapter and Chap. 1,

Collection $\not<:$ Collection<Employee> and Collection<Employee> $\not<:$ Collection. Assuming otherwise will lead to programs that compile and create run-time type errors. That is exactly the unfortunate consequence of the Java type erasure. In that type system assignments of an object of type Collection<Employee> in place of an object of type *Collection* is allowed. Likewise, assignment of an object of type *Collection* to an object of type Collection<Employee> is also allowed with a compiler warning.

The above example shows the implications of violation of static type checking rules. Dynamic violations due to type erasure are caused by the fact that the types Collection<Employee> and Collection<Department> are identified at run-time and their run-time type signature is in fact the signature of the class Collection. This makes it possible to assign an object of type Collection<Employee> to an object of type Collection<Department> and the other way around and the run-time type checking will not detect the type violation.

4.3.2 Static Type Checking

Here is an example showing how the type system is broken because of type erasure in a situation where a legacy class and a newly developed generic class interact. In the example below *LegacyClass* was compiled with a legacy compiler. When *TestLegacy* is compiled with the compiler based on type erasure, no warnings are issued. So a collection of departments is added to a collection of employees.

```
public class LegacyClass {
 public void addEmployees(Collection c) {
 for (int i=0; i < 5; i++)
      c.add(new Employee());
 }
}
public class TestLegacy {
 public static void main(String[] args) {
 Collection<Department> c =
      new LinkedList<Department>();
      new LegacyClass().addEmployees(c);
 }
}
```

In the example below the substitution of actual parameters of types LinkedList<Employee> and LinkedList<Department> for the formal parameters of type *LinkedList* is successfully performed following the type erasure rules. A compiler warning will be issued when the collectionA.addAll(collectionB) statement is compiled. But if legacy classes are involved, or even if a class is precompiled, those warnings will not be visible, and in either case the program will compile and fail at run-time at an unexpected place.

```
public class MessagePassing {
 public static void main(String[] args) {
 LinkedList<Employee> employees =
      new LinkedList<Employee>();
 LinkedList<Department> departments =
      new LinkedList<Department>();
 updateCollection(employees,departments);
 }
 public static void updateCollection
      (LinkedList collectionA, LinkedList collectionB) {
 collectionA.addAll(collectionB);
 }
}
```

Type erasure causes problems with well established overloading rules. This is illustrated by the following example of a class that cannot be compiled:

```
public class UpdateTransaction {
 public void update(Collection<Employee> employees) {... }
 public void update(Collection<Department> departments) {... }
}
```

The reason is that Collection<Employee> and Collection<Department> have the same type erasure which is in fact equivalent to *Collection*. Because of this, the standard overloading rules have been modified making them more complicated to take care of special cases caused by type erasure. But consider now two views specified as interfaces.

```
public interface EmployeeView {
 public Collection<Employee>
      select(Collection<Employee> x);
}
public interface DepartmentView
      extends EmployeeView {
 public Collection<Department>
      select(Collection<Department> x);
 }
public class Company
      implements DepartmentView {
 public Collection<Employee>
      select(Collection<Employee> x) {...}
 // public Collection<Department>
      // select(Collection<Department> x) {...}
 }
```

The class *Company* will not compile for a good reason: one of the abstract methods is not implemented. So the compiler requests implementation of both methods and the programmer responds by removing the comment signs. But now the class *Company* will not compile because of type erasure. Hence there is no way to implement *DepartmentView* although it type checks and compiles.

In a Java program updates are expressed by an assignment statement. But we show that type erasure allows objects of completely unrelated types to be assigned to each other in a major violation of the Java type system. The compiler issues a warning, but it will not reject the assignment statement. The consequence is that at run time, assignments that violate the type system will be executed. In addition, a Java program will fail at a completely unexpected place, so that a confusing run-time exception will not be caught in the program code. These assignments will generate compiler warnings but will be allowed. This is illustrated by the following example:

```
public class EmployeesAreDepartments {
 public static void main(String[] args) {
 LinkedList<Department> departments =
      new LinkedList<Department>();
 LinkedList<Employee> employees =
      new LinkedList<Employee>();
```

```
 LinkedList objects = employees;
 departments = objects;
 }
}
```

The assignment *objects = employees* does not create a compiler warning, and the assignment *departments = objects* does.

The problem is that the definition of subtyping <: in the recent versions of the Java type system is very different from the standard notion and hence the above rule has the implications illustrated by the following example. In this example, the method *updateCollection* whose formal parameter type is *LinkedList* is invoked with the argument whose type is LinkedList<Employee>. Likewise, the method *updateDepartment* whose formal type parameter is of type

LinkedList<Department> is invoked with the argument of type *LinkedList*. But *LinkedList<Department>* is not a subtype of *LinkedList* nor the other way around in any other type safe system.

```
public class UpdateTransaction {
 public static void main(String[] args) {
 LinkedList<Employee> employees =
      new LinkedList<Employee>();
 employees.add(new Employee("Joe"));
 LinkedList objects = new LinkedList();
 objects.add(new Object());
 updateCollection(employees);
 updateDepartment(objects);
 }
 public static void updateCollection(
      LinkedList collection) {
 collection.add(new Department());
 }
 public static void updateDepartment(
      LinkedList<Department> collection) {
 Department d = collection.getFirst();
 }
}
```

The compiler does not issue a warning in the method invocation

updateCollection(employees). The method *updateCollection* is type correct, but the compiler issues a warning in *collection.add(**new** Department())*. The compiler also issues a warning in the method invocation *updateDepartment(objects)*, but the whole program compiles in spite of type violations. The program executes up to *collection.getFirst()* where there should be nothing wrong and hence no exception handling is provided. The program thus fails due to a *ClassCastEception* error.

4.3.3 Dynamic Type Checking

The Java Virtual Machine includes a variety of dynamic type checks. These checks would in particular disallow assignments and passing arguments in method invocations that violate typing rules. Dynamic type checks are built into the virtual platform because Java includes a variety of features that cannot be type checked statically. But type erasure has the most negative impact in dynamic type checking.

The implications of type violations that are caused by type erasure are illustrated in the example below. The method *updateCollection* adds a *Department* object to a collection of objects of type *Employee*.

```
public class UpdateTransaction {
 public static void main(String[] args) {
 LinkedList<Employee> employees =
      new LinkedList<Employee>();
 updateCollection(employees);
 Employee emp = employees.remove();
 // runtime exception here
 }
 public static void updateCollection(Object collection) {
 ((LinkedList<Department>)collection).add(new Department());
 }
}
```

The compiler issues a warning regardless of whether the type cast is correct or not. So even if the type cast was the correct one *(LinkedList<Employee>)collection* the compiler would still issue a warning. If later on we attempt a perfectly correct statement *Employee emp = employees.remove()* it will create a run-time exception for inexplicable reasons.

It is important to point out the difference between the problems that are described above and the Java unsafe rule for arrays. Of course, the array type is naturally parametric. Java and C# have the rule that if B<:A then []B <: []A. Although this covariant subtyping rule for arrays does not guarantee type safety by static type checking, the run-time type information for arrays is correct, and dynamic checks that are in place work correctly.

4.3.4 Persistence by Serializability

Incorrect run-time type information has major implications when Java is used in persistent and database systems. When an object is promoted to persistence its type information also becomes persistent and will be incorrect because of type erasure. For example, an ordered collection of employees will be recorded in

the persistent store with the type Collection<Comparable>. Since type casts are not working correctly, major violations of the type system are possible when referencing persistent objects. The reason for this is that interpretation of their types may be completely wrong. These problems will be illustrated by the Java persistent mechanism. Recall that the type signatures for methods *writeObject* and *readObject* are:

```
void writeObject(Object obj)
Object readObject()
```

A simple program below promotes a collection of employees to persistence.

```
public class StoreCollectionObject {
 public static void main(String[] args)
      throws Exception {
 FileOutputStream fileout =
      new FileOutputStream("employees");
 ObjectOutputStream out =
      new ObjectOutputStream(fileout);
 Collection<Employee> employees =
      new LinkedList<Employee>();
 employees.add(new Employee());
 out.writeObject(employees);
 }
}
```

In a separate program given below an attempt to access this persistent collection is made in such a way that its type is misinterpreted on purpose. The compiler issues a warning in this type cast no matter what the type in the type cast is, even if it is the correct one (Collection<Employee>)in.readObject(). This is wrong because any type should pass a static check in the type cast because any type is a subtype of *Object*, which is the result type of *readObject*. Since the dynamic type information is incorrect, the incorrect type cast will succeed and the class cast exception will not be thrown.

```
public class ReadCollectionObject {
 public static void main(String[] args)
      throws Exception {
 FileInputStream filein =
      new FileInputStream("employees");
 ObjectInputStream in =
      new ObjectInputStream(filein);
 Collection<Department> departments = null;
 try { departments =
```

```
        (Collection<Department>) in.readObject();
        } catch (ClassCastException e) {exception handling }
}
```

A perfectly correct statement

```
Iterator<Department> it = departments.iterator();
Department d = it.next();
```

will fail later on where no exception handling is provided and hence the program will fail for inexplicable reasons. In this situation static type checking rules do not help much.

4.3.5 Reflection

Java reflective capabilities allow dynamic type introspection so that a program can perform actions based on types discovered at run-time. But we show that the most severe implications of type erasure are precisely in the Java reflective capabilities.

The following example is a transaction that interrogates the types dynamically and obtains wrong type information. Based on this incorrect information a program performs a wrong action. Java Core Reflection is not capable of detecting the type violation and hence it does not throw the illegal argument exception. The program thus fails at a completely unexpected place where no exception handling is provided. The compiler issues no warnings in this case as no static rules are violated.

```
public class ReflectiveTransaction {
 public void updateEmployees(
        LinkedList<Employee> collection) {
        Employee e = collection.remove();
 // unexpected ClassCastException!?
 }
 public static void main(String[] args) {
 ReflectiveTransaction trans =
        new ReflectiveTransaction();
 Collection<Department> departments =
        new LinkedList<Department>();
 departments.add(new Department());
 try {
        Method method =
        trans.getClass().getMethod
        (“updateEmployees”, departments.getClass());
        method.invoke(trans, departments);
        } catch (Exception e) {exception handling }
 }
}
```

4.4 Bibliographic Remarks

A good general reference on the formal rules for object-oriented type systems is [5]. A clear evidence on the implications of violating these rules is [8]. The formal rules presented here were published in [2]. A distinctive feature of these rules is that they apply to dynamic type checking and reflection. A foundational paper on parametric polymorphism in object-oriented language is [6].

The thorny problem of introducing parametric types in Java had several proposed solutions [1, 4, 7, 11–13]. Regrettably, the officially accepted solution that follows [4, 12] is based on the type erasure idiom which is provably wrong as shown here. The implications on static, and more so on dynamic type checking and reflection are nontrivial. The presentation of these problems given here was published in [2].

Contrary to the above problems of parametric types in Java, a correct solution was accepted for C# [10, 14].

4.5 Exercises

1. Specify the typing rule for the *for* statement as defined in Java. Specify the typing rules for the *foreach* statement as defined in C#.
2. Specify the typing rule for the *switch* statement.
3. Specify the typing rule for structures as defined in C#.
4. A record type R_B is a subtype of a record type R_A if R_B extends R_A in the following sense:

   – R_B has all the fields of R_A along with some additional fields.
   – The types of the common fields in R_B are subtypes of the types of the corresponding fields in R_A.

   Specify formally this subtyping rule.
5. Specify formally the subtyping rule for function types with n arguments. Recall that this rule is covariant in the result type and contravariant in the argument types. How does this rule apply to Java and C#?
6. Specify the subtyping rule for Java and C# interfaces viewing an interface type as a record of function types.
7. Specify the subtyping rule for immutable and mutable arrays. How does this rule apply to Java and C# arrays?
8. The body of a method is a block, i.e., it consists of a sequence of declarations and a sequence of statements. Specify the formal typing rule for blocks. Assume that a block contains return statements.
9. Specify the typing rule for methods with n arguments. This rule requires usage of the typing rule for blocks. In addition, it must take into account the typing information for the formal parameters of methods.

10. Specify the formal typing rule for a message, i.e., an invocation of a method with *n* arguments.
11. A class may be viewed as a record of fields and a record of functions representing methods. Specify the formal rule for a class B to be a subtype of the class A. Recall that the types of the inherited fields remain invariant and that the overridden methods must satisfy the subtyping rules for function types. How does this rule apply to Java and C# classes?
12. Specify the type erasure of a parametric class so that it would apply to the interface *OrderedCollection* given below. Consult Java Language Specification.

```
public interface OrderedCollection<T extends Comparable<T>>
                 extends Collection<T> {
// . . .
}
```

    Compare the type signature of the interface OrderedCollection<Employee> obtained by standard notion of substitution of the actual type parameter *Employee* for the formal type parameter T with the type signature of the interface OrderedCollection<Employee> obtained by type erasure.
13. Standard requirements for overloading require that two methods of the same class must have different type signatures of their arguments, i.e. either the types or the number of arguments must be different. Java overloading rules have been changed because of type erasure so that the following class will not compile. Specify the current Java overloading rules. Consult [3, 9].

```
public class UpdateTransaction {
 public void update(Collection<Employee> employees) {. . . }
 public void update(Collection<Department> departments) {. . . }
}
```

References

1. O. Agesen, S.N. Freund, J.C. Mitchell, Adding type parameterization to the Java language, in *Proceedings of OOPSLA '97* (ACM, New York, 1997), pp. 49–65
2. S. Alagić, M. Royer, Genericity in Java: persistent and database systems implications. VLDB J. **17**(4), 847–878 (2007)
3. K. Arnold, J. Gosling, D. Holmes, *The Java Programming Language*, 4th edn. (Addison-Wesley, Boston, 2005)
4. G. Bracha, M. Odersky, D Stoutmire, P. Wadler, Making the future safe for the past: adding genericity to the Java programming language, in *Proceedings of OOPSLA 1998* (ACM, New York, 1998), pp. 183–200
5. K. Bruce, *Foundations of Object-Oriented Languages* (MIT Press, Cambridge, 2002)
6. P. Canning, W. Cook, W. Hill, W. Olthoff, J. Mitchell, F-bounded polymorphism for object-oriented programming languages, in *Proceedings of Functional Programming Languages and Computer Architecture* (ACM, New York, 1989), pp. 273–280

7. R. Cartwright, G.L. Steele, Compatible genericity with run-time types for the Java programming language, in *Proceedings of OOPSLA '98* (1998), pp. 201–218
8. W.R. Cook, A proposal for making Eiffel type safe. Comput. J. **32**, 305–311 (1989)
9. J. Gosling, B. Joy, G. Steel, G. Bracha, *The Java Language Specification*, 3rd edn. (Prentice Hall, New Jersey, 2005)
10. A. Kennedy, D. Syme, Design and implementation of generics for the .NET Common Language Runtime, in *Proceedings of PLDI* (ACM, New York, 2001), pp. 1–12
11. A. Myers, J. Bank, B. Liskov, Parameterized types for Java, in *Proceedings of POPL* (ACM, New York, 1997), pp. 132–145
12. M. Odersky, P. Wadler, Pizza into Java: translating theory into practice, in *Proceedings of POPL* (ACM, New York, 1997), pp. 146–159
13. J. Solorzano, S. Alagić, Parametric polymorphism for Java^{TM}: a reflective solution, in *Proceedings of OOPSLA '98* (ACM, New York, 1998), pp. 216–225
14. D. Yu, A. Kennedy, D. Syme, Formalization of generics for .NET Common Language Runtime, in *Proceedings of POPL* (ACM, New York, 2004), pp. 39–63

Chapter 5
Concurrent Models

In this chapter we consider concurrent object-oriented models. In section we present the Java model of concurrent threads of execution. In Sect. 5.1.1 we introduce thread objects. Section 5.1.2 explains the basic mechanism of synchronizing concurrent access of multiple threads to objects on the common heap. In Sect. 5.1.3 we discuss the relationship between synchronization and inheritance and in Sect. 5.1.4 the relationship between serialization and synchronization. In Sect. 5.1.5 we demonstrate what the implications are of synchronized versus unsynchronized executions.

Section 5.2 leads to a more general concurrent and distributed model based on asynchronous messages. The basis of this model is the fact that messages may be viewed as objects, which leads to a hierarchy of message types presented in Sect. 5.2.1. In Sect. 5.2.2 we extend the previously specified typing rules by introducing typing rules for messages of different types. In this new framework it is natural to define formally the rules of behavioral subtyping in Sect. 5.2.3.

In Sect. 5.3 we introduce a very general model of concurrent active objects, each equipped with its own virtual machine. Ambients of such objects are discussed in Sect. 5.3.1. Reflective capabilities for support of concurrent objects are specified in Sect. 5.3.2. A distinctive feature of the reflection as presented here is the presence of assertions. Finally, in Sect. 5.3.3 we specify the virtual machine for concurrent objects able to execute various types of asynchronous messages.

5.1 Concurrent Threads

5.1.1 Thread Objects

Java introduced the notion that a thread of execution is an object. As such, it is created dynamically as all objects are. A thread object has a state (of execution) and it is equipped with methods that introspect and change the thread object state. This

S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0_5

way a Java program can have multiple threads of execution that exist concurrently and contribute to the overall outcome of the program execution.

The core of the Java model of concurrent threads is based on the interface *Runnable* and the class *Thread*. The interface *Runnable* contains only one method *run*. Its implementation in a specific class specifies the actual thread execution process.

```
public interface Runnable {
 void run();
}
```

The class *Thread* has a constructor that takes a *Runnable* object as the argument and creates a thread of execution. The method *run* in the class *Thread* has an empty implementation so that it must be overridden in a specific class. Thread execution is started by invoking the method *start*. A thread execution can be interrupted, but the basic idea of the Java model is that all threads created by a program should run to their completion.

```
public class Thread
      extends Object, implements Runnable {
 public Thread(Runnable target);
 public void start();
 public void run();
 public void interrupt();
 // other methods
}
```

An example of using the interface *Runnable* and the class *Thread* is the class *TestRun* given below.

```
class TestRun implements Runnable {
 private long firstNumber;
 public TestRun(long firstNumber) {
 this.firstNumber = firstNumber;
 }
 public void run() {
 // compute suitable numbers larger than firstNumber
 ...
 }
}
```

In the code given below, an object of the class *TestRun* is created, as well as a new *Thread* object. The *Thread* constructor takes an object of *TestRun* as its argument. The newly created thread is then started.

```
 TestRun p = new TestRun(147);
 new Thread(p).start();
```

5.1.2 Synchronized Objects

Existence of multiple concurrent threads that access objects on the common heap creates some well-known problems. Actions of two threads performed concurrently on the same object may produce incorrect results such as incorrect updates or incorrect results of introspection of the object state. This is why concurrent access to objects must be controlled to avoid these problems.

A well known approach is illustrated below by the class *SynchronizedObject*. The methods that access and modify the hidden object state are declared as synchronized. This means that a thread executing one of these methods gets exclusive access to the underlying object state. The object state is made available to other threads when the method completes it execution. This basic model is extended with a more sophisticated synchronization protocol for concurrent threads accessing the same object.

```
public class SynchronizedObject {
 private Object state;
 public SynchronizedObject(Object initialState) {
      state=initialState;
 }
 public synchronized Object get() {
      return state; }
 public synchronized void set(Object obj) {
      state=obj ;}
 // methods inherited from Object:
 // public wait()
 // public void notifyAll()
 // other methods
}
```

The undesirable effect of unsynchronized access of two threads to the same object is illustrated in Fig. 5.1. The update of Thread 1 will be lost.

The effects of synchronized access that avoids the above problem is illustrated in Fig. 5.2.

A more sophisticated scheduling strategy for concurrent access is based on the methods *wait* and *notifyAll* inherited from the root class *Object*. This model is illustrated by the parametric class *SynchronizedContainer*. Adding new elements to the container by a synchronized method *add* works in accordance with the synchronization access model described above. The container is assumed to be unbounded. However, removing an element from the container is possible only if a container is not empty. This is why a thread that attempts to remove an element from an empty container is put in a wait state. The method *add* will send a message *notifyAll* to all waiting threads when it successfully adds an element to the container. One of those threads waiting for this message will be selected nondeterministically and its remove action will be performed.

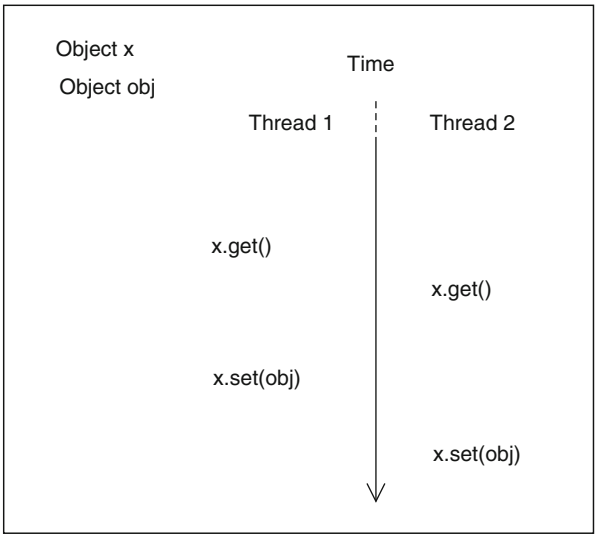


Fig. 5.1 Unsynchronized object access

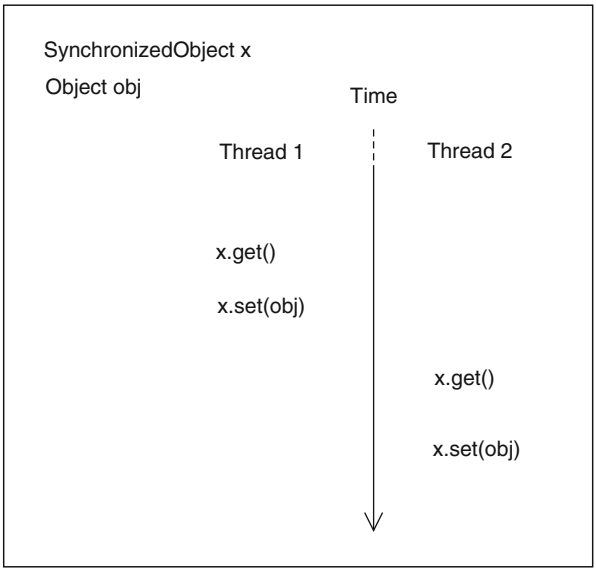


Fig. 5.2 Synchronized object access

```
class SynchronizedContainer<T> {
 private Container<T> container = new Container<T>;
 public synchronized void add(T x) {
       container.add(x);
       notifyAll();
}
 public synchronized void remove(T x) {
       throws InterruptedException;
 { while (container.size() = 0)
       wait();
       container.remove(x);
 }
}
```

5.1.3 Synchronization and Inheritance

Many classes are developed with no considerations for possible concurrent access to their objects. Producing a class that allows concurrent access from a class that does not is accomplished by inheritance. A class that enforces synchronized access is derived from the base class by overriding the inherited methods and declaring them as synchronized. The bodies of these synchronized methods can now just invoke the corresponding methods in the base class. This is illustrated by the classes *OrderedCollection* and *OrderedCollectionSync*. Note the general rule that the synchronized property of a method is not inherited. In this example, the methods in *OrderedCollection* are not synchronized, and their overridden versions in *OrderedCollectionSync* are synchronized. The class *OrderedCollectionSync* is derived by inheritance from the class *OrderedCollection* by overriding all the inherited methods, declaring them as synchronized, and invoking the methods in the superclass.

```
public class OrderedCollectionSync< T extends Comparable<T>>
              extends OrderedCollection<T> {
 public OrderedCollectionSync() { super(); }
@Override
 public synchronized boolean contains(Object e) {
 if (e ≠ null) {return super.contains(e); }
 else return false;
}
@Override
 public synchronized void add(T e) {
 super.add(e); }
```

```
@Override
public synchronized void remove(Object e) {
 super.remove(e); }
}
```

5.1.4 Concurrency and Serialization

In this section we show how a complex object is serialized and deserialized using synchronized methods. The appropriate exception handling is also specified. This makes serialization and deserialization safe in a concurrent environment. The class *RunwayQueue* extends the class *Queue* and introduces synchronized methods for serialization and deserialization of the *RunwayQueue* objects.

The Java type constraint T **extends** Aircraft & Serializable guarantees two things. The first is that the objects in the runway queue are aircraft objects (i.e. objects of subtypes of the type *Aircraft*). The second constraint guarantees that these objects are serializable. This is an instance of multiple inheritance as available in Java. The actual parameter corresponding to the formal type parameter T must be a class that extends the aircraft class and implements the interface *Serializable*. In Java and C# single inheritance applies to classes and multiple inheritance to interfaces. In this case, a class extends a single class and implements an interface. This is illustrated in Fig. 5.3. This way serializing the runway queue will include serializing the objects in the queue as well.

```
public class RunwayQueue<T extends Aircraft & Serializable >
      extends Queue<T> implements Serializable {
// fields, constructors and methods
  public synchronized void serialize()
{  try {
      FileOutputStream fileStream = new FileOutputStream("RunwayData");
      ObjectOutputStream objectStream = new ObjectOutputStream(fileStream);
      objectStream.writeObject(this);
      objectStream.close();
 }
```

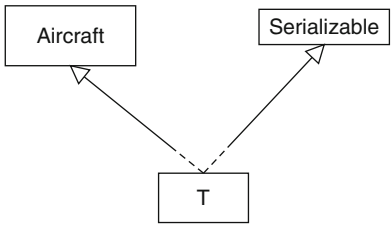


Fig. 5.3 Multiple inheritance

```
 catch ( FileNotFoundException fileEx )
      {exception handling }
 catch ( IOException ioEx )
      { exception handling }
}
  public synchronized RunwayQueue<T> deserialize()
{// symmetric code
 }
}
```

5.1.5 Synchronized Versus Unsynchronized Executions

The impact of synchronized versus unsynchronized access to collection objects is illustrated by a class *Testrun*. Its *run* method performs insertion of integers in the given range but then immediately deletes the odd ones. Synchronized access to the collection object is accomplished by the Java synchronized statement. The synchronized statement guarantees exclusive access to the synchronized object by a single thread. This will guarantee that the resulting collection object will contain only even integers. In the absence of the synchronize statement multiple threads will produce wrong results, i.e., a mix of odd and even integers in the collection object.

```
public class TestRun implements Runnable {
 private OrderedCollection<Integer> collection;
 public TestRun(OrderedCollection<Integer> collection) {
      this.collection = collection;
 }
@Override
public void run() {
 //insertions to the list (0–100) and deletion of odd insertions
 synchronized(collection) {
      Integer toAdd;
      for (int i = 0; i < 100; i++) {
      toAdd = new Integer(i);
      collection.add(toAdd);
      if (i div 2 = 0)
      {collection.remove(new Integer(i -1));}
           }
      }
 }
}
```

In this example we need methods that are defined in a somewhat more complete class *Thread* given below. A thread can be interrupted, for example when a thread

execution is suspended by invoking the method *sleep* or the method *wait*. These methods will throw *InterruptedException*. Invocation of the method *join* on a thread object will make the current execution wait until the thread completes its execution.

```
public class Thread
      extends Object, implements Runnable {
 public Thread(Runnable target);
 public void interrupt();
 public boolean isInterrupted();
 public void join()
      throws InterruptedException;
 public void run();
 public static void sleep()
      throws InterruptedException;
 public void start();
 public static void yield();
 // other methods
}
```

In the main method given below two threads are created and started. The Java *join* statement is used to wait until these two threads complete. The underlying collection is not equipped with synchronized methods, but the synchronized statement will guarantee correct results. Otherwise, the resulting collection will contain both odd and even numbers.

```
public static void main(String[] args) {
 private OrderedCollection<Integer> intCollection =
      new OrderedCollection<Integer>();
 Thread thread1 = new Thread(new TestRun(intCollection));
 Thread thread2 = new Thread(new TestRun(intCollection));
 thread1.start();
 thread2.start();
 try {//wait for threads to finish before checking the collection
      thread1.join();
      thread2.join();
 } catch (InterruptedException e) { exception handling }
 for (Integer i: intCollection)
      System.out.print(i);
 }
```

An alternative solution to guarantee a correct outcome is to use the previously defined synchronized collection as in the code below.

```
OrderedCollectionSync<Integer> syncIntCollection =
    new OrderedCollectionSync<Integer>();
Thread thread1 = new Thread(new TestRun(syncIntCollection));
Thread thread2 = new Thread(new TestRun(syncIntCollection));
```

5.2 Messages as Objects

5.2.1 Types of Messages

A message in mainstream object-oriented languages such as Java or C# is specified in a functional notation as an application of a method to its arguments where one of them is distinguished as the receiver. This follows the traditional notation of procedural and functional languages. But in fact, dynamic binding makes the semantics of message sends quite different. The functional view for message send does not reflect correctly the fact that there are different types of messages.

The functional view fits the observer and the constructor messages. An observer message just reports the properties of the hidden object state and a constructor message constructs an object of the specified result type. In both cases a message has arguments and a result and hence represents an expression of the result type.

Other categories of messages do not fit the functional notation. A mutator message is a message that changes the state of the receiver and possibly other objects as well. A mutator message does not have a result and its semantics does not fit the functional notation. An object creation message is sent to a class object and its semantics does not include dynamic binding of a message to a method. This is why this type of a message is expressed in a different notation.

An asynchronous message in general does not have a result and hence the functional notation is not appropriate. In addition, an asynchronous message has a lifetime and specific types of asynchronous messages have other attributes. A particular type of an asynchronous message has a result but this result is not necessarily immediately available at the point of the message send. In fact, a two way message has an attribute called *future* which makes it possible to access the result of a two way message but only when it is actually available.

A transient message has a discovery life time. If a transient message is not scheduled for execution within its discovery life time, it will expire and it will never be executed. A sustained message is a message that does not have this limitation. A message may be one-to-one with a single receiver or a message may be a broadcast message sent to a set of receiver objects. Many messages naturally combine the features of the above mentioned message types. For example, a two-way transient message, a one-to-one sustained message, a one-to-many mutator message, etc.

The above informal analysis shows that messages have different types. Different types of messages can be naturally structured into a hierarchy of types including multiple inheritance. A message send is interpreted as a creation of an object of the appropriate message type. Beyond that point a message object behaves according to the specification of its type. Specifying the behavior of objects of a message type is naturally done using an object-oriented assertion language. The assertion language should be based on a logic that fits this object-oriented paradigm in which messages are viewed as typed objects.

Non-functional messages in this paradigm are objects. A message is created dynamically and it has a unique identifier like any other object. In the concurrent architecture described in Sect. 5.3.1 object identifiers must be global. The attributes of a message are the receiver object and the array of arguments along with a reference to a method. Messages of specific subtypes will have other attributes. This produces a hierarchy of message types that are subtypes of the type *Message*:

```
interface Message {
 Method m();
 Object receiver();
 Object[] arguments();
 int timeStamp();
}
```

When a message object is created its time stamp is recorded. The implementing class would have a constructor:

```
class MessageObject implements Message {
 MessageObject(Method m, Object receiver,
               Object[] arguments);
 int timeStamp();
 Method m();
 Object receiver();
 Object[] arguments();
}
```

Creating a message could be done just like for all other objects:

```
Message msg = new MessageObject(Method m, Object receiver,
                                Object[] arguments)
```

This implies message send in the underlying implementation. However, *Message* and *MessageObject* belong to the reflective core of this language along with *Class*, *Method*, and *Constructor*. These types should be final in order to guarantee type safety at run-time. So an alternative is to have a special notation to create an

asynchronous message. A functional (and hence synchronous) message is denoted using the usual dot notation:

x.m(a1,a2,. . .,an)

A non-functional (asynchronous etc.) message would be created as follows:

Message msg = x ⇐ m(a1,a2,. . .,an)

In general, an asynchronous message does not have a result. The basic type of a message is point-to-point, one-way, and immediately executed. This type of a message could be expressed in a traditional notation

receiver.m(arguments)

In the new paradigm the result of an asynchronous message send is a reference to the created message object.

A mutator message is a message that mutates the state of the receiver object and possibly other objects as well. A mutator message does not have a result, hence we have:

interface MutatorMessage **extends** Message {. . .}

A special notation for a mutator message is

X <:= m(a1,a2,. . .,an)

The type of this expression is *MutatorMessage*.

A two way message requires a response which communicates the result of a message. The result is produced by invoking the method *future* on a two way message. This method has a precondition which is that the future is resolved, i.e., that it contains the response to the message.

```
interface TwoWayMessage extends Message {
 boolean futureResolved();
 boolean setFuture(Object result);
 Object future() requires this.futureResolved();
}
```

The implementing class would contain a constructor which takes the reply interval as one of its parameters.

```
class TwoWayMessageObject implements TwoWayMessage {
 TwoWayMessageObject(Method m, Object receiver, Object[] arguments,
      int replyInterval);
 boolean futureResolved();
 boolean setFuture(Object result);
 Object future() requires this.futureResolved();
}
```

A one-to-many message is of the type *BroadcastMessage* and it is sent to multiple objects. A suggestive notation for a one to many message is <=>>.

A transient message has a discovery life time specified as a finite time interval. If a message is not discovered and scheduled for execution before its discovery time has expired, the message will be regarded as expired and will never be scheduled for execution. The discovery life time will be specified in the constructor of the implementing class:

```
class TransientMessageObject implements TransientMessage {
 TransientMessageObject(Method m, Object receiver, Object[] arguments,
      int discoveryLifeTime);
 int discoveryLifeTime();
}
```

A suggestive notation for a transient message is <=|. A sustained message (i.e. a message whose discovery time is not limited) denoted as <=∼ is specified by a special message type:

```
interface SustainedMessage extends Message {. . .}
```

Object creation is expressed by creating a message object of the type *CreateMessage*:

```
interface CreateMessage extends Message {
      Object newObject();
}
```

So if *C* is a class the expression

C ⇐!**new**(a1,a2,. . . ,an)

is of type *CreateMessage*.

A variety of non-functional message types can now be expressed via interfaces as illustrated in Fig. 5.4.

5.2.2 Typing Rules

In this section we present some of the typing rules for a language that supports the notion of a message as a typed object, described so far. This set of rules is by no means complete. We focus on the rules that apply to the distinctive features of this paradigm: messages, reflection and assertions. Many other standard rules are omitted. The rules for typing classes follow the typing notation introduced in Chap. 4 and they are based on the availability of reflection. The basic reflective classes are *Class*, *Method*, *Constructor*, *Message* and its subtypes, *Assertion* and *Expression* with their subtypes.

An important point is that the static and the run-time type systems contain equivalent type info, hence both typing environments are denoted as $\mathcal{T}$.

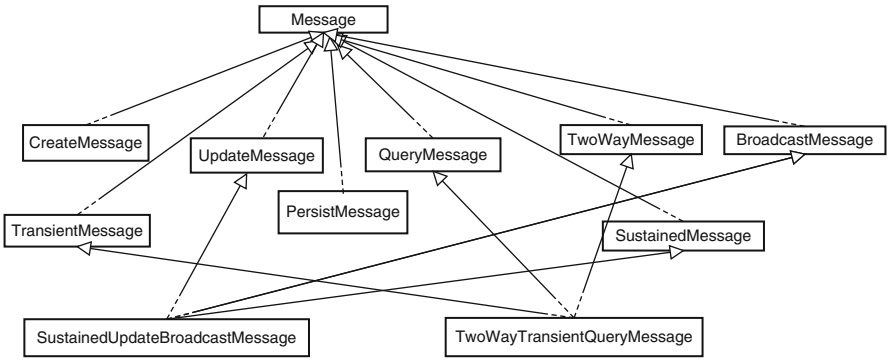


Fig. 5.4 Message type hierarchy

The first rulebelow specifies a well-typed class. An expression that is a well-typed specification of a class is of the type *Class*. The main difference in comparison with the rule given in Chap. 4 is the presence of an invariant.

$$\begin{gathered}\mathcal{T} \vdash C : \mathit{Class},\\ \mathcal{T} \vdash \mathit{constructor}(D) : \mathit{Constructor},\\ \mathcal{T} \vdash \mathit{methods}(D) : \mathit{Method}[\,],\\ \mathcal{T} \vdash \mathit{methods}(D)\ \mathit{compatibleWith}\ \mathit{methods}(C),\\ \mathcal{T} \vdash \mathit{invariant}(D) : \mathit{Assertion}\end{gathered}$$

$$\begin{gathered}\mathcal{T} \vdash \textbf{class}\ D\ \textbf{extends}\ C\\ \{\mathit{constructor}(D); \mathit{methods}(D); \mathit{invariant}(D)\} : \mathit{Class}\end{gathered}$$

A special case in the above rule is D=Object. The relation *compatibleWith* (which we do not elaborate fully) in the above rule captures the standard requirements for type signatures of inherited methods that we already discussed in Chap. 4. *compatibleWith* specifies that method signatures are inherited as specified in the superclass. The only redefinition of the inherited method signatures is covariant change of the result type. With *compatibleWith* defined like that we have the following implication as in Chap. 4:

$$\begin{gathered}\mathcal{T} \vdash C : \ \mathit{Class},\\ \mathcal{T} \vdash A\ m(A1, A2, \ldots, An)\{\ldots\} \in \mathit{methods}(C),\\ \mathcal{T} \vdash \textbf{class}\ D\ \textbf{extends}\ C : \ \mathit{Class},\\ \mathcal{T} \vdash B\ m(A1, A2, \ldots, An)\{\ldots\} \in \mathit{methods}(D)\end{gathered}$$

$$\mathcal{T} \vdash B <: A$$

A class expression which specifies a well-typed parametric class with the type parameter T and the bound type B is of the type $\mathit{Class} < T \ <: \ B >$ where the bound B could also be parametric. We do not give the full typing rules for parametric classes as they have been already discussed in Chap. 4. We just give the rule below that governs instantiation of parametric classes. Recall that $e < D/T >$ denotes

an expression e in which all free occurrences of T are replaced by D. The main difference in comparison with the previous rules is that substitution applies to the invariant.

$$\frac{\begin{array}{c}\mathcal{T} \vdash \textbf{class } C < T \textbf{ extends } B > \\ \{constructor(C); methods(C); invariant(C)\} : Class < T \ <: \ B >, \\ \mathcal{T} \vdash D <: B < D/T >\end{array}}{\begin{array}{c}\mathcal{T} \vdash \textbf{class } C < D > \\ \{constructor(C) < D/T >; methods(C) < D/T >; invariant(C) < D/T >\} : \\ Class\end{array}}$$

The rule for typing method bodies is given below. This is a rule that we did not present in Chap. 4. In addition, in the rule below we have the precondition, the postcondition and the modifies list. These assertions and the expression representing the method body are type checked in an environment that is obtained from the environment of the method extending it by bindings for formal parameters. The symbol for union $\cup$ is used to denote extending the typing environment with new bindings.

$$\frac{\begin{array}{c}\mathcal{T} \vdash B : \ Class, \mathcal{T} \vdash Ai : \ Class \ for \ i = 1, 2, \ldots, n, \\ \mathcal{T} \cup \{x1 : A1, x2 : A2, \ldots, xn : An\} \vdash preCondition(m) : Assertion, \\ \mathcal{T} \cup \{x1 : A1, x2 : A2, \ldots, xn : An\} \vdash postCondition(m) : Assertion, \\ \mathcal{T} \cup \{x1 : A1, x2 : A2, \ldots, xn : An\} \vdash modifyList(m) : List < modifyPair >, \\ \mathcal{T} \cup \{x1 : A1, x2 : A2, \ldots, xn : An\} \vdash expression(m) : B\end{array}}{\begin{array}{c}\mathcal{T} \vdash B \ m(A1 \ x1, A2 \ x2, \ldots, An \ xn) \\ \{preCondition(m); postCondition(m); modifyList(m); return \ expression(m)\} : \\ Method\end{array}}$$

The typing rule for functional messages is the usual one:

$$\frac{\begin{array}{c}\mathcal{T} \vdash x : A, \\ \mathcal{T} \vdash C \ m(D1, D2, \ldots, Dn)\{\ldots\} \in methods(A), \\ \mathcal{T} \vdash ei : Ei \ for \ i = 1, 2, \ldots, n, \\ \mathcal{T} \vdash Ei <: Di \ for \ i = 1, 2, \ldots, n\end{array}}{\mathcal{T} \vdash x.m(e1, e2, \ldots, en) : \ C}$$

The typing rule for asynchronous messages is somewhat different. In general, an asynchronous message does not have a result. It creates a new message object and hence the type of an asynchronous message expression $x <= m(e1,e2,\ldots,en)$ is *Message*.

$$\frac{\begin{array}{c}\mathcal{T} \vdash x : A, \\ \mathcal{T} \vdash C \ m(D1, D2, \ldots, Dn)\{\ldots\} \in methods(A), \\ \mathcal{T} \vdash ei : Ei \ for \ i = 1, 2, \ldots, n, \\ \mathcal{T} \vdash Ei <: Di \ for \ i = 1, 2, \ldots, n\end{array}}{\mathcal{T} \vdash x <= m(e1, e2, \ldots, en) : \ Message}$$

The rules for other types of messages would follow the above pattern. For example, the rule for a one to many asynchronous message has the following form:

$$\frac{\begin{array}{c}\mathcal{T} \vdash x : A,\\ \mathcal{T} \vdash C\ m(D1, D2, \ldots, Dn)\{\ldots\} \in methods(A),\\ \mathcal{T} \vdash ei : Ei\ for\ i = 1, 2, \ldots, n,\\ \mathcal{T} \vdash Ei <: Di\ for\ i = 1, 2, \ldots, n\end{array}}{\mathcal{T} \vdash x <=>> m(e1, e2, \ldots, en) :\ BroadcastMessage}$$

The typing rule for a message representing creation of a new object of a class C has the following form:

$$\frac{\begin{array}{c}\mathcal{T} \vdash C :\ Class,\\ \mathcal{T} \vdash constructor(C) : C(D1, D2, \ldots, Dn),\\ \mathcal{T} \vdash ei : Ei\ for\ i{=}1, 2, \ldots, n,\\ \mathcal{T} \vdash Ei <: Di\ for\ i{=}1, 2, \ldots, n\end{array}}{\mathcal{T} \vdash C\ \ <=!\mathbf{new}(e1, e2, \ldots, en) :\ CreateMessage}$$

The typing rules for assertions that are first-order predicates have the following form:

$$\frac{\mathcal{T} \cup \{x : C\} \vdash e : Assertion}{\mathcal{T} \vdash (\forall\ C\ x)(e) : Assertion}$$

5.2.3 Behavioral Subtyping

The inheritance rules for assertions follow the discipline of behavioral subtyping. These rules were discussed previously, and now we can put them in a formal notation.

A subclass can only strengthen the inherited invariant from the superclass (usually by conjunction). An invariant applies to all object states, hence all variables in an invariant are implicitly universally quantified. An invariant in a subclass typically introduces additional bound variable in comparison with the bound variables of the superclass. For any type correct substitution of bound variables, the invariant in the subclass implies the invariant of the superclass.

$$\frac{\begin{array}{c}\mathcal{T} \vdash \mathbf{class}\ D\ \mathbf{extends}\ C :\ Class,\\ \mathcal{T} \vdash boundVariables(invariant(D)) = \{x_1, x_2, \ldots, x_n\},\\ \mathcal{T} \vdash boundVariables(invariant(C)) \subset boundVariables(invariant(D)),\\ \mathcal{T} \vdash x_i : A_i \text{ for } i = 1, 2, \ldots, n,\\ \mathcal{T} \vdash a_i : A_i \text{ for } i = 1, 2, \ldots, n\end{array}}{invariant(D)[a_i/x_i]\ \Rightarrow\ invariant(C)[a_i/x_i]}$$

The rules of behavioral subtyping require that the precondition of an inherited method is weaker in the subclass in comparison with the precondition of that method in the superclass (contravariance). A typical implementation of this rule is that the preconditions of the inherited methods remain invariant in a subclass. A precondition contains free variables, such as those that represent arguments of a method call. The rule below specifies only the requirement for those variables. A precondition may also contain bound variables, but we omit them from the rule below as they are treated as in invariants.

$$\frac{\begin{array}{c}\mathcal{T} \vdash \textbf{class } D \textbf{ extends } C : \mathit{Class},\\ \mathcal{T} \vdash A\ m(A1, A2, \ldots, An)\{\ldots\} \in \mathit{methods}(C),\\ \mathcal{T} \vdash \mathit{freeVariables}(\mathit{precondition}_D(m)) = \{x_1, x_2, \ldots, x_n\},\\ \mathcal{T} \vdash \mathit{freeVariables}(\mathit{precondition}_C(m)) \subseteq \mathit{freeVariables}(\mathit{precondition}_D(m)),\\ \mathcal{T} \vdash x_i : A_i \text{ for } i = 1, 2, \ldots, n,\\ \mathcal{T} \vdash a_i : A_i \text{ for } i = 1, 2, \ldots, n\end{array}}{\mathit{preCondition}_D(m)[a_i/x_i] \;\Leftrightarrow\; \mathit{preCondition}_C(m)[a_i/x_i]}$$

The post condition of an inherited method may be strengthened in a subclass (usually by conjunction):

$$\frac{\begin{array}{c}\mathcal{T} \vdash \textbf{class } D \textbf{ extends } C : \mathit{Class},\\ \mathcal{T} \vdash A\ m(A1, A2, \ldots, An)\{\ldots\} \in \mathit{methods}(C),\\ \mathcal{T} \vdash \mathit{freeVariables}(\mathit{postcondition}_D(m)) = \{x_1, x_2, \ldots, x_n\},\\ \mathcal{T} \vdash \mathit{freeVariables}(\mathit{postcondition}_C(m)) \subset \mathit{freeVariables}(\mathit{postcondition}_D(m)),\\ \mathcal{T} \vdash x_i : A_i \text{ for } i = 1, 2, \ldots, n,\\ \mathcal{T} \vdash a_i : A_i \text{ for } i = 1, 2, \ldots n\end{array}}{\mathit{postCondition}_D(m)[a_i/x_i] \;\Rightarrow\; \mathit{postCondition}_C(m)[a_i/x_i]}$$

5.3 Concurrent Objects

5.3.1 Ambients of Concurrent Objects

In this section we describe the environments based on the view of messages as typed objects. An ambient is a dynamic collection of service objects. When a message is sent to an ambient object, one or more service objects is selected depending upon the type of the message, and the message is sent to those service objects. The types of service objects are assumed to be derived from the type *ServiceObject*. Thus the class *Ambient* is parametric and its type parameter has *ServiceObject* as its bound type as follows:

abstract class Ambient<T **extends** ServiceObject>
{. . . }

Messages sent to an ambient are in general asynchronous, hence they are of the type *Message*. When such a message object is created, it has its identity, a lifetime, and behaves according to one of the specific subtypes of the type *Message*. For example, a transient message has a limited lifetime and a sustained message does not. Moreover, messages can be sent to message objects. For example, if a message is a two-way message, a message that refers to the future method may be sent to the two-way message object to obtain the result when it becomes available.

An ambient has a filter which is a predicate defined as:

abstract boolean filter(T x)

This method filters the relevant service objects that belong to the ambient. It is defined for a specific *Ambient* class, i.e, a class that is obtained from the class *Ambient* by instantiating it with a specific type of service objects. An ambient has a communication range which determines a collection of service objects that are in the ambient's range:

Set<T> communicationRange()

The reach of an ambient object is then the collection of all service objects of the given type that satisfy the filter predicate and are within the communication range of the ambient object:

Set<T> reach()

where for an ambient a and a service object x we have

$(x \in a.reach()) \Leftrightarrow (a.filter(x) \wedge (x \in a.communicationRange())$.

The class *Ambient* is equipped with a scheduler that selects the next message for execution according to some strategy. So the *Ambient* class looks like this:

```
abstract class Ambient<T extends ServiceObject> {
 abstract boolean filter(T x);
 Set<Message> messages();
 Set<T> communicationRange();
 Set<T> reach();
 invariant (∀ T x)
 ((x ∈ this.reach()) ⇔ (this.filter(x) ∧ (x ∈ this.communicationRange())))
}
```

An example of a specific ambient class is

```
class StockBroker extends ServiceObject {
 int quote(String stock);
 int responseTime();
 . . .
}

class StockBrokerAmbient extends Ambient<StockBroker> {
```

```
 String[] displayStocks(){. . .};
 requestQuote(String stock){. . .};
 boolean filter(StockBroker x)
 {return x.responseTime() ≤ 10; }
}

StockBrokerAmbient stockbrokers = new StockBrokerAmbient();
```

An example of an asynchronous message is:

```
Method requestQuote = getClass("StockBrokerAmbient").getMethod(
      "requestQuote", getClass("String"));
Message requestQuoteMsg = new MessageObject(requestQuote,
      stockBrokers,stock)
```

An alternative notation looks like this:

```
Message requestQuoteMsg = stockbrokers ⇐ requestQuote(stock)
```

An example of a two way message is:

```
TwoWayMessage requestQuoteMsg = ! new
 TwoWayMessageObject(requestQuote, stockBrokers,stock,20)
```

A suggestive notation for a two way message is:

```
TwoWayMessage requestQuoteMsg =
      stockbrokers <=> requestQuote(stock,20)
```

In order to access the result of a two way message, its future must be resolved, which requires testing the precondition of the future of the message.

Using a suggestive notation for a one-to-many message, we would have the following example:

```
Message requestQuoteMsg = stockbrokers <=>> requestQuote(stock)
```

In a more general concurrent setting a concurrent object is equipped with its own virtual machine:

```
class ConcurrentObject {
 private VirtualMachine VM();
}
```

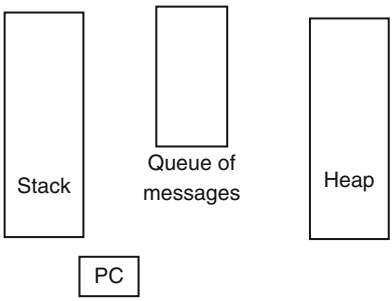


Fig. 5.5 A concurrent object

In the extreme case, all objects are concurrent objects, i.e., the class *ConcurrentObject* is identified with the class *Object*. A virtual machine is equipped with a stack, a heap, and a queue of messages as shown in Fig. 5.5. Further details are elaborated in Sect. 5.3.3. A concurrent object executes messages. If it is busy executing a message, a message is placed in the queue for subsequent execution. A service object is now defined as a concurrent object:

class ServiceObject **extends** ConcurrentObject { . . . }

We can now redefine an ambient in this new setting as a concurrent object which represents a dynamic collection of concurrent service objects:

```
class Ambient <T extends ServiceObject>
      extends ConcurrentObject {
 . . . }
```

Since an ambient is a concurrent object, it has its own virtual machine with a queue of messages sent to the ambient object and not serviced yet.

A mobile object is a concurrent object that is equipped with a location:

```
class MobileObject extends ConcurrentObject {
 Location loc ();
}
```

A region is an ambient that captures the notion of locality. It consists of all concurrent objects within the region as well as the service objects in that region, as illustrated in Fig. 5.6.

```
class Region <T> extends Ambient<T> {
 Set<ConcurrentObject> objects();
 boolean withinRegion(MobileObject x);
 invariant
 (∀ MobileObject x)(this.withinRegion(x) ⇒
      (x ∈ this.objects()))
}
```

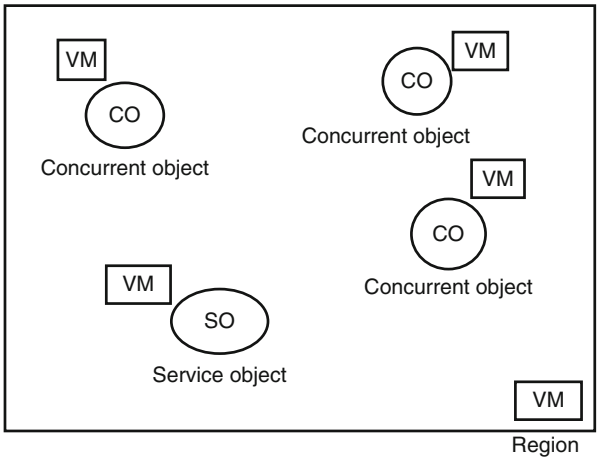


Fig. 5.6 Regions of concurrent and service objects

For example, if

class Server **extends** ServiceObject {. . .}

then *Region<Server>* would be an example of a region type. Since a region is a concurrent object, it is equipped with its own virtual machine. Also, since a region is an ambient, it receives messages that are queued in the message queue of the region's virtual machine to be serviced. Servicing a message sent to a region amounts to selecting a server object and sending the message to that server.

5.3.2 Reflection

Reflection in a language that supports messages as typed objects includes classes *Class*, *Field, Method*, and *Constructor*. In addition:

- Reflection includes the interface *Message* with it various subtypes.
- Reflection includes the interfaces *Assertion* and *Expression* with their various subtypes.

Reflection also includes the interface *State* whose implementation details are hidden. The hidden components of *State* include the states of the stack, the heap, the queue of messages, and the state of the program counter which determines the currently executed message.

A class in the source language has the following format:

```
class C extends D {
 constructor(C);
```

```
 methods(C);
 invariant(C);
}
```

Following the notation introduced in Chap. 4, the signatures of constructors and methods of a class C are specified below:

– *constructor(C)*: *C(E1,E2,. . .,En)* where *Ei* is a type
– *method(C)*: *B m(A1,A2,. . .,An)* where *B, Ai* are types.

An invariant is an expression of type *Assertion*. The core reflective class *Class* has the following abbreviated signature. A distinctive feature is an assertion representing a class invariant.

```
final class Class {
 String name();
 Method[] methods();
 Method getMethod(String name, Class[] arguments);
 . . .
 Assertion invariant();
}
```

Specification of a method in the surface language follows the following general format:

```
B m(A1 x1, A2 x2, . . ., An xn){
 preCondition(m);
 postCondition(m);
 modifyList(m);
 return expression(m);
}
```

where *B, A1, A2, . . ., An* are types. Preconditions and postconditions are assertions. The modify list specifies the objects that are mutated by the method. If this list is empty the method is a pure function with no effects on the receiver or side effects on other objects. A mutator message refers to a method whose modify list is not empty. The form of a modify list is:

modifyList(m): *C1 x1, C2 x2, .. . , Cn xn*

where *Ci* is a type and *xi* is an object name for $i=1, 2, \ldots, n$.

The effect of a mutator message is specified by assertions. The body of a method is an expression. Expressions include constants, variables, as well as the following expressions:

– !**new** C(e)—object creation where C is a class and e is an expression
– x.m(e)—synchronous message

– x <= $m(a1, a2, \ldots, an)$, x <=> $m(a1, a2, \ldots, an)$, x <=>> $m(a1, a2, \ldots, an)$, etc.—asynchronous messages

The reflective class *Method* is defined as follows. Its distinctive features are a precondition and a postcondition expressed as assertions. Their type is *Assertion*.

```
final class Method {
 String name();
 Class declaringClass();
 Assertion preCondition();
 Assertion postCondition();
 Class[] arguments();
 Class result();
 Expression body();
 Object eval(Object receiver, Object[] args);
}
```

The body of a method is an expression evaluated by the function *eval*. Just like *Assertion*, the type *Expression* belongs to the reflective core. The method *eval* evaluates the method body after binding of variables occurring in the expression representing the method body. The variables to be supplied to *eval* are the receiver and the arguments.

Availability of assertions in the classes *Method* and *Class* is a major distinction with respect to the current virtual machines such as JVM or CLR. This is at the same time a major difference with respect to the assertion languages such as JML or Spec#.

Assertions are built from logical constants and boolean synchronous messages representing terms of the assertion language. More complex assertions are built using standard logical connectives. The assertion language includes formulas of the typed first order predicate calculus. So the following expressions are in the assertion language, where *C* is a type, *x* is a variable, and e is an assertion:

– (∀ C x)(e)
– (∃ C x)(e)

The assertion language is enriched with features of more sophisticated logics. The interface *Assertion* is specified as follows:

```
interface Assertion {
 Set<Variable> boundVariables();
 Set<Variable> freeVariables();
 Formula formula();
 boolean bindVariables(Object[] vars);
 boolean eval();
}
```

Run-time representation of a variable as visible by reflection would look like this:

```
interface Variable<T> {
 String name();
 Method declaringMethod();
 StackValue<T> value();
}
```

So in general, an assertion will contain both bound variables and free variables. An invariant has only bound variables, hence it is an instance of the type *Sentence* specified below. A sentence is a closed formula, i.e., all its variables are bound (quantified).

```
interface Sentence extends Assertion {
 QuantifierType quantifier();
 Formula body();
 boolean eval();
 invariant (∀ Sentence s)(s.freeVariables().isEmpty())
}
```

Preconditions and postconditions may contain both free and bound variables. These assertions are instances of the type *Formula*:

```
interface Formula extends Assertion {. . .}
```

5.3.3 Virtual Machine

In this paradigm a virtual machine is equipped with a run-time stack, a heap, and a queue of messages waiting for execution, as specified in the class below:

```
final class VirtualMachine {
 int currentTime();

 private Stack<StackValue< T >> VMstack();
 private Heap<Object> VMheap();
 private Queue<Message> messages();

 enum Exception = (noException, preCondition, postCondition,
      invariant, expiredMessage);

 private boolean executeMessage(Message msg);
 private boolean setException(Exception e);
 Exception getException();
 . . .
}
```

The method *executeMessage* has the general decomposition given below. *eval* denotes a standard stack-oriented evaluation of an expression which we do not elaborate further. The classes *Expression* and *Assertion* are equipped with this function. The class *Assertion* is also equipped with a method *bindVariables* which is invoked before and after method execution. For a postcondition this method is invoked twice as binding of variables must be done before and after method execution. Invoking a method amounts to evaluation of its body which is an expression to which the function *eval* applies.

```
if msg.instanceOf(TransientMessage) ∧
 (msg.discoveryTime() - msg.timeStamp()) < currentTime()
then
 {setException(expiredMessage); return false};
if msg.instanceOf(TwoWayMessage) ∧
 (msg.replyInterval() - msg.timeStamp() < currentTime())
then
 {setException(expiredMessage); return false};
if msg.m.preCondition().bindVariables(msg.receiver(),msg.arguments()) ∧
 msg.m.postCondition().bindVariables(msg.receiver(),msg.arguments()) ∧
 msg.m.preCondition.eval()
then
 {Object result = msg.m.eval(msg.receiver(), msg.arguments());
 if msg.instanceOf(TwoWayMessage) then {msg.setFuture(result)};
else {setException(preCondition); return false};
if msg.m.postCondition().bindVariables() ∧
 msg.m.postCondition().eval()
then return true
else { setException(postCondition); return false}
```

Of course, the invariant must also hold before and after message execution which we do not show above.

In a concurrent paradigm described in Sect. 5.3.1, a concurrent object is equipped with its own virtual machine. A concurrent object executes messages that it receives by invoking the corresponding methods. In order to be able to do that, the heap of the object's virtual machine must contain reflective classes such as *Class*, *Method*, *Message* etc. These classes are stored on the heap of the object's virtual machine. The heap also holds the object state. Execution of a method is based on the object's stack according to the standard stack-oriented evaluation model.

A concurrent object gets activated by receiving a message. If a concurrent object is busy executing a method, the incoming message is queued in the message queue of the object's virtual machine. Messages in the queue will be subsequently picked for execution when an object is not busy executing a method. So at any point in time an object is either executing a single message or else it is inactive (i.e., its queue of messages is empty).

Note that in the above described concurrent object architecture, object identities must be global. Such a global object identifier would naturally be tied to a reference to the object state on the object's heap. Reflective class objects must be repeated on each individual object heap. This is a consequence of a typed reflective paradigm.

The computational values are either of simple types or else typed references to objects. The computational values conform to the following interface:

interface StackValue <T> {. . .}

Values of simple type occur on the run-time stack:

```
interface SimpleStackValue <T extends SimpleType>
          extends StackValue<T>
{T value(); . . .
}
```

Typed object references also occur on the run-time stack:

```
interface RefStackValue <T extends Object>
          extends StackValue<T>
{Ref<T> value(); . . .
}
```

At the level of the virtual machine typed references are explicit and conform to the following specification expressed in the object-oriented notation:

```
class Ref<T extends Object > {
 static Ref<T> new();
 boolean allocated();
 boolean null();
 T heap();
 boolean mutate(T state);
}
```

Execution of a message amounts to a sequence of steps of the virtual machine each corresponding to a particular operation. In the standard functional notation we have:

$heap : Ref < T > \rightarrow T$
$allocated : Ref < T > \rightarrow boolean$
$top : Stack < StackValue < T > > \rightarrow StackValue < T >$

As explained previously, *eval* denotes the standard stack-based evaluation of an expression *e* which we do not elaborate further. In accordance with the standard stack model (and the current virtual machines) we will assume that the result of evaluation of an expression *e* is placed on top of the virtual machine stack.

5.4 Bibliographic Remarks

The presented model and the associated notation for concurrent threads is based on Java [3, 4]. A general reference for concurrent programming in Java is [5]. Samples of concurrent programming in Java are based on examples due to Russell Gillen and David Ericson.

The presented view of concurrent, distributed and mobile objects appeared in [1] and [2]. The motivation comes from languages ABCL [9, 10] and AmbientTalk [8]. The core difference is that both of these two languages are untyped, whereas our approach here is based on a type system. A further distinction is that ABCL and AmbientTalk are object-based and our approach is class based. A related work is [7]. Unlike ABCL reflective capabilities, reflection in this paper is type-safe. A major distinction is the assertion language as a core feature of the approach presented in this chapter.

The rules of behavioral subtyping are a special case of the rules specified in [6].

5.5 Exercises

1. Specify a class *JointAccount* with synchronized methods *withdraw* and *deposit* so that the state of the account will not be incorrectly managed in a concurrent environment. Does a constructor require synchronization?
2. Specify a class *PrintQueue* that contains a field representing a queue of objects of type *PrintJob* and synchronized methods for adding a job to the print queue and removing a job from the print queue. Make use of methods *notifyAll* and *wait* assuming that many threads may be adding and removing jobs to the print queue.
3. Specify a class *PrintServer* that implements the interface *Runnable*. This class has a field of type *PrintQueue* representing a queue of print requests. The run method of the class *PrintServer* is a loop with no termination condition. In this loop a print job is removed from the queue and the method *printout* that does the actual printing is invoked.
4. Consider a method *initializeArray* that takes an array of objects as its argument and initializes all objects to **null**. Specify the postcondition of this method. Specify the body of this method using the Java synchronize statement in such a way that the postcondition will hold even if this method is executed in a concurrent environment.
5. Specify a class *Company*. Objects of this class contain two collections: a collection of employees and a collection of departments. Methods operating on the collection of employees may be invoked concurrently with the methods on the collection of departments with no synchronization required. However, methods operating on either collection separately must be synchronized. Specify this situation using the Java synchronize statement.

6. Assume that a method X has a synchronized method that invokes a synchronized method on object Y. At the same time assume that the object Y has synchronized method that invokes a synchronized method on the object X. Specify a class whose main method creates two threads in such a way that this creates a deadlock, i.e., a situation in which each thread is waiting for the other to complete and neither thread can run.
7. A virtual machine that supports multiple threads like JVM or CLR has one run-time stack per thread. Specify a class whose main method creates two threads that invoke the same method of that class that has at least one local variable. Show the outcome of a test execution.
8. Specify a class with static fields that creates multiple threads of execution. Are these static fields shared data of these multiple threads? Run an experiment that verifies your conclusion.
9. Specify a post office application where the class *PostOffice* is viewed as an ambient of concurrent objects. Specify representative assertions of this application.
10. Elaborate further the interface *Formula* based on the Java expressions. Consult Java Language Specification for different forms of expressions.
11. Specify classes *Bag* and *Sequence* choosing an appropriate representation of these collections. Specify class invariants, preconditions and postconditions that make *Sequence* a behavioral subtype of *Bag*.
12. Extend the rules of behavioral subtyping specified in Sect. 5.2.3 to capture the requirements that apply to *modify* statements.
13. Extend the rules of behavioral subtyping specified in Sect. 5.2.3 to capture the requirements that apply to exceptions thrown by methods.

References

1. S. Alagić, A. Yonezawa, Ambients of persistent concurrent objects, in *Proceedings of DBKDA 2011 (Advances in Databases, Knowledge, and Data Applications)*, IARIA 2011 (2011), pp. 155–161
2. S. Alagić, A. Anumula, A. Yonezawa, Verifiable constraints for ambients of persistent objects. Adv. Softw. **4**, 461–470 (2011)
3. K. Arnold, J. Gosling, D. Holmes, *The Java Programming Language*, 4th edn. (Addison-Wesley, Boston, 2005)
4. J. Gosling, B. Joy, G. Steel, G. Bracha, *The Java Language Specification*, 3rd edn. (Prentice Hall, New Jersey, 2005)
5. D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd edn. (Addison-Wesley, Boston, 1991)
6. B. Liskov, J.M. Wing, A behavioral notion of subtyping. ACM TOPLAS **16**, 1811–1841 (1994)
7. J. Schafer, A. Poetzsch-Heffter, JCoBox: generalizing active objects to concurrent components, in *Proceedings of ECOOP 2010*. Lecture Notes in Computer Science, vol. 6183 (Springer, New York, 2010), pp. 275–299

8. T. Van Cutsem, Ambient references: object designation in mobile ad hoc networks, Ph.D. dissertation, Vrije University Brussels, 2008
9. T. Watanabe, A. Yonezawa, Reflection in an object-oriented concurrent language, in *Proceedings of OOPSLA* (ACM Press, New York, 1988), pp. 306–315
10. A. Yonezawa, J.-P. Briot, E. Shibayama, Object-oriented concurrent programming in ABCL/1, in *Proceedings of OOPSLA* (ACM Press, New York, 1986), pp. 258–268

Chapter 6
Object Databases

Object-oriented database technology addresses the limitations of relational and extended relational technologies in managing complex objects that exhibit complex behavior. The relational technology has no good way of representing or managing such persistent objects. Object-oriented database technology is also addressing the problem of the mismatch between the dominating software technology, which is object-oriented, and the dominating database technology that is relational or object-relational at best. This specifically applies to the so called impedance mismatch between data and programming languages.

Queries are a key feature of any database technology. In Sect. 6.1 we discuss the basic features of object-oriented query languages. We start with the ODMG schema definition language ODL in Sect. 6.1.1. We then discuss in Sect. 6.1.2 the main features of OQL, the ODMG query language. Embedding of OQL in the Java binding of ODMG is then discussed in Sect. 6.1.3. This discussion of the problems of Java OQL is followed by a presentation of LINQ in Sect. 6.1.4 which resolves many problems of the Java binding of ODMG.

Section 6.2 is devoted to the most important object-oriented database technologies. In Sect. 6.2.1 we explain the core features of the ODMG technology which include database collection classes, interfacing databases from an object-oriented programming language, and the ODMG notion of a transaction. In Sect. 6.2.2 we revisit the typing issues for persistent collections. The core features of a specific recent technology Db4 Objects are presented in Sect. 6.2.3. Section 6.2.4 contains a sample application implemented using Db4 Objects.

S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0_6

6.1 Query Languages

6.1.1 Object Definition Language

In the ODMG (Object Data Management Group) approach there exists a declarative, object-oriented language ODL for specifying database schemas. ODL allows specification of types of database objects and collections of objects of those types. A class specification includes signatures (names and types) of attributes (fields), signatures of applicable methods, and relationships among database objects (inheritance and associations). ODL was meant to be independent of particular object-oriented programming languages. Queries are specified in a stand-alone object query language OQL, and transactions are written in languages such as Java, C++ and Smalltalk. The core idea is that there should be a unified type system across all of these languages, which is provably impossible. The reason is that the type systems of the language bindings are very different. Smalltalk does not have a type system to speak of, and even the Java and the C++ type systems are different, particularly so at the time the ODMG document was developed. At that time C++ had parametric types and Java did not. The C++ does not have the root object type like Java does, etc.

The core features of ODL and OQL will be described using the following sample of an ODMG schema. This schema contains types (classes) *Publication* and *Author*, and two classes derived from *Publication* by inheritance, *Book* and *Article* (Fig. 6.1).

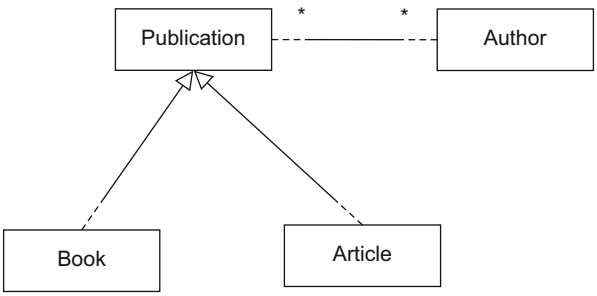


Fig. 6.1 Sample ODMG schema

The relationship between the types *Author* and *Publication* is M:N. That is, an author is associated with some finite number of publications, and a publication has a finite number of authors. These relationships requite parametric collection types as indicated in the specification of the relationship *publicationOf* in the class *Author* and the relationship *authoredBy* in the class *Publication*. But in fact, parametric types do not exist in the ODMG Object Model, contrary to the ODL notation in this schema.

Every relationship comes with specification of its inverse: *authoredBy* in the class *Publication* and *publicationOf* in the class *Author*. Classes in this schema come with specification of attributes and signatures of methods. Methods are meant to be implemented using ODMG interfaces (bindings) for mainstream object-oriented languages. Classes *Article* and *Book* inherit attributes and methods from the class *Publication*.

Each one of the classes in this schema contains specification of their extents. The extent of a class is the collection of all objects of that class. This notion is in the database tradition, relational in particular, where there is a single collection of all entities of a given type in a relational database. Queries are operating on class extents.

```
class Author (extent Authors) {
 attribute string name;
 relationship set<Publication> publicationOf
      inverse Publication::authoredBy;
 string getName();
}

class Publication (extent Publications) {
 attribute string title;
 attribute integer year;
 relationship list<Author> authoredBy
      inverse Author::publicationOf;
}

class Article extends Publication (extent Articles) {
 attribute string journal;
}

class Book extends Publication (extent Books) {
 attribute double price;
}
```

6.1.2 Object Query Language

OQL (ODMG Object Query Language) is a stand-alone, SQL-like object query language. Unlike SQL, OQL allows only specification of queries. Database actions such as insertion, update and deletion are meant to be specified in the ODMG interfaces (bindings) for object-oriented programming languages. OQL is based on *select-from-where* block from SQL as illustrated in the queries below.

The first query selects the titles of all publications authored after 1995 by more than one author. Note a reference to the relationship *authoredBy*.

```
select p.title
from Publications p
where p.year > 1995
and count(p.authoredBy) > 1
```

The next query selects titles of publications of a specific author and refers to the relationship *publicationOf*.

```
select p.title
from Authors a, a.publicationOf p
where a.name = "Tony Hoare"
```

As in SQL, the results of the above queries are in general bags, i.e., collections in which multiple copies of an element may occur. Eliminating duplicates and producing a set is performed by an explicit specification that the result must contain distinct elements.

```
select distinct a.name
from Authors a
```

As in SQL, it is possible to specify that the result of a query is an ordered collection. The query below will produce a result of the type *List* (of titles of publications) in the descending order of the year of publication.

```
select p.title
from Publications p
order by p.year desc
```

The two queries that follow illustrate how operations similar to relational joins are expressed using relationships. Relationships are meant to be implemented efficiently by appropriate access paths. Since the relationships come in pairs (direct and inverse), they are meant to be maintained automatically.

The query below produces a set of (distinct) names of authors. This is an example of a nested query. The outer query selects distinct names of authors of a particular publication selected in the inner query. The inner query selects publications of a particular author by relating authors and publications using the relationship *publicationOf* in the class *Author*.

```
select distinct a.name
from (select p
      from Authors a, a.publicationOf p
```

```
        where a.name = "Tony Hoare") p
where p.authoredBy a
```

Yet another example of a nested query selects titles of articles that appear in the year selected by the inner query.

```
select p.title
from Articles p,
      (select b.year
      from Books b
       where b.title= "Inferno") y
where p.year in y
```

OQL queries may be specified using quantifiers. The following query illustrates the usage of the existential quantifier. The predicate in the where clause of this query will be true if there exists a book of a particular author that satisfies the constraint on the price of the book. The query produces a collection of names of authors of those books.

```
select a.name
from Authors a
where exists b in Books:
       b.price < 200 and b in a.publicationOf
```

The following query selects names of authors such that all publications of those authors appear before or in the year 2000.

```
select a.name
from Authors a
where forAll p in a.publicationOf:
        p.year ≤ 2000
```

6.1.3 Java OQL

The ODMG Standard contains specification of the Java binding which is a collection of Java interfaces and classes to be used for interfacing with object-oriented databases from the Java programming language. Query capabilities are integrated into the Java binding using two different and related techniques. The first technique amounts to providing query methods for database collection types specified in the ODMG Standard.

ODMG collection types extend standard Java collection types by query methods. Database collections are meant to be equipped with special implementation suitable for large scale persistent collections. The implementation architecture of database collections is expected to have access paths such as indices allowing query optimization that produces efficient query execution algorithms.

The query method *selectElement* selects an element of a database collection that satisfies the predicate specified as the argument of this method. The method *select* produces an iterator over the collection that represents the result of this method. The method *query* produces the actual collection selected by the argument predicate. The query method *existsElement* tests whether a database collection contains an object that satisfies a given predicate.

```
public interface DCollection extends java.util.Collection {
 public Object selectElement(String predicate)
      throws QueryInvalidException;
 public java.util.Iterator select(String predicate)
      throws QueryInvalidException;
 public DCollection query(String predicate)
      throws QueryInvalidException;
 public DCollection existsElement(String predicate)
      throws QueryInvalidException;
}
```

The interface *DCollection* illustrates the problem of the so called impedance mismatch between data languages and programming languages. Java has no way of expressing predicates of the sort that query languages have. In order for this interface to compile by a Java compiler, predicates in the query methods are specified as strings. That prevents compile time syntactic and type checking of queries so that those errors will be detected only at run-time. This explains the exceptions thrown by these query methods. The unfortunate implication is that database transactions using the query methods may fail at run-time if the queries are not well formed syntactically or if they do not type check.

An example of using query methods that illustrates the above problem is given below. The schema relationships are shown in Fig. 6.2.

```
DCollection specialStudents;
specialStudents = Students.query(
      "exists s in this.takes: s.isTaughtBy.name="Turing");
```

In the above example, a string representing an OQL query expression is passed at run time to the method query. Because of this, syntactic and type errors will be detected only at run time and the query will fail if those errors occur.

Consider a simple schema which consists of a class *Employee* and a collection of objects of this class.

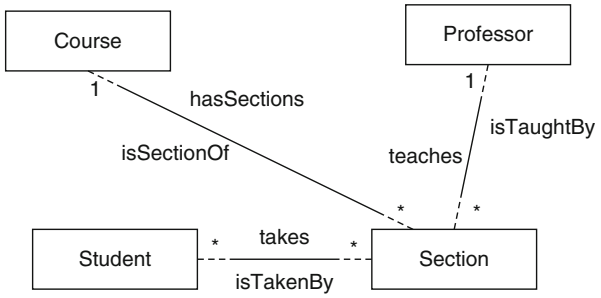


Fig. 6.2 Relationships in an ODMG sample schema

```
class Employee {
 public Employee(String name, short id, Date hiringDate, float salary){...}
 public String getName(){...}
 public short getId(){...}
 public Date getHiringDate(){...}
 public void setSalary(float s){...}
 public float getSalary(){...}
 private String name;
 private short id;
 private float salary;
 private Date hiringDate;
}

DCollection employees;
```

The state of an employee object is hidden in the above specification, as one would naturally expect to be the case in the object-oriented paradigm. The private fields *name*, *id*, *salary* and *hiringDate* can be accessed only by invoking accessor methods *getName*, *getId*, *getHiringDate*, *getSalary*. The object state can be modified only via methods, as illustrated by the method *setSalary*.

Consider now an example of usage of the method *query*.

```
DCollection employees, wellPaidEmployees;
...
wellPaidEmployees = employees.query("this.getSalary() ≥ 50000");
```

According to the Standard, *this* refers to the element of the queried collection, rather than to the receiver collection itself. This convention is actually different from the Java convention for using *this*. So the program generator must interpret the appropriate usage of the keyword *this*. However, the above remarks, although important, do not address the core problem: the Java type system cannot check

the embedded queries at compile-time. Query optimization cannot be performed at compile time either.

The other technique of incorporating OQL queries into the Java binding is by providing an interface *OQLQuery* and its implementing class. This is in fact a particular form of reflection. Queries are generally specifications in the source language, but now queries become objects, hence they are created dynamically. This approach has the same problems as query methods of *DCollection* class.

Query objects are created by supplying an OQL query as a string passed to the method *create* at run-time. This again prevents any static checking of the syntax and correct typing of the query string. The query is completed using the method *bind* that supplies specific arguments for the query. Errors in the above actions could be detected only when the method *execute* is invoked causing failure of Java transactions that create and execute queries if queries are not well-formed.

```
public interface OQLQuery {
 public void create(String query)
      throws QueryInvalidException;
 public void bind(Object parameter)
      throws QueryParameterCountInvalidException,
      QueryParameterTypeInvalidException;
 public Object execute()
      throws QueryException;
}
```

An ODMG example below illustrates how the OQLQuery class is used. *Impl* is a special ODMG class which allows construction of special ODMG objects such as query, database, transaction etc. A query object is first constructed by invoking the method *newOQLQuery* of the class *Impl*. Then the method *create* of the class *OQLQuery* is invoked to assign a query to the object *query*.

```
DCollection specialProfessors;
Double x;
OQLQuery query;
query=Impl.newOQLQuery();
query.create(
      "select p
      from p in Professors
      where p.salary > $1 and p in $2");
x=new Double(50000.00);
query.bind(x); query.bind(specialProfessors);
DBag selectedProfs = (DBag)query.execute();
```

A query is created by passing an OQL query as a string to the method *create*. The formal parameters are denoted by $1 and $2. Contrary to the appearance, no

syntactic or type checking can be done at compile time because the Java compiler sees the argument query just as a string. The method *bind* is invoked to supply the actual arguments of the query. There is no way of checking at compile time that the actual arguments are of the correct type. All of this creates obvious run time problems.

6.1.4 Language Integrated Queries (LINQ)

LINQ attempts to resolve the impedance mismatch between data and programming languages by incorporating object-oriented view of SQL queries into C#. LINQ is thus an integrated query and object-oriented language that overcomes many problems that the ODMG Java interface has.

LINQ operates on linearly ordered collections or sequences of elements. The interfaces *Enumerator* and *Enumerable* specify the required features that classes specifying enumerable collections must implement. An enumerator object is a cursor over an enumerable collection. It is equipped with a method *moveNext* that moves the cursor to the next element in the underlying sequence. The method *current* returns the current element determined by the cursor.

The two enumerator interfaces are:

System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>

An enumerator class will typically have the following specification:

```
class EnumeratorClass
 // implements IEnumerator or IEnumerator<T>
{
 public IteratorVariableType Current { get {...} }
 public bool moveNext() {...}
}
```

The two enumerable interfaces are:

System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>

An enumerable collection will typically have the following specification:

```
class EnumerableClass
 // implements IEnumerable or IEnumerable<T>
```

```
{
 public Enumerator getEnumerator() {...}
}
```

A query operates on an enumerable collection transforming it into another sequence representing the result of the query. LINQ queries have the form that is very similar to the SQL *select-from-where* block as illustrated below.

```
IEnumerable<String> query =
 from e in employees
 where e.salary() ≥ 100,000
 orderby e.salary()
 select e.name();
```

The above is just a specification of a query. A query is executed by a *foreach* statement over a query as follows:

```
foreach (String name in query) Console.WriteLine (name);
```

The above SQL-like queries are called comprehension queries. LINQ also has static methods of the class *Enumerable* that perform operations specified in the select, where and order by clauses in comprehension queries. These queries are called lambda queries because arguments are lambda expressions. Specifically, the *where* operator has a predicate specified as a function with the boolean result. The operator *select* projects elements of the input sequence into elements of the output sequence where this projection is specified as a function in lambda notation. Lambda expressions are here simply unnamed (anonymous) functions. The argument is bound to an element of the input sequence, and the result of the function is specified by an expression that shows how the argument is used to compute the result. The previous query has the following lambda query form.

```
IEnumerable<String> query = employees
 .Where (e ⇒ e.salary() ≥ 100,000)
 .OrderBy (e ⇒ e.salary())
 .Select (e ⇒ e.name());
```

As in OQL, queries can generate enumerable collections of objects rather than just enumerations of values. In the example below the type of objects in the sequence generated by a query is defined first, and then objects of that sequence are generated in the query.

```
class EmpSalary {
 String name;
 float salary;
 // . . .
}
```

```
IEnumerable<EmpSalary> =
 from e in employees
 select new EmpSalary
       { name = e.name();
       salary= e.salary() }
 where e.salary() ≥ 100,000;
```

As in OQL, queries can be nested. The query that follows produces a sequence of names of those employees whose salary is larger than the salary of their manager.

```
IEnumerable<String> =
 from e in employees
 where e.salary() ≥
         Max(from m in employees
         where e.manager()=m
         select m.salary())
 select e.name();
```

Interfacing with a relational database works as follows. Consider a simple specification of an SQL table *Employee*:

```
create table Employee
(
 ID int not null primary key,
 Name varchar(30)
)
```

LINQ makes use of optional predefined attributes that C# has to indicate that a class in fact corresponds to a database relation (table). Likewise, using predefined attributes, fields of a class can be specified as columns of a table.

```
[Table] public class Employee
{
 [Column(IsPrimaryKey=true)]
 public int ID;
 [Column]
 public String Name;
}
```

Access to a relational database is defined by providing a connection string that identifies the database. The class *DataContext* is equipped with a method *getTable* that delivers a table of the database with elements of a given type. There is only one

such table in a relational database. So *Table* is a parametric class that implements the interface *Queryable*. This interface extends the interface *Enumerable*, hence queries that operate on sequences can be specified on objects of type *Queryable* as in the example below. While objects of type *Enumerable* are internal sequences, objects of type *Queryable* are meant to be database sequences that allow queries to be optimized.

```
DataContext dataContext = new DataContext ("connection string");
Table<Employee> employees = dataContext.getTable <Employee>();
IQueryable<String> query =
 from c in employees
 where c.Name.First="A"
 orderby c.Name.Length
 select c.Name;

   foreach (String name in query) Console.WriteLine (name);
```

6.2 Object Databases

6.2.1 ODMG Java Binding

Databases typically consist of large scale persistent collections. These collections have special implementation. A database collection is typically equipped with efficient access paths, such as indices. This makes it possible to optimize queries for efficient execution while at the same time allowing efficient insertions, updates and deletions of persistent objects. This is why database collections have a special place in the ODMG approach. The ODMG database collection types are specified by the following interfaces.

```
public interface DCollection extends java.util.Collection {. . .}
public interface DSet extends DCollection, java.util.Set{. . .}
public interface DBag extends DCollection {. . .}
```

The above hierarchy of the ODMG collections is shown in Fig. 6.3.

The above interfaces are equipped with the usual methods for collections, sets and bags. A distinctive feature of database collections is that they have query methods. These methods are specified in the interfaces *DCollection* and inherited in the interfaces *Set* and *Bag*.

Accessing a database is performed through an interface *Database* whose simplified specification is given below.

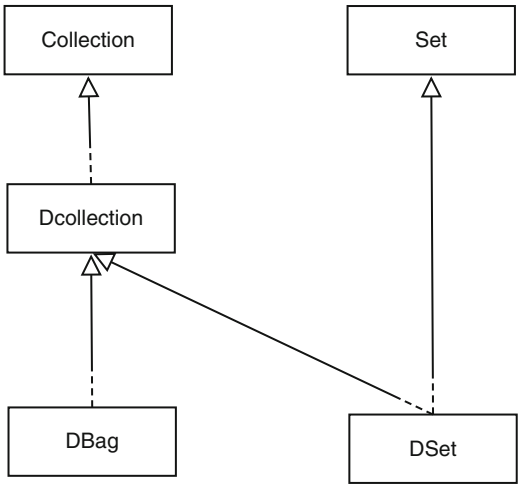


Fig. 6.3 Collections in the ODMG Java binding

```
public interface Database {
// access modes
 public void open(String name, int accessMode)
      throws ODMGException;
 public void close()
      throws ODMGException;
 public void bind(Object object, String name)
      throws ObjectNameNotUniqueException;
 public Object lookup(String name)
      throws ObjectNameNotFoundException;
// other methods
}
```

An object is promoted to persistence by invoking the method *bind* that makes the object supplied as the first argument persistent. This method also binds the name supplied as the second argument to the newly established root of persistence. This is the same idea that we discussed in the section on orthogonal persistence. The type signature of the method *bind* seems to suggest that the ODMG model of persistence is orthogonal because of the argument of type *Object*. But in fact, the ODMG model is based on persistence capable classes, like database collection types, hence the model is not orthogonal. On the other hand, the model supports transitivity, i.e., a complex object of any type will be correctly stored with all of its direct and indirect components. The model also satisfies the transparency requirement, i.e., the underlying persistence supporting architecture is completely hidden from the users.

The method *bind* establishes a root of persistence. An object is thus persistent if it is a root of persistence, or a direct or indirect component of a root of persistence.

The method *lookup* retrieves the root of persistence with the given name. The result is necessarily of type *Object*, hence a type cast (a dynamic type check) is required in order to perform specific actions on the retrieved object. This is a situation where the typing notions such as subtype and parametric polymorphism do not help. A dynamic check is necessary to assure that the type of the object retrieved from the database is indeed the type that the user program specifies (expects).

In the illustration given below the type of the resulting objects of a query is first specified by providing a suitable constructor and hiding the object state. A bag object is then created to hold the result of the query method. An *OQLQuery* object is then created. In the string object passed to the constructor of the query object the formal parameters of a query are denoted as $i, where $i is an integer. A specific query is formed at run-time by passing specific argument objects for these formal parameters using the method *bind*. A query is then executed, and its result is type cast to *DBag* in order to make the assignment of the resulting collection possible. The class *Employee* was specified in Sect. 6.1.2.

```
public class WellPaid {
  public WellPaid(String name, float salary){...}
  public String getName(){...}
  public float getSalary(){...}
  private String name;
  private float salary;
};
DBag selectedEmployees = new DBag();
// open the employee database
OQLQuery aQuery = Impl.newOQLQuery(
      "select wellPaid(e.getName(), e.getSalary())
      from e in employees
      where e.getSalary() ≥ $1
      and e.getHiringDate().after($2)");
aQuery.bind(new Float(50000));
aQuery.bind(NewYears);
selectedEmployees = (DBag)query.execute();
```

The above query expression would not type check in the type system of the ODMG Java binding because the type of elements of the collection *employees* is *Object*. This means that a type cast *(Employee)e* should be used in OQL queries in order for them to type check in the type system of the Java binding. This has two implications: the queries have a form which is not natural for the users, and dynamic type checks are required.

When the employee database is opened in the above example, its name space is opened. Of course, the identifiers *Employee* and *employees* should be defined in this scope. In spite of that checking this simple scoping rule is impossible at compile

time. All the compiler can check is that the argument passed to query constructor is a valid Java string.

An obvious implementation technique of the query class amounts to storing a query object as a string in the file system. Executing the *bind* method for the actual query arguments is then implemented as updating (in fact, reconstructing) the string representing the query. The *execute* method is implemented as a query interpreter, where a query is provided as a string.

In this interpretive technique parsing a query is then carried out every time a query is executed. All the errors in the string representation of the query are rediscovered every time the query is executed. This in particular includes scoping and typing errors. Furthermore, expensive query optimization techniques are also carried out every time a query is executed.

Consider now an example in which a database object is created and a database with a given name is opened. A query object is created; this object is then made persistent by invoking the *bind* method, and finally the database is closed.

```
Database d = Database.open("employeeDatabase", openReadWrite);
OQLQuery aQuery = Impl.newOQLQuery("
      select wellPaid(e.getName(),e.getSalary())
      from e in employees
      where e.getSalary() ≥ $1
      and e.getHiringDate().after($2)");
d.bind(aQuery,"sampleQuery");
d.close();
```

In the next example the same database is opened, the query object is looked up, and supplied with specific arguments using the method *bind*, the query is then executed, and the result made persistent.

```
Database d = Database.open("employeeDatabase", openReadWrite);
OQLQuery aQuery;
aQuery = (OQLQuery) d.lookup("sampleQuery");
aQuery.bind(new Float(50000));
aQuery.bind(NewYears);
DBag selectedEmployees = (DBag)query.execute();
d.bind(selectedEmployees, "wellPaidEmployees");
d.close();
```

Complex database actions are specified as transactions. A transaction is an object-oriented program that has the properties specified in the interface *Transaction*. In the ODMG object model a transaction is an object.

```
public interface Transaction {
 public void begin();
```

```
 public void commit();
 public void abort();
 public void checkpoint();
// other methods
}
```

ODMG transactions are meant to satisfy the ACID properties. *A* stands for atomicity. This means that a transaction is executed in its entirety and then it has a persistent effect on the database. Otherwise, a partial execution will not have any impact on the database, as if it never happened.

C stands for consistency. A transaction is assumed to start in a consistent database state, and when it terminates, the database must also be in a consistent state. The problem with this requirement is that a consistent database state is a state that satisfies the integrity constraints specified explicitly in the database schema. ODMG has no such general capabilities. Even the basic constraints such as keys, foreign keys and relationships cannot be specified in Java classes because Java itself would have no such capabilities.

I stands for isolation. This property guarantees that in a concurrent execution of multiple transactions, each transaction is isolated from the impact of other concurrent transactions.

D stands for durability. This means that once a transaction performs a successful commit action, its effects on the database will persist.

The method *abort* executed in a transaction will roll back all the changes the transaction performed on persistent objects and will bring the underlying database to the state prior to the transaction start.

Samples of ODMG style transactions given below follow those of O2, a legacy ODMG compliant database management system. In the first example, a binding for a database collection named "People" is introduced in the database with a given name. When the transaction *Tx* commits, this collection will persist.

```
Database db = new Database("server parameters");
 db.connect();
 db.open("database name");
 Transaction Tx = new Transaction();
 Tx.begin();
      DCollection people = new DCollection();
      Database.bind(people, "People");
 Tx.commit();
 db.close();
 db.disconnect();
```

In the next example the database collection named "People" will be looked up first. A new person object named "John" is created and added to the collection

"People". When the transaction *Tx* commits, the object named "John" will persist in the collection "People".

```
Transaction Tx = new Transaction();
Tx.begin();
 DCollection people =
      (DCollection) Database.lookup("People");
 Person p= new Person("John", 35);
 people.add(p);
Tx.commit();
```

In the transaction that follows two person objects named "John" and "Mary" are created and assigned to each other as spouses. Then the database collection named "People" is established and the objects named "John" and "Mary" are added to this collection. When the transaction *Tx* commits, these two person objects will persist in the collection "People", and their relationship will be correctly maintained.

```
Transaction Tx = new Transaction();
Tx.begin();
 Person john = new Person("John", 30);
 Person mary = new Person("Mary", 28);
 john.assignSpouse(mary);
 mary.assignSpouse(john);
DCollection people = new DCollection();
try {Database.bind(people, "People");}
 catch (DbException e) { handle exception };
 people.add(john);
 people.add(mary);
Tx.commit();
```

6.2.2 Parametric Persistent Collections

In this section we revisit the problems of parametric types in Java in the database context. The problems are illustrated by an experiment in which the Java compiler is used with Versant's FastObjects, a legacy object-oriented database management system that conforms to the ODMG Java binding.

In the Java program *BindTransaction* given below, a collection of employees is made persistent by associating it with the name *"employees"* in the database. This is naturally performed in a transaction to guarantee a correct outcome. The hidden problem is that FastObjects does not know that the run-time type of *employees* is wrong and as such it will be recorded in the database.

```
public class BindTransaction {
 private Database database;
 public static void main(String[] args) {
      String databaseName = args[0];
      BindTransaction transaction =
            new BindTransaction();
      transaction.initialize(databaseName);
      transaction.bind();
 }
 public void bind() {
      Bag<Employee> employees = new Bag<Employee>();
      Transaction txn = new Transaction();
      txn.begin();
      try {
      database.bind(employees, "employees");
      } catch (Exception e) {
      // ObjectNameNotUniqueException
      txn.abort();
      e.printStackTrace();
      }
      for(int i=0; i < 5; i++)
      employees.add(new Employee());
      txn.commit();
      }
// initialize, etc.
}
```

Consider the class *ViewDepartments* that has been previously compiled. There is nothing wrong in this class. No exception is expected and hence no exception handling is provided.

```
public class ViewDepartments {
 public static void displayDepartments(
 Bag<Department> departments) {
 for (int i = 1; departments.hasElements(); i++)
 System.out.println("Department " + i + ": " + departments.remove().name());
 // Unexpected ClassCastException?
 }
}
```

In the *LookupTransaction* given below, the database collection *employees* is looked up and type cast wrongly to a bag of departments. The compiler issues a warning here, although there is no static type error because the result type of *lookup*

is *Object*. The real problem is that this wrong type cast succeeds at run time with no class cast exception. The program fails unexpectedly in a class *ViewDepartments*

```
public class LookupTransaction {
 private Database database;
 public static void main(String[] args) {
 String databaseName = args[0];
 LookupTransaction transaction =
      new LookupTransaction();
 transaction.initialize(databaseName);
 transaction.lookup();
 }
 public void lookup() {
 Transaction txn = new Transaction();
 txn.begin();
 Bag<Department> departments = null;
 try {
      departments =
      (Bag<Department>) database.lookup("employees");
      // Cast should fail but does not
      } catch (ClassCastException e) {
      // No ClassCastException
      e.printStackTrace();
      } catch (ODMGException e) {
      txn.abort();
      e.printStackTrace();
      }
 ViewDepartments.displayDepartments(departments)
 txn.commit();
 }
 // initialize, etc.
}
```

Of course, discovering the specifics of types would be performed by reflection. But reflection is particularly impaired by type erasure, as we already showed in the chapter on type systems.

6.2.3 Db4 Objects

Db4 Objects is an object-oriented database management system representing the current state of the commercial products used in a variety of applications. Unlike ODMG, Db4 does not have an explicit notion of a database schema or a transaction

class. An object-oriented database is viewed through the methods of the interface *IObjectContainer*. This interface is equipped with the methods *open*, *close*, *commit* and *abort*, as in the ODMG interface *Database*. Associating an object of the type *IObjectContainer* with a database and performing actions on the database has the following form. The code samples in this section follow the C# interface of Db4 Objects.

```
IObjectContainer db = Db4oFactory.OpenFile(FileName);
try {
 access db4o
 }
finally
 { db.close(); }
```

Database actions expressible by the methods of the interface *IObjectContainer* will be illustrated by the following sample class. Fields *name* and *points* are private according to the C# rules. *Name* and *Points* are public properties, i.e., pairs of methods *get* and *set*, with the corresponding fields specified as private components of the object state.

```
public class Pilot
{String name;
 int points;

 public Pilot(String pName, int pPoints)
 { name = pName; points = pPoints; }

 public String Name {
 { get {return name; }
 { void set {name = value;}
 }
 public int Points {
 { get {return points; }
 {void set{points = value;}
 }

 public void addPoints(int Ppoints)
 { points += Ppoints; }
}
```

An object is created and then stored in the database by invoking the method *store* of the interface *IObjectContainer*.

```
Pilot pilot = new Pilot("Mark Sellinger", 100);
db.store(pilot);
```

Db4 has several query languages that allow retrieval of persistent objects. The simplest one is Query by Example. Its usage is illustrated below. A prototype object is created and passed as the argument of the method *get* of the interface *IObjectContainer*. The method *get* returns a collection of objects that match the prototype object specification. Default values (zero for numeric types) are used to indicate that any value qualifies. The query given below will return all pilot objects with the given name.

```
Pilot protoObj = new Pilot("Mark Sellinger", 0);
IObjectSet result = db.get(protoObj);
```

The next query returns all pilot objects that have the specified number of points.

```
Pilot protoObj = new Pilot(null, 100);
IObjectSet result = db.get(protoObj);
```

In the query below the prototype object contains default values of the relevant fields, hence all pilot objects will qualify.

```
Pilot protoObj = new Pilot(null, 0);
IObjectSet result = db.get(protoObj);
```

Updating an object is performed by the method *set* of the interface *IObjectContainer*. In order to establish the identity of a persistent object, the object must be first retrieved by the method *get*. In the example below a collection of pilots is selected. The first such object in the returned collection is updated and then the database update method *set* is invoked. The actual database update will be committed when the transaction that contains the code below performs that action.

```
IObjectSet result = db.get(new Pilot("Mark Sellinger", 0));
Pilot found = (Pilot) result.next();
found.addPoints(50);
db.set(found);
```

Deleting an object follows the same pattern. An object is first located by executing a query and then the method *delete* of the interface *IObjectContainer* is invoked.

```
IObjectSet result = db.get(new Pilot("Mark Sellinger", 0));
Pilot found = (Pilot)result.next();
db.delete(found);
```

Db4 has a more sophisticated query language that allows more complex selection (filtering) predicates. Db4 has a parametric type *Predicate<T>* whose method

match is overridden in order to specify a particular predicate bound to the actual type parameter substituted for the formal type parameter T. This is illustrated by the query below in which *Predicate<T>* is instantiated to *Predicate<Pilot>*. The type signature of the actual predicate *match* becomes *boolean match(Pilot x)*. The body of the method *match* now specifies the filtering condition. The result of the *IObjectContainer* method *query* is now a list of persistent pilot objects.

```
List <Pilot> result = db.query(new Predicate<Pilot>() {
 public boolean match(Pilot pilot) {
 return (pilot.Points ≥ 99
          ∧ pilot.Points ≤ 199)
      ∨ pilot.Name.equals("Mark Sellinger");
 }
});
```

Db4 supports transitive persistence. An example of a complex object type is *Aircraft* that has a pilot object as its component.

```
public class Aircraft
{String model;
 Pilot pilot;

 public Aircraft(String aModel)
 { model = aModel; pilot = null; }

 public Pilot Pilot {
      get { return pilot;}
      set { pilot = value;}
 }
public String Model
{get { return model; }
 }
}
```

Now we can create an aircraft object and a pilot object and update the aircraft object by assigning to it the pilot object. Database *set* action of the interface *IObjectContainer* is then executed and the update is committed to the database.

```
Pilot pilot = new Pilot("Mark Sellinger", 99);
Aircraft plane = new Aircraft("Boeing 777");
plane.Pilot = pilot;
db.set(plane);
db.commit();
```

Although Db4 does not have an interface *Transaction*, transaction support is an important component of Db4. The methods *commit* and *rollBack* perform committing actions on persistent objects and rolling back the changes performed by a transaction. *commit* is illustrated by the code above and rolling back by the example that follows.

```
Pilot pilot = new Pilot("Mark Sellinger", 100);
Aircraft plane = new Aircraft("Boeing 777");
plane.Pilot = pilot;
db.set(plane);
db.rollBack();
```

Db4 transactions are meant to satisfy the ACID properties. However, like other object database technologies, Db4 has no way of specifying general integrity constraints and hence cannot truly support the C component of ACID transactions.

6.2.4 Database Application

In this section we present a more elaborate database application and show database actions specified in Db4 Objects. This application contains a hierarchy of aircraft types. The two immediate subtypes of the *Aircraft* type are *Airplane* and *Helicopter*. The *Airplane* type has two subtypes: *CargoPlane* and *PassengerJet* (Fig. 6.4). The notation in this section is from the Java interface of Db4.

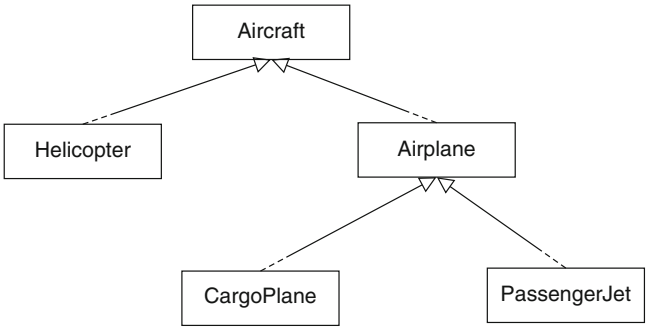


Fig. 6.4 Aircraft hierarchy

In the code below we show only some sample fields of these classes that will appear in queries.

```
public abstract class Aircraft {
 protected String aircraftId;
```

```
 // constructor and methods
}

public class Airplane extends Aircraft {
 protected List<AirplaneEngine> engines;
 // constructor and methods
}


public class CargoPlane extends Airplane {
 private int storageCapacity;
 // constructor and methods
}


public class PassengerJet extends Airplane {
 private int maxPassengers;
 // constructor and methods
}
```

The method *saveObject* given below encapsulates actions of storing the argument object in the database, committing the changes, and handling the relevant exceptions. *db* is of the type *IObjectContainer* as in the previous examples.

```
private void saveObject(Object object) {
 try {db.store(object);
      db.commit();
      }
 catch (Db4oIOException | DatabaseClosedException |
      DatabaseReadOnlyException e)
      {exception handling }
 }
```

A specific query type given below makes use of the method *queryByExample* to return *Airplane* objects with a given identifier (probably just one).

```
public ObjectSet<Object> findAirplaneById(String id)
{
 Airplane plane = new Airplane(id, null);
 try { return db.queryByExample(plane);
      } catch (Db4oIOException | DatabaseClosedException e)
      { exception handling;}
 return null;
}
```

A complex native query given below produces a list of aircraft objects. The matching predicate of the query specifies that only airplane objects are selected and only those with the specified size of engines.

```
public List<Aircraft> selectAirplanes() {
 List<Aircraft> matches =
 db.query(new Predicate<Aircraft>() {
 public boolean match(Aircraft aircraft) {
      if (aircraft instanceof Airplane) {
      Airplane airplane = (Airplane) aircraft;}
      if (airplane.getEngines().size() ≥ 500)
      return true;
      else return false;
 });
 return matches;
}
```

The generality of native Db4 queries is appealing but it comes with a nontrivial drawback. The predicate *match* could be specified using arbitrary C# code which makes query optimization practically intractable.

Storing an object with specific properties such as a user key is performed by first querying to find out whether such an object already exists in the database. If not, the object is stored using the method *saveObject*.

In order to update a database object, the database is first queried to locate the object. If a database object is located, it is updated. The updated object is stored in the database invoking the method *saveObject*.

In order to delete a database object, the database is first queried to locate the object to be deleted. The located object is deleted by invoking the method *delete* of the interface *IObjectContainer*. Deleting an object may be performed in such a way that all components of the object are also deleted. This is accomplished by enabling cascading on deletion.

Examples in this and the previous section reveal one more aspect of the mismatch between data languages and object-oriented programming languages. In object-oriented programming languages the object state should be preferably declared as private so that it can be accessed only by public methods, like in properties of C#. In database systems the components of the object state could be still accessed directly for query purposes. There is a practical reason for this and it has to do with query optimization. If queries make use of methods, such as the C# method *get* in a property, query optimization becomes practically impossible. The method *get* could contain complex code, and so can the method *match* of Db4. There is no technology for optimizing a query which contains arbitrarily complex code. Query optimization techniques are based on the relational model which is so much simpler.

6.3 Database Constraints

Complete lack of declarative, logic-based constraint capabilities in mainstream object-oriented languages has major implications on incorporating queries and transactions into those languages. If an object-oriented language had a suitably defined constraint language, such a language would naturally include (or be easily extended with) full-fledged query capabilities.

In traditional database technologies queries are functions expressed in a high-level declarative specification language. In the ODMG paradigm queries are naturally objects. Correct functioning of this reflective feature depends upon the availability of constraints. As in ODMG, a query in the paradigm presented in this section is an instance of a query class given below. A major distinction is in constraints that are not available in the ODMG Object Model.

The invariant of the class *Query* (a constraint that applies to all visible object states outside of method execution) specifies the properties of the query result, and it thus plays the role of a qualification (filtering) expression in traditional queries. The constraints are expressed in Horn clause logic. All variables are assumed to be universally quantified and the only logical connectives are implication and conjunction.

The predefined class *Query* is both abstract and parametric. The method *qualification* is abstract. It is thus required to be implemented in a specific, user query class, by providing the actual qualification expression. An actual query method is *selectFrom*. It is applied to the argument collection of this method and the qualification expression is used for selecting the relevant elements. This condition is expressed by the invariant of this class.

Bounded parametric polymorphism allows specification of queries that involve projection and/or ordering. The query class given below has two type parameters. *T* stands for the element type of the queried collection, and *P* for the element type of the collection that represents the result of the query. *P* is a subset of features of *T* (projection), hence the type constraint *T* **extends** *P*. The other type constraint *P* **extends** *Comparable* guarantees that the selected elements are equipped with the ordering methods.

```
public abstract class Query< P extends Comparable,
                             T extends P>;
{ public Query<P,T>();
 public abstract boolean qualification(T x);
 public final Collection<T>
                             selectFrom(Collection<T> S);
 public final Collection<P>
                             selectAndProject(Collection<T> S);
 public final OrderedCollection<P>
                             selectProjectAndOrder(Collection<T> S);
 invariant
```

```
 (∀ Query<P,T> this) (∀ Collection<T> S) (∀ T x)
 (this.selectFrom(S).contains(x) ⇐
       S.contains(x) ∧ this.qualification(x))
 // other clauses
}
```

The fact that all user-defined query classes must extend the abstract *Query* class is an explicit indication to the compiler that the assertions of these classes are subject to optimization. This is why the query methods *selectFrom*, *selectAndProject* and *selectProjectAndOrder* are final. Their implementation is entirely system-oriented and contains query optimization in particular.

The above described model for object-oriented queries comes with some major advantages in comparison with other competitive approaches. The approach is truly object-oriented, and it allows static parsing, type checking and query optimization. By way of comparison, the ODMG model, although object-oriented, does not have any of the remaining advantages of the query model presented in this section.

The database class given below offers features similar to the ODMG class *Database* and the PJama interface *PJStore* with a fundamental difference: *constraints*. The method *bind* of this class binds a name (the second argument) to an object of any type (the first argument of type *Object*). As *Class* extends *Object*, a database contains bindings for both classes and objects. The method *lookUp* returns an object (a root of persistence) of a database bound to a name. Note that the precondition and the postcondition of the method *bind* specify the semantic relationship with the method *lookUp*. A database naturally contains additional constraints.

```
public class Database
{ public Database(String name);
 public final boolean isOpen();
 public final void open()
       ensures this.isOpen();
 public final void close()
       requires this.isOpen();
 public final Object lookUp(String name)
       requires this.isOpen();
 public final boolean bind(Object x, String name)
       requires this.isOpen(),
       requires this.lookUp(name)= null,
       ensures this.lookUp(name).equals(x);
// invariant etc.
}
```

In traditional database technologies transactions are programs. An object-oriented model of transactions is naturally reflective: transactions are objects, instances of a class *Transaction*. This view is extended in the paradigm presented

in this section by an interplay of bounded parametric polymorphism for binding a transaction to a database schema and the enforcement of database integrity constraints.

A transaction is required to maintain the integrity constraints of its schema. The precondition of the method *start* requires that the integrity constraints of the transaction's schema are satisfied before the transaction begins. The postconditions of the methods *commit* and *abort* require that the integrity constraints of the transaction's schema are satisfied.

```
public abstract class Transaction <T extends Database>
{ protected T schema;
 public Transaction<T>(T schema);
 public final void start()
       requires schema.invariant;
 public final void commit()
       ensures schema.invariant;
 public final void abort()
       ensures schema.invariant;
 public abstract void execute()
       requires schema.invariant,
       ensures schema.invariant;
}
```

The class *Transaction* is abstract because its method *execute* is. Specification of the actual transaction code which the *execute* method contains is thus deferred to subclasses of the *Transaction* class. However, the precondition and the postcondition of the *execute* method are specified. They require that *execute* acts as a unit of integrity with respect to the database constraints.

An example of a corporate database is given below. The example illustrates the primary features of the model. Usage of parametric polymorphism is shown in the collections of departments and employees, as well as in a sample transaction. A variety of constraints are given, including key and referential integrity constraints, in the form of invariants of the *Corporation* class, as well as preconditions and postconditions of several methods.

The class *Corporation* includes inner interfaces *Employee* and *Department* defined as follows.

```
public interface Employee {
 String name();
 String ssn();
 Float salary();
 Department department();
 void assignDepartment(Department d)
```

```
      ensures this.department().equals(d);
 invariant
 (∀ Employee X,Y)
 (X.equals(Y) ⇐ X.ssn().equals(Y.ssn());
}

public interface Department {
 Integer deptNum();
 Collection<Employee> employees();
 Float allocatedPayroll();
 void addEmployee (Employee e)
      ensures this.employees().contains(e);
 void removeEmployee (Employee e)
      requires this.employees().contains(e);
 invariant
 (∀ Department X,Y)
 (X.equals(Y) ⇐ X.deptNum().equals(Y.deptNum());
}
```

The class *Corporation* given below contains two database collections *dbEmployees* and *dbDepartments* and a sample transaction *HireTrans*.

```
public class Corporation extends Database {
 public interface Employee
{ . . .}
 public interface Department
{. . . }
 public class EmployeeCollection
      implements Collection<Employee> {
 public void add (Employee e);
 }
 public class DepartmentCollection
      implements Collection<Department> {
 public void add (Department d);
 }
 EmployeeCollection dbEmployees;
 DepartmentCollection dbDepartments;
invariant
(∀ Employee W) (∀ Department Y)
(dbEmployees.contains(W) ⇐
      dbDepartments.contains(Y) ∧
      Y.employees().contains(W),
 dbDepartments.contains(Y) ⇐
```

```
        dbEmployees.contains(W) ∧
        W.department().equals(Y));
}
public class HireTrans extends
        Transaction<Corporation>
{. . . }
}
```

A sample transaction *HireTrans* that satisfies the above database integrity constraints is specified as follows:

```
public class HireTrans extends Transaction<Corporation> {
 public HireTrans (Corporation corp, Employee emp, Department dept)
 {. . . .}
public void execute()
 requires corp.dbDepartments.contains (dept)
 { emp.assignDepartment (dept);
 dept.addEmployee (emp);
 corp.dbEmployees.add (emp);
 }
}
```

6.4 Bibliographic Remarks

The first defining document on object-oriented databases was Object-Oriented Database Systems Manifesto [7]. Subsequently, the Object Oriented Database Management Group (ODMG) took the task of developing the industrial standard for object-oriented databases. The final version of this document is [9]. A critical view of the ODMG Standard was presented in [2]. Perhaps the most developed system that accepted this standard was O2 [8].

Object Definition Language (ODL) and Object Query Language (OQL developed for O2) are specified in the ODMG Standard [9]. Some examples of Java OQL queries are from [6]. The problems in type checking these languages are formally elaborated in [3]. Presentation of the main features of these two languages is based on the examples from the lecture notes on object databases [11]. Java OQL is defined in the ODMG Standard [9]. Language Integrated Query (LINQ) is specified at [12]. Presentation of its basic features follows [1].

Database and transaction interfaces are specified in the ODMG Standard. and so are the ODMG collection interfaces [9]. Examples of ODMG style transactions are based on O2 [8]. The example that illustrates problems with persistent parametric collections in Java is from [5]. The two most important recent industrial database systems are Db4 Objects [10] and Objectivity [13]. The presentation of the basic

Db4 features is based on the Db4 tutorial [10]. Examples of Db4 actions in the sample database application are due to Russell Gillen. The presented model of database constraints appeared in [4].

6.5 Exercises

1. Specify the sample ODMG *Publication* schema in Fig. 6.1 using C# and LINQ.
2. LINQ is not just a query language. It allows insertion, updating and deletion with respect to a data context. Load sample data for the *Publication* schema and explore updates and deletions as defined in LINQ.
3. Specify LINQ queries that correspond to the OQL queries given in Sect. 6.1.2 as:
   - Comprehension queries
   - Lambda queries.
4. LINQ has the ability to specify associations. Specify the ODMG schema in Fig. 6.1 by making use of LINQ associations.
5. Load sample data for the above schema, specify representative comprehension and lambda queries, as well as representative updates and deletions.
6. Implement *BindTransaction* given in Sect. 6.2.2 in Java using Java serialization mechanism as follows:
   - A class *LoadData* will represent a transaction that specifies types *Employee* and *Department*. This class loads *Employee* and *Department* data into different files using Java serialization.
   - A class *ViewDepartment* representing a second transaction will read the data from the departments file and display the data read performing the correct type cast.
   - A class *Lookup* will represent a third transaction that reads data from the employees files and performs a wrong type cast of a collection of *Employee* objects read into a collection of *Department* objects.

   Show that the same problems occur as in the database implementation given in Sect. 6.2.2.
7. Specify an airport schema in LINQ. An airport has a list of runways, a list of traffic patterns (incoming and departing aircraft), a set of ground structures including the control tower, and a list of runways. Make use of the LINQ association capabilities.
8. Load data for the above schema, specify a collection of sample queries in the comprehension and lambda format, and typical transactions performing updates and deletions. LINQ has a limited transaction-like capabilities that allow committing changes with respect to a data context.
9. Implement the airport schema in Db4.
10. Implement the ODMG class *Database* using the support provided by Db4.

11. Produce a representative collection of complex native queries on the airport database.
12. Implement the ODMG class *Query* using support provided by Db4 and its native queries.
13. Write Db4 transactions that perform insertion, update and deletion of objects of the airport database. Use cascading when appropriate in these operations.
14. Implement the ODMG class *Transaction* using the transaction support provided by Db4.

References

1. J. Alabahari, B. Albahari, *C# 5.0 in a Nutshell* (O'Reilly, Beijing, 2012)
2. S. Alagić, The ODMG object model: does it make sense?, in *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (ACM, New York, 1997), pp. 253–270
3. S. Alagić, Type checking OQL queries in the ODMG type systems. ACM Trans. Database Syst. **24**, 3 (1999)
4. S. Alagić, J. Logan, Consistency of Java transactions, in *Proceedings of DBPL 2003 (Database Programming Languages)*. Lecture Notes in Computer Science, vol. 2921 (Springer, Berlin, Heidelberg, 2003), pp. 71–89
5. S. Alagić, M. Royer, Genericity in Java: persistent and database systems implications. VLDB J. **17**(4), 847–878 (2007)
6. S. Alagić, J. Solorzano, Java and OQL: a reflective solution for the impedance mismatch. L'objet **6**, 275–296 (2000)
7. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, S. Zdonik, The object-oriented database systems manifesto, in *Proceedings of the First Conference on Object-Oriented and Deductive Databases*, Kyoto, 1989, pp. 223–240
8. F. Bancilhon, C. Delobel, P. Kanellakis, *Building an Object Oriented Database System: The Story of O2* (Morgan Kauffman, San Mateo, 1989)
9. R.G.G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, F. Velez, *The Object Data Standard: ODMG 3.0* (Morgan Kaufmann, San Francisco, 2000)
10. Db4 objects, http://www.db4o.com (2010)
11. M. Grossnicklaus, *Object Oriented Databases*. Lecture Notes (ETH Zurich, Zurich, 2009)
12. Language Integrated Query, Microsoft Corporation, http://msdn.microsoft.com/en-us/vbasic/aa904594.aspx
13. Objectivity, http://www.objectivity.com/

Chapter 7
Types and Constraints

In this chapter we show how two critical formal notions in this book: types and constraints, apply to two significant practical problems. The first problem is specification of database type systems in a formal manner. The second problem is specification and verification of object-oriented transactions.

Database systems are typically not based on a formally specified type system. This leads to nontrivial problems which we demonstrate in Sect. 7.1. We consider the type system required by the ODMG Standard and establish a collection of the required formal rules showing the typing problems in the ODMG languages. We first give running examples of an ODL schema and OQL queries in Sect. 7.1.1. The notion of a schema is defined in Sect. 7.1.2. Examples of inheritance and the associated formal rules are given in Sect. 7.1.3. In Sect. 7.1.4 we specify the formal rules for structures. Interfaces and their associated formal rules are discussed in Sect. 7.1.5. Classes with extents are the topic of Sect. 7.1.6. Rules for messages are given in Sect. 7.1.7. With this formal apparatus we consider the problems of type checking OQL queries in Sect. 7.1.8. OQL queries and parametric types are discussed in Sect. 7.1.9. Type checking queries with order by clause is the topic of Sect. 7.1.10. Typing problems of checking Java OQL are discussed in Sect. 7.1.11. Type checking of ordered collections and indices is the topic of Sect. 7.1.12.

Section 7.2 is devoted to another key feature of database technologies: transactions. Unlike other approaches, object-oriented schemas and transactions are in our approach equipped with constraints expressed in an object-oriented assertion language (Sect. 7.2.1 and 7.2.2). This is followed by object-oriented consistency issues presented in Sect. 7.2.3. A sample object-oriented schema equipped with very general integrity constraints is specified in Sect. 7.2.4, and sample transactions in Sect. 7.2.5. The relationship between the integrity constraints and queries is discussed in Sect. 7.2.6. Inheritance for object-oriented schemas equipped with constraints is discussed in Sect. 7.2.7. Abstraction techniques for object-oriented schemas are the topic of Sect. 7.2.8. The role of dynamic versus static checking of constraints is explained in Sect. 7.2.9.

S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0_7

7.1 Database Type Systems

7.1.1 ODL and OQL Examples

Let us start with a familiar example specified in ODL: a class *Employee* with attributes *name*, *id* and *dept* and methods *salary*, *hire* and *fire*. The fact that the type of the attribute *dept* is *Department* makes an employee object complex. The *Employee* class has an extent *employees*. This is the set of all objects of the class *Employee* maintained by the object-oriented database system.

```
class Employee
(extent employees)
{ attribute string name;
  attribute short id;
  attribute Department dept;
  float salary();
  void hire(in Department d);
  void fire();
}
```

The class *Employee* above refers to the class *Department*. Specification of the class *Department* is given below. A department object has attributes *name* and *id*, and a method *numOfEmployees*. The class *Department* has an extent called *departments*.

```
class Department
(extent departments)
{ attribute string name;
  attribute short id;
  short numOfEmployees();
}
```

Note that the above definitions differ from similar definitions in an object-oriented programming language in one essential feature: class extents are not available in a typical object-oriented programming language. In addition, method code is specified in specific language bindings.

We consider OQL queries that have the form of a select expression. This is, indeed, the most important class of OQL queries. A simple and self-explanatory OQL query of this form is given below.

```
select well_paid(emp: x.name, sal: x.salary())
from  employees as x
where x.salary() > 50,000
```

The result of the above query is a *bag* of objects representing well paid employees. *well_paid* is the name of an already defined type. The expression *well_paid*(*emp* : *x.name*, *sal* : *x.salary*()) constructs an object of the *well_paid* type. Objects of the *well_paid* type have two attributes named *emp* and *sal*. In order to get a *set* of objects, i.e., to eliminate duplicates from the result, *select distinct* option must be used.

The above query does not involve traversal of the complex structure of an employee object. Because of that it looks just like an SQL query. Let us now consider a complex query that produces objects representing well-paid employees from large departments:

```
select   well_paid(emp: x.name, sal: x.salary())
from     employees as x,
         x.dept as y
where    x.salary() > 50,000
and      y.numOfEmployees() > 100
```

Note how the range variable *y* is bound in the above query to the employee's department. There are of course many other options for OQL queries that we do not consider at all. But it turns out that from the viewpoint of type checking, queries with *order by* clause involve further subtleties. An example of such a query follows:

```
select   well_paid(emp: x.name, sal: x.salary())
from     employees as x,
         x.dept as y
where    x.salary() > 50,000
and      y.numOfEmployees() > 100
order by emp
```

Note that the result of the above query is a *list* of objects, i.e., an ordered collection.

7.1.2 Schemas

A *schema* is a collection of bindings. As such, a schema is a name space. In the ODMG Object Model identifiers in a schema may be bound to interfaces, classes, class extents, and objects.

In the formal system to be developed in the subsequent sections, a statement of the form $\mathcal{S}$ *schema* asserts that a schema $\mathcal{S}$ is *well-formed* or *valid*. The formal rules specify how valid schemas may be formed. A simple and obvious requirement is that a schema $\mathcal{S}$ may not contain duplicate bindings for the same name in $\mathcal{S}$.

If a well-formed schema contains a binding of a name to an interface or to a class, then the specification of such an object type must satisfy the formal rules regarding method signatures and attributes' specification. All other object types referred to in these specifications must obviously be available in the schema in order for the schema to be valid. If a schema $\mathcal{S}$ contains a binding of a name to an object, the type of that object must already be available in $\mathcal{S}$ in order for $\mathcal{S}$ to be valid.

When an interface or a class is added to a schema $\mathcal{S}$, the formal rules specify what conditions must be satisfied in order for $\mathcal{S}$ to be valid. If an object type B is added to a schema $\mathcal{S}$ in such a way that it is derived by inheritance from an object type A, then A must already be in $\mathcal{S}$. In addition, B can introduce signatures of additional methods (those not in A), and, for classes, additional attributes as well. If a class with an extent is added to a schema $\mathcal{S}$, the name of the extent becomes available in $\mathcal{S}$. The name of an extent is implicitly bound to a collection type.

Every schema necessarily contains a collection of predefined bindings. These bindings include the root object type *Object*, other predefined simple types (*string*, *int*, *boolean*, etc.), collection types (*Collection*, *Bag*, *Set*, *List*, and *Array*), the *Database* class, and the *Transaction* class. If $\mathcal{S}$ is a schema, then $dom(\mathcal{S})$ denotes the set of all names whose bindings are defined in $\mathcal{S}$.

7.1.3 Inheritance

The inheritance ordering of object types in the ODMG Object Model is identified with the subtyping ordering. In the ODMG Object Model the top interface is *Object*. All other interfaces are necessarily derived from *Object*. The subtyping ordering (identified with inheritance) has the familiar properties. The rules given below assert reflexivity and transitivity.

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash C <: Object}{\mathcal{S} \vdash C <: C}$$

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash C_3 <: Object, \quad \mathcal{S} \vdash C_2 <: C_3, \quad \mathcal{S} \vdash C_1 <: C_2}{\mathcal{S} \vdash C_1 <: C_3}$$

The inheritance ordering governs substitutability. An object of type C_2 may be substituted where an object of type C_1 is expected, as long as $C_2 <: C_1$. This is expressed formally as follows:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash e : C, \quad \mathcal{S} \vdash y : C_2, \quad \mathcal{S} \vdash x : C_1, \quad \mathcal{S} \vdash C_2 <: C_1}{\mathcal{S} \vdash e < y/x >: C}$$

Suppose that the class *Employee* has been derived by inheritance from the class *Person*. Assume that the class *Person* is equipped with an extent *persons* and the class *Employee* with the extent *employees*.

class Person {
(**extent** persons)
...
}
class Employee: Person {
(**extent** employees)
...
}

According to the ODMG Object Model this implies that the extent *employees* is a subset of the extent *persons*. The following query illustrates type casting feature of OQL:

select well_paid(emp: x.name, sal: ((Employee)x).salary())
from persons **as** x
where ((Employee)x).salary() > 50,000

The formal rule for type casts is:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash e : C_1, \quad \mathcal{S} \vdash C_2 <: C_1}{\mathcal{S} \vdash (C_2)e : C_2}$$

7.1.4 Structures

Specification of interfaces and classes in the ODMG Object Model requires structures containing signatures of methods and attributes as their components. In order for R to be a valid structure, the names of all its fields (attributes or methods) must be distinct. A structure is in fact a static name space or a scope. The types of fields of a structure must already be defined.

If a component of a structure is a method signature $C\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n)$, then the types of the arguments C_i, $1 \leq i \leq n$, and the result type C must be available in the schema. This requirement is expressed by the conditions $C_i <: Object$, $1 \leq i \leq n$, and $C <: Object$. Of course, the names of the arguments must also be distinct ($x_i \neq x_j$ for $i \neq j$), as in the rule given below:

$$\frac{\begin{array}{c}\mathcal{S}\ schema, \quad \mathcal{S} \vdash C_i <: Object,\ 1 \leq i \leq n, \\ x_i \neq x_j,\ 1 \leq i < j \leq n, \\ \mathcal{S} \vdash C <: Object\end{array}}{\mathcal{S} \vdash \{C\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n);\}\ structure}$$

The above rule specifies structures containing method signatures. Rules that specify valid structures containing signatures for attributes are just simple modifications of the above rules, and are thus omitted. Indeed, signatures for attributes

are just a special case of signatures for methods. Note that an extension of the rules specified here is required in order to capture overloading of method names.

Selection of a component of a structure (and thus of an object type as well) is governed by the following familiar rule:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash e : \{C_1\ m_1; \ldots; C_k\ m_k;\}}{\mathcal{S} \vdash e.m_i : C_i}$$

When an object type B is defined using inheritance as an extension of an object type A, the method suit of B must be an extension of the method suit of A. This relationship is defined as $R_B\ <:\ R_A$. R_B contains signatures of additional methods (and attributes if B is a class), i.e., signatures of those methods that are not methods of A. In addition, B may redefine the methods inherited from A. Method redefinition is indicated by an occurrence of the same method name both in R_A and R_B. The type systems underlying the ODMG Standard require that the signature of a redefined method in R_B is the same as the signature of that method in R_A. This means that the types of all arguments, their names, and the type of the method result must be the same.

These requirements are reflected in the definition of the <: relationship for structures. The rules given below are, in fact, an inductive definition of the <: relation. The first step is the obvious reflexive assertion $R_{Object}\ <:\ R_{Object}$, which must hold in any valid schema $\mathcal{S}$.

$$\frac{\mathcal{S}\ schema}{\mathcal{S} \vdash R_{Object}\ <:\ R_{Object}}$$

The inductive step specifies the conditions which a structure R_B must satisfy in order to be a valid extension of a structure R_A, where R_A is assumed to be a valid extension, i.e., $R_A\ <:\ R_{Object}$.

$$\frac{\begin{array}{c}\mathcal{S}\ schema, \quad \mathcal{S} \vdash R_A\ <:\ R_{Object}, \quad \mathcal{S} \vdash R_B\ structure \\ C_B\ m(C_{B_1}\ x_1, \ldots, C_{B_p}\ x_p) \in R_B, \quad C_A\ m(C_{A_1}\ x_1, \ldots, C_{A_k}\ x_k) \in R_A \Rightarrow \\ C_B = C_A, p = k, C_{B_j} = C_{A_j},\ for\ 1 \leq j \leq p\end{array}}{\mathcal{S} \vdash R_B\ <:\ R_A}$$

The above rule which requires invariance of the return type of an inherited method comes from earlier versions of Java available at the time the ODMG Standard was developed.

7.1.5 Interfaces

All object types extend the method suit of *Object* with additional method signatures. The first rule given below simply states that if a schema $\mathcal{S}$ is valid, it must contain a binding for *Object*. Note that in the ODMG Object Model *Object* is an interface.

$$\frac{\mathcal{S}\ \mathit{schema}}{\textbf{interface}\ \mathit{Object}\ R_{\mathit{Object}} \in \mathcal{S}}$$

Similar rules would apply to other predefined bindings for simple types, collection types, etc.

The method suit of any other object type is an extension of R_{Object}. If **interface** $C\ R_C$ is in a schema $\mathcal{S}$, then the method suit $R_C <: R_{Object}$.

In order to add an interface C to a valid schema $\mathcal{S}$, $\mathcal{S}$ may not already contain a binding for C ($C \notin dom(\mathcal{S})$). In addition, the method suit R of C must be well formed. This condition is expressed as $R\ <:\ R_{Object}$.

The rules that govern this ordering are naturally an essential component of this formal system. References to C will typically occur in R. For example, R_{Object} contains occurrences of *Object*. Because of this, the rule given below is recursive. This explains why the condition $\mathcal{S}, C <: \mathit{Object} \vdash R\ <:\ R_{Object}$ for validity of R already contains the assumption that C is a valid interface ($C <: \mathit{Object}$). The expression $\mathcal{S} \cup \{\ \textbf{interface}\ C\ R\}$ denotes a schema obtained by extending $\mathcal{S}$ with a binding for the interface C.

$$\frac{\begin{array}{c}\mathcal{S}\ \mathit{schema},\quad C \notin dom(\mathcal{S}),\\ \mathcal{S}, C <: \mathit{Object} \vdash R\ <:\ R_{\mathit{Object}}\end{array}}{\mathcal{S} \cup \{\textbf{interface}\ C\ R\}\ \mathit{schema}}$$

In order to derive an interface C_2 in a schema $\mathcal{S}$ from an interface C_1 that already exists in $\mathcal{S}$, $\mathcal{S}$ may not contain a binding for C_2. In addition, the method suit R_2 must be well formed. This condition is expressed as $R_2\ <:\ R_1$. The rule that follows is recursive for the same reason as the previous one.

$$\frac{\begin{array}{c}\mathcal{S}\ \mathit{schema},\quad \textbf{interface}\ C_1\ R_1 \in \mathcal{S},\quad C_2 \notin dom(\mathcal{S}),\\ \mathcal{S}, C_2 <: \mathit{Object} \vdash R_2\ <:\ R_1\end{array}}{\mathcal{S} \cup \{\textbf{interface}\ C_2 :\ C_1\ R_2\}\ \mathit{schema}}$$

Note that **interface** $C_1\ R_1 \in \mathcal{S}$ implies $R_1\ <:\ R_{Object}$. Because of this, $R_2\ <:\ R_1$ implies $R_2\ <:\ R_{Object}$ by transitivity of $<:$.

If an interface C is in the schema $\mathcal{S}$, then we can naturally deduce $C <: \mathit{Object}$ from $\mathcal{S}$.

$$\frac{\mathcal{S}\ \mathit{schema},\quad \textbf{interface}\ C\ R\ \in \mathcal{S}}{\mathcal{S} \vdash C <: \mathit{Object}}$$

If we derive an interface C_2 in a schema $\mathcal{S}$ from an interface C_1, then we can deduce $C_2 <: C_1$ from $\mathcal{S}$, as in the rule that follows.

$$\frac{\mathcal{S}\ \mathit{schema},\quad \textbf{interface}\ C_2\ \textbf{extends}\ C_1\ R_2\ \in \mathcal{S}}{\mathcal{S} \vdash C_2 <: C_1}$$

The formal rule for adding a binding of an object a of a class C to a schema $\mathcal{S}$ is now immediate:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash C <: Object, \quad a \notin dom(\mathcal{S})}{\mathcal{S} \cup \{C\ a\}\ schema}$$

7.1.6 Classes

In addition to methods, classes in the ODMG Standard have attributes as well. Since the signatures for attributes are just a special case of the signatures for methods, the rules for interfaces given in the previous section are easily reformulated so that they apply to classes. However, a class in ODL may also be equipped with an extent. When a class with an extent is added to a schema, it introduces two bindings. One of them is a binding for the class itself, and the other binding is for its extent.

Adding a class C with an extent E and a method suit R to a schema $\mathcal{S}$ must satisfy the same requirements as adding a class C with a method suit R to $\mathcal{S}$. This requirement is expressed by the condition $\mathcal{S} \cup \{\textbf{class}\ C\ R\}\ schema$. In addition, $\mathcal{S}$ may not contain a binding for the name E of the extent of the class C. Under these conditions a class C with an extent E and a method suit R may be added to a schema $\mathcal{S}$ to produce a valid schema ($\mathcal{S} \cup \{\textbf{class}\ C\ (\textbf{extent}\ E)\ R\}\ schema$). These considerations are reflected in the formal rule given below.

$$\frac{\mathcal{S} \cup \{\textbf{class}\ C\ R\}\ schema, \quad E \notin dom(\mathcal{S})}{\mathcal{S} \cup \{\textbf{class}\ C\ (\textbf{extent}\ E)\ R\}\ schema}$$

The type of a class extent is *Collection*.

$$\frac{\mathcal{S}\ schema, \quad \textbf{class}\ C\ (\textbf{extent}\ E)\ R \in \mathcal{S}}{\mathcal{S} \vdash E : Collection}$$

A range variable x in an OQL query is bound to a collection e by a construct of the form e **as** x. An alternative notation is x **in** e or simply $e\ x$. The type of x is thus inferred to be the same as the type of elements of the collection e. In type checking a query, the collection of bindings of the relevant schema $\mathcal{S}$ is extended with the bindings of the form e **as** x.

In the type system of the Java binding of the ODMG Standard, elements of *Collection* are of type *Object*.

$$\frac{\mathcal{S}\ schema, \quad S \vdash e : Collection}{\mathcal{S} \cup \{e\ \textbf{as}\ x\} \vdash x : Object}$$

In a type system that supports parametric polymorphism (such as the type system of the C++ binding of the ODMG Standard) one can be a lot more precise about the

type of a class extent. If a class C has an extent E, in this type system we can specify that the type of elements of E is C. This is expressed formally as follows:

$$\frac{\mathcal{S}\ schema, \quad \textbf{class}\ C\ (\textbf{extent}\ E)\ R \in \mathcal{S}}{\mathcal{S} \vdash E : Collection < C >}$$

Likewise, a binding of a range variable to a collection now produces a precise specification of the type of this variable.

$$\frac{\mathcal{S}\ schema, \quad S \vdash e : Collection < C >}{\mathcal{S} \cup \{e\ \textbf{as}\ x\} \vdash x : C}$$

As one may suspect, the above differences in the typing rules for class extents and collections will play a major role in the ability to type check OQL queries in these different type systems.

7.1.7 Objects and Messages

The type of OQL queries that we consider in this paper have object construction expressions in the *select* clause. In this section we consider type checking of such expressions, and then turn our attention to the most critical type checking issue of all: type checking of messages.

Let C be a class with attributes $f_1, f_2, \ldots, f_n$ of respective types $C_1, C_2, \ldots, C_n$. An OQL expression $C(f_1 : e_1, f_2 : e_2, \ldots, f_n : e_n)$ denotes a construction of an object of class C. This expression is correctly typed if the expression e_i is of type C_i for $i = 1, 2, \ldots, n$. In the rule given below the condition $R <: \{C_i\, f_i; \}$ guarantees that C is equipped with a field $C_i\, f_i$.

$$\frac{\begin{array}{c}\mathcal{S}\ schema, \quad \textbf{class}\ CR \in \mathcal{S}, \\ \mathcal{S} \vdash R <: \{C_i\, f_i; \}, \quad \mathcal{S} \vdash e_i : C_i, \quad 1 \le i \le n\end{array}}{\mathcal{S} \vdash C(f_1 : e_1, f_2 : e_2, \ldots, f_n : e_n) : C}$$

Consider now a truly critical formal rule for type checking messages. Type checking messages makes object-oriented type systems different from other type systems. In type checking a message the goal is to avoid run-time errors of the type "message not understood". In other words, static type checking should guarantee that the receiver object is indeed equipped with an appropriate method for handling the message.

An additional subtlety in the ODMG Object Model is that specification of method bodies is not part of ODL. Rather, method bodies must be specified in particular languages (C++, Smalltalk and Java). This obviously makes this most critical type checking rule dependent upon the properties of these three different type systems.

Consider a message $o.m(a_1, a_2, \ldots, a_n)$. Let the type of object o be C_o. The first requirement is that C_o is equipped with an appropriate method signature.

This requirement will be satisfied if either C_o is equipped with such a method signature, or else it is derived by inheritance from a class C with that property. This explains the conditions $C_o <: C$ and $R_C <: \{C_m\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n);\}$, where R_C denotes the method suit of C.

A suitable method signature $C_m\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n)$ must also satisfy the condition that the type C_{a_i} of the argument a_i may be derived by inheritance from C_i for $i = 1, 2, \ldots, n$. Hence the condition $C_{a_i} <: C_i$. Note that we applied the substitution rule both to the receiver and to the arguments of a message.

The condition for type checking the method body is

$\mathcal{S} \vdash body_m < a_i/x_i >: C_m$.

In this condition, $body_m < a_i/x_i >$ denotes the result of substitution of the formal parameter x_i in the body of the method m with the actual parameter a_i. The type of the result is, of course, C_m.

All of this is summarized in the following rule.

$$\frac{\begin{array}{c}\mathcal{S}\ schema, \quad \textbf{interface}\ C\ R_C \in \mathcal{S}, \\ \mathcal{S} \vdash R_C <: \{C_m\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n);\} \\ \mathcal{S} \vdash o : C_o, \quad \mathcal{S} \vdash C_o <: C, \\ \mathcal{S} \vdash a_i : C_{a_i},\ \mathcal{S} \vdash C_{a_i} <: C_i,\ 1 \le i \le n \\ \mathcal{S} \vdash body_m < a_i/x_i > : C_m\end{array}}{\mathcal{S} \vdash o.m(a_1, a_2, \ldots, a_n) : C_m}$$

7.1.8 Type Checking OQL Queries

As stated earlier, OQL queries considered in this paper have the form of select expressions. Among those, we will first consider type checking of OQL queries of the following general form:

select projection
from e_1 **as** x_1, e_2 **as** x_2, ..., e_n **as** x_n
where e

In the above query, the qualification expression e is, of course, required to be of type *boolean*. We consider the most general case in which each e_i for $i = 1, 2, \ldots, n$ is an expression of a collection type. This is in fact explicitly stated in the ODMG Standard for the type of expressions e_i. The type of a range (bound) variable x_i is obviously intended to be the same as the element type of the collection e_i for $i = 1, 2, \ldots, n$. The result of this query is a bag. The type of *projection* determines the type of elements of this resulting bag. If *select distinct* option is used, the result is a set.

In type checking a query expression of the above form, the types of range variables x_i must be inferred first. This is done by inspecting the *from* clause from left to right. The expression e_i is checked to be of a collection type for

$i = 1, 2, \ldots, n$. The type of the range variable x_i is then inferred to be the same as the type of elements of the collection type of e_i.

When the type of the expression e_i is being inferred in this process, the types of the (collection) expressions e_j, for $1 \leq j < i$, is already determined, and so is the type of the range variables x_j, for $1 \leq j < i$. With the type information for the range variables x_i inferred from the *from* clause, type checking must determine that the type of the qualification expression e is *boolean*. In addition, the type of the expression *projection* must be inferred.

The overall type of a query expression of the above form is a suitable collection type. If the type of *projection* is C, then C should obviously be the type of elements of the collection representing the query result. In determining the type of this collection the limitations of the type systems of the ODMG Object Model and the ODMG Java binding again come into play. In both type systems the type of elements of the resulting collection is not specific. Thus in the absence of *select distinct* option, the type of the above query expression is *Bag*, otherwise it is *Set*. Neither type specifies the element type.

Consider the previous example of an OQL query.

select well_paid(emp: x.name, sal: x.salary())
from employees **as** x
where x.salary() > 50,000

The best that we can accomplish in the Java binding type system is to assert that the type of a range variable x is *Object*. An attempt to formulate a rule for type checking of OQL queries in the type system of the Java binding of the ODMG Standard leads to the following:

$$\frac{\begin{array}{c}\mathcal{S}\ \mathit{schema}, \quad \mathcal{S} \vdash e_1 : \mathit{Collection}, \\ \mathcal{S} \cup \bigcup_{i=1}^{k} \{\mathit{Object}\ x_i\} \vdash e_{k+1} : \mathit{Collection}, 1 \leq k < n, \\ \mathcal{S} \cup \bigcup_{i=1}^{n} \{\mathit{Object}\ x_i\} \vdash e : \mathit{boolean}, \\ \mathcal{S} \cup \bigcup_{i=1}^{n} \{\mathit{Object}\ x_i\} \vdash \mathit{projection} : C\end{array}}{\begin{array}{c}\mathcal{S} \vdash \textbf{select}\ \text{projection}\ \textbf{from}\ e_1\ \textbf{as}\ x_1, \ldots, e_n\ \textbf{as}\ x_n \\ \textbf{where}\ e\text{: Bag}\end{array}}$$

Consider now a message $x_i.m(a_{i_1}, a_{i_2}, \ldots, a_{i_n})$ either in the projection or in the qualification of an OQL query where m is not a method of the type Object. Then the type check fails since the type system only allows to conclude that the type of x_i is *Object*. In other words, all queries referring to classes other than *Object* fail a static type check, irrespectively of whether they are type correct or not.

Consider the following Java collections:

```
class Professor {
...
public float salary(){... }
```

```
public Collection courses(){...}
...
}
class Course {
...
public int enrollment(){...}
...
}
Collection professors;
Collection courses;
```

Note that it does not help if we derive classes *EmployeeCollection* and *PersonCollection* from the generic class *Colection*. The signatures of the methods inherited from *Collection* would have to be the same in *EmployeeCollection* and in *PersonCollection*. This is a property of the Java type system, but also of the ODMG Object Model and C++. This property guarantees type safety if dynamic binding of message to methods is applied. Formally, this property is captured by the rules for <:.

An example of a valid OQL query on the above Java collections which fails type checking follows.

```
select    x
from      professors as x,
          x.courses() as y
where     x.salary() > 70,000
and       y.enrollment() > 50
```

The messages *x.salary*() fails the type check because in the ODMG Java binding *x* is of type *Object*. The message *x.courses*() also fails the type check. Even if the type of *x.courses*() could be determined at this point, type checking of the message *y.enrollment*() would still not succeed, because the type of *x.courses*() would be *Collection*, and the type of *y* would be *Object*.

Let us consider possible fix-ups. And indeed, there is one. OQL could be changed to require explicit type casts of range variables down the inheritance relationships. The general form of such a query is now:

select projection
from e_1 **as** $(C_1)x_1$, e_2 **as** $(C_2)x_2$, ..., e_n **as** $(C_n)x_n$
where e

There are two problems with the above fix-up. The first one is that the user is required to specify an apparently redundant type information for range variables. The second negative of the above solution is that type casts require dynamic type

checking. This was surely not the intent of the authors of the ODMG Standard. In order to see how awkward this situation is, one has to keep in mind that optimization is intended to be carried out at compile time. And still a query fails at run time due to a type error!

If explicit type casts are used for range variables, type checking of OQL queries in the type system of the Java binding of the ODMG Standard becomes possible, but only at the expense of dynamic checks. Although the type of e_i is still *Collection*, explicit type casts in the bindings e_i **as** $(C_i)x_i$ of a query **select** projection **from** e_1 **as** $(C_1)x_1$, e_2 **as** $(C_2)x_2$, ..., e_n **as** $(C_n)x_n$ **where** e, now allow the type system to assign a specific type C_i to the range variable x_i. This step is expressed by the following deduction:

$$\frac{\mathcal{S}\ \mathit{schema},\quad \mathcal{S} \vdash e : \mathit{Collection},\quad \mathcal{S} \vdash C <: \mathit{Object}}{\mathcal{S} \cup \{e\ \textbf{as}\ (C)x\} \vdash x : C}$$

The types of range variables are now specific, rather than being just *Object*. This fact is expressed in the following proof rule.

$$\frac{\begin{array}{c}\mathcal{S}\ \mathit{schema},\quad \mathcal{S} \vdash e_1 : \mathit{Collection},\\ \mathcal{S} \cup \bigcup_{i=1}^{k}\{C_i\ x_i\} \vdash e_{k+1} : \mathit{Collection}, 1 \leq k < n,\\ \mathcal{S} \vdash C_i <: \mathit{Object}, 1 \leq i \leq n,\\ \mathcal{S} \cup \bigcup_{i=1}^{n}\{C_i\ x_i\} \vdash e : \mathit{boolean},\\ \mathcal{S} \cup \bigcup_{i=1}^{n}\{C_i\ x_i\} \vdash \mathit{projection} : C,\end{array}}{\begin{array}{c}\mathcal{S} \vdash \textbf{select}\ \text{projection}\ \textbf{from}\ e_1\ \textbf{as}\ (C_1)x_1, \ldots, e_n\ \ \textbf{as}\ (C_n)x_n\\ \textbf{where}\ e\text{: Bag}\end{array}}$$

The type casts of the above form, and the associated dynamic type checks, are very typical for the Java programming language prior to Java 5.0. Suppose that we are given the following two Java collections:

```
Collection professors;
Collection courses;
```

Type checking of the query given below succeeds because of the usage of explicit type casts.

```
select   x
from     professors as (Professor)x,
         x.courses() as (Course)y
where    x.salary() > 70,000
and      y.enrollment() > 50
```

But the run-time checks must be generated.

7.1.9 OQL Queries and Parametric Types

We now show how critical parametric polymorphism is for database type systems and for type checking of OQL in particular. A rule for adding a parametric interface to a schema is given below:

$$\frac{\mathcal{S}\ schema, C \notin dom(\mathcal{S}),\quad \mathcal{S}, T <: Object, C < T >< : Object \vdash R\ <:\ R_{Object}}{\mathcal{S} \cup \{\textbf{interface}\ C < T >\ R\}\ schema}$$

There are two subtleties in checking whether the method suit R of a parametric class $C < T >$ is well formed ($R\ <:\ R_{Object}$). The first one of them is that R contains occurrences of the type parameter T. This is illustrated by the parametric $Collection < T >$ class. The methods *add* and *remove* of this class refer to the type parameter T as the type of their argument. Because of this we must assume that T stands for a valid class ($T <: Object$).

Likewise, occurrences of $C < T >$ may appear in R, as in the $Bag < T >$ class. The methods *union* and *intersection* refer to the type $Bag < T >$ as the type of their argument and the result type is also $Bag < T >$. This is why we must assume that $C < T >$ stands for a valid class, i.e., $C < T > <: Object$. The rule is thus recursive, just as the rule for type checking interfaces that do not have type parameters.

Adding a parametric interface C_2 to a schema $\mathcal{S}$ by deriving C_2 from another parametric interface C_1 already available in $\mathcal{S}$ is:

$$\frac{\mathcal{S}\ schema,\quad \textbf{interface}\ C_1 < T >\ R_1 \in \mathcal{S},\quad C_2 \notin dom(\mathcal{S}),\quad \mathcal{S} \vdash C_2 < T >< : Object \vdash R_2\ <:\ R_1}{\mathcal{S} \cup \{\textbf{interface}\ C_2 < T >:\ C_1 < T >\ R_2\}\ schema}$$

Instantiating a parametric interface $C < T >$ with a specific object type A is governed by the following rule:

$$\frac{\mathcal{S}\ schema,\quad \mathcal{S} \vdash A <: Object,\quad \textbf{interface}\ C < T > R \in \mathcal{S}}{\textbf{interface}\ C < A > R < A/T > \in \mathcal{S}}$$

The above rule states that if we have an interface $C < T >$ in the schema, then effectively we also have an interface $C < A >$ in the schema, for each existing interface A in the schema. There are some obvious additional requirements for avoiding name clashes that may be caused by this substitution.

We are now equipped to state the following: A type system that supports parametric polymorphism restricted to the universal type quantification allows static type checking of OQL queries. This a consequence of the fact that this type system allows compile-time specification of the type of elements of a collection type. This then allows deduction of the specific types of range variables in an OQL query from the bindings of the form e_i **as** x_i. The type of the collection expression e_i is now $Collection < C_i >$. The type of elements of this collection type is now a specific type C_i, where $C_i <: Object$. Because of this the type of the range variable x_i is a specific type C_i. A message $x_i.m(a_1, a_2, \ldots, a_n)$ will now type check if C_i is equipped with a method m with the appropriate signature, otherwise it will fail.

This is summarized in the following rule:

$$\frac{\begin{array}{c}\mathcal{S}\ schema, \quad \mathcal{S} \vdash e_1 : Collection < C_1 >, \\ \mathcal{S} \cup \bigcup_{i=1}^{k} \{C_i\ x_i\} \vdash e_{k+1} : Collection < C_{k+1} >, 1 \leq k < n, \\ \mathcal{S} \cup \bigcup_{i=1}^{n} \{C_i\ x_i\} \vdash e : boolean, \\ \mathcal{S} \cup \bigcup_{i=1}^{n} \{C_i\ x_i\} \vdash projection : C\end{array}}{\begin{array}{c}\mathcal{S} \vdash \textbf{select}\ \text{projection}\ \textbf{from}\ e_1\ \textbf{as}\ x_1, \ldots, e_n\ \textbf{as}\ x_n \\ \textbf{where}\ e\text{: Bag}< C >\end{array}}$$

Note that the type of the result of a query is now also precisely determined. If *select distinct* is used, the type of the result is $Set < C >$. With proper usage of parametric polymorphism, the collections from a previous example now have the following form:

```
Collection<Professor> professors;
Collection<Course> courses;
```

The following query now type checks.

```
select    x
from      professors as x,
          x.courses() as y
where     x.salary() > 70,000
and       y.enrollment() > 50
```

The type of x is now inferred to be *Professor*, and the type of y is inferred to be *Course*. Type checking succeeds. It is performed entirely at compile time. This specifically means that the type system of the C++ binding of the ODMG Standard allows static type checking of OQL queries because C++ is equipped with universal parametric polymorphism.

7.1.10 Queries with Order by Clause

We will consider type checking of OQL queries with *order by* clause of the following general form:

select projection
from e_1 **as** x_1, e_2 **as** x_2, ..., e_n **as** x_n
where e
order by $e'_1, e'_2, \ldots, e'_m$

Additional subtleties in type checking queries with *order by* clause come from the requirement that "message not understood" run-time error messages must be

avoided. This requirement means that we must make sure that the result of a query is equipped with an ordering method.

Consider now type checking of OQL queries with order by clause in the Java binding of the ODMG Standard. Define an interface *Ordered* as follows:

```
interface Ordered: Object {
 boolean leq(Object e);
}
```

In the above interface *leq* stands for less-than-or-equal method. The Java interface *Comparable* could serve the purpose of the interface *Ordered* with somewhat different notation.

If the type of e'_j in an OQL query is C'_j, then the condition that C'_j must be equipped with *leq* method may be expressed as $C'_j <: Ordered$. This is really the only difference.

$$\frac{\begin{array}{c}\mathcal{S}\ schema, \quad \mathcal{S} \vdash e_1 : Collection, \\ \mathcal{S} \cup \bigcup_{i=1}^{k}\{C_i\ x_i\} \vdash e_{k+1} : Collection, 1 \le k < n, \\ \mathcal{S} \vdash C_i <: Object, 1 \le i \le n, \\ \mathcal{S} \cup \bigcup_{i=1}^{n}\{C_i\ x_i\} \vdash e : boolean, \\ \mathcal{S} \cup \bigcup_{i=1}^{n}\{C_i\ x_i\} \vdash projection : C, \\ \mathcal{S} \cup \bigcup_{i=1}^{n}\{C_i\ x_i\} \vdash e'_j : C'_j, \\ \mathcal{S} \vdash C'_j <: Ordered, 1 \le j \le m\end{array}}{\begin{array}{c}\mathcal{S} \vdash \textbf{select}\ \text{projection}\ \textbf{from}\ e_1\ \textbf{as}\ (C_1)x_1, \ldots, e_n\ \ \textbf{as}\ (C_n)x_n \\ \textbf{where}\ e\ \textbf{order by}\ e'_1, \ldots, e'_m\text{: List}\end{array}}$$

The above solution is not applicable to the C++ binding of the ODMG Standard. The reason is that this type system does not have the top of the inheritance ordering. Because of this we cannot define the class *Ordered* the way we did. But a different solution is possible that allows even static type checking of OQL queries with order by clause.

Let us define an interface *Ord_element* in this type system as follows:

```
interface Ord_element< T > {
boolean leq(T e);
}
```

If the type of projection is C, then, in order to guarantee that C is equipped with a method *leq*, we have to have $T <: Ord\_element < T >$. This condition obviously guarantees that an interface T is equipped with a method *leq*. The formal rule has the following form:

$$\frac{\begin{array}{c}\mathcal{S}\ schema, \quad \mathcal{S} \vdash e_1 : Collection < C_1 >, \\ \mathcal{S} \cup \bigcup_{i=1}^{k}\{C_i\ x_i\} \vdash e_{k+1} : Collection < C_{k+1} >, 1 \le k < n, \\ \mathcal{S} \cup \bigcup_{i=1}^{n}\{C_i\ x_i\} \vdash e : boolean, \\ \mathcal{S} \cup \bigcup_{i=1}^{n}\{C_i\ x_i\} \vdash projection : C, \\ \mathcal{S} \cup \bigcup_{i=1}^{m}\{C_i\ x_i\} \vdash e'_j : C'_j, \\ \mathcal{S} \vdash C'_j <: Ord\_element < C'_j >, 1 \le j \le m\end{array}}{\begin{array}{c}\mathcal{S} \vdash \textbf{select}\ \text{projection}\ \textbf{from}\ e_1\ \textbf{as}\ x_1, \ldots, e_n\ \textbf{as}\ x_n \\ \textbf{where}\ e\ \textbf{order by}\ e'_1, e'_2, \ldots, e'_m\text{: } List < C >\end{array}}$$

Note that $T <: Ord\_element < T >$ is in fact a type constraint that appears in F-bounded polymorphism. Hence the conclusion that the type system of the C++ binding of the ODMG Standard allows static type checking of OQL queries with order by clause. In Java extended with parametric polymorphism as in recent editions, static type checking of queries with order by clause becomes possible.

7.1.11 Java OQL

Java OQL refers to the sublanguage of the Java binding of the ODMG Standard which offers OQL facilities in Java. As discussed in Chap. 6, OQL facilities are available in the Java binding of the ODMG Standard in two forms: as *methods* and as *objects*. The same applies to the C++ binding of the ODMG Standard.

In this section we prove that the ODMG approach introduces non-trivial subtleties in type checking. We prove that unexpected results may happen when one tries to combine a strongly and mostly statically typed query language with a strongly and mostly statically typed programming language. These problems led to the design of LINQ that integrates SQL like queries into a major object-oriented language.

Consider first queries as methods. The interface *DCollection* of the Java binding of the ODMG Standard extends the Java *Collection* interface with a variety of self explanatory query methods. A distinctive feature of these methods is that they take a string argument representing an OQL predicate. We repeat below the interface *DCollection* specified in Chap. 6:

```
interface DCollection extends java.util.Collection {
  Object selectElement(String predicate);
  java.util.Iterator select(String predicate);
  DCollection query(String predicate);
  boolean existsElement(String predicate);
}
```

ODMG interfaces *DSet*, *DBag*, *DList* and *DArray* are derived from the above *DCollection* interface. An example of using these collection query methods is given below:

```
DCollection bestPaid;
bestPaid = employees.query(
"forall x in employees: this.salary() >= x.salary()");
}
```

The above query constructs the collection of the best paid employees. The type system of the Java binding cannot handle type checking of OQL query facilities of the *DCollection* class. In a truly integrated type system for Java and OQL we would have the following:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash e : DCollection, \quad \mathcal{S} \cup \{Object\ this\} \vdash unquote(s) : boolean}{\mathcal{S} \vdash e.query(s) : DCollection}$$

In the above rule *unquote*(*s*) stands for the string argument *s* with the surrounding quotes removed. If the predicate *s* contains references of the form *this.m* where *m* is not a method of *Object*, type checking always fails contrary to the intent.

In the type system of the ODMG Java binding we have:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash e : DCollection, \quad \mathcal{S} \vdash s : String}{\mathcal{S} \vdash e.query(s) : DCollection}$$

In this type system, type checking succeeds for any string *s* regardless of whether *s* represents a valid OQL predicate. The reason for this is that the Java type system cannot recognize OQL predicates as valid boolean expressions.

The argument string *s* is passed to the query method at run-time. The rule thus must be applied at run-time. The string must be compiled at run-time in order to make use of the type checker. This is very different and much more complicated from dynamic type checking associated with type casts.

In the C++ binding of the ODMG Standard the class that corresponds to *DCollection* is called *d_Collection*. The type system of the C++ binding of the ODMG Standard cannot handle either type checking of query methods of *d_Collection* class.

The only difference is now that the type of *this* is made specific in the formal rule given below. But this makes little difference in the overall conclusion. In an integrated type system for OQL and C++ we would have:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash e : d\_Collection < C >, \quad \mathcal{S} \cup \{C\ this\} \vdash unquote(s) : boolean}{\mathcal{S} \vdash e.query(s) : d\_Collection < C >}$$

In this type system *this* has a specific type. However, in the C++ type system type checking succeeds for any string argument *s* regardless of whether *s* represents a valid OQL predicate. The situation is the same as in the Java type system.

The other OQL query feature of the Java and the C++ bindings of the ODMG Standard discussed in Chap. 6 is the query class. Instances of this class are queries. Note that this is a truly reflective feature. In more traditional database technologies, a query is a declarative specification of a function, which is either interpreted, or (preferably) compiled into code. In the ODMG approach, queries may be viewed as objects. As such, they are dynamically created, and can even be updated at run-time, just as other objects can. However, this attractive reflective object-oriented feature has non-trivial implications.

```
class OQLQuery{
  // Java constructors
  OQLQuery create(String query); //construct query object
  void bind(Object parameter);
  Object execute();
}
```

Type checking problems of the methods of the above class start with the fact that all that we can assert about the type of the bound variables and the formal parameters of the query is that their type is Object. Thus extensive dynamic type checking is the only option. But this is not the only problem. An illustrative example follows.

```
DBag selectedEmployees;
OQLQuery query = new OQLQuery();
query = create("select well_paid(emp: x.name, sal: x.salary())
                from  employees as x, x.dept as y
                where x.salary() > $1 and y.numOfEmployees() > $2");
query.bind(50,000); query.bind(100);
selectedEmployees = (DBag) query.execute()
```

The OQL query in the above example constructs a collection of objects representing well-paid employees from the large departments. The salary limit (lower bound for high salaries) and the departmental size (the number of employees) are the formal parameters of this query. Note that these parameters are denoted by $i, where i is an integer. In the actual typing rule all we can assert about the type of $i is that it is *Object*. When a query is constructed, its actual parameters are passed at run-time using the *bind* method. The query is subsequently executed by invoking the *execute* method. This method has *Object* as the type of its result. A type cast is thus required into a bag of objects representing the selected employees.

Neither the type system of the Java binding nor the type system of the C++ binding can handle static type checking of messages sent to the object of the class OQLQuery. Parametric polymorphism does not solve this problem. Consider the Java OQLQuery class extended with usage of parametric polymorphism:

```
class OQLQuery< T >{
  //Java constructors
  OQLQuery< T > create(String query); //construct query object
  void bind(Object parameter);
  DCollection< T > execute();
}
```

In a type system with parametric polymorphism, required for the integrated Java and OQL, we would have the following:

$$\frac{\mathcal{S}\ \mathit{schema}, \quad \mathcal{S} \vdash e : \mathit{Query}, \quad \mathcal{S} \cup \{\$i : \mathit{Object}\} \vdash \mathit{unquote}(s) : \mathit{DCollection} < C >}{\mathcal{S} \vdash e.\mathit{create}(s) : \mathit{Query}}$$

For any query s that refers to classes other than *Object*, type checking fails. In the Java type system extended with parametric polymorphism we have:

$$\frac{\mathcal{S}\ schema, \quad \mathcal{S} \vdash e : Query, \quad \mathcal{S} \vdash s : String}{\mathcal{S} \vdash e.create(s) : Query < C >}$$

In this type system type checking succeeds for all strings s regardless of whether s represents a valid OQL query. Note that an OQL query is passed at run-time as the string argument to the *create* method. This requires either an interpretative model or dynamic compilation.

The implications of the above situation are non-trivial. While one would like to have query optimization carried out at compile time if at all possible, this observation shows that it must be performed at run-time.

7.1.12 Typing Ordered Collections and Indices

The ODMG Standard recognizes the importance of ordered collections by providing an explicit interface *List*. In fact, the result of a query is often required to be an ordered collection. Query evaluation algorithms also often depend upon the notion of ordered collections. We now show that correct typing of ordered collections causes non-trivial problems for any of the type systems of the ODMG Standard.

A type system that is equipped with the top object type and supports bounded type quantification (constrained genericity) handles properly typing of ordered collections. With constrained genericity we can define the interface Ordered_Collection as follows:

```
interface Ordered_Collection<T extends Ordered>
                    extends Collection<T> {
  ...
};
```

But the type system of C++ does not have the root object type and cannot handle this situation, even if it is extended with bounded quantification. The top of the inheritance ordering is still required in order to make the above solution possible.

A more general technique solves this problem, but it requires a nontrivial extension of the C++ type system. The technique is F-bounded polymorphism. Recall that the difference between bounded and F-bounded parametric polymorphism is that in F-bounded polymorphism the bound itself is parametric. So if a parametric object type C has a type parameter T whose bound is a parametric class $F < T >$, then this form of bounded type quantification is F-bounded polymorphism.

If the type system of the C++ binding of the ODMG Standard is extended to support F-bounded polymorphism, it can handle static type checking of ordered collections. With proper usage of F-bounded polymorphism, Ordered_Collection may now be specified as follows:

```
interface Ordered_Collection< T extends Ord_element< T >>
                extends Collection < T > {
   ...
};
```

Consider now an illustrative example:

```
interface Employee {
String name();
short id();
//boolean leq(Employee e);
};
```

The object type *Employee* does not satisfy the F-bounded constraint of the interface *Ordered_Collection*. Indeed, it is easy to see that the condition

Employee <: *Ord_element* < *Employee* > does not hold. But if we extend the interface *Employee* with the method *leq*, as indicated by the comment in this interface, we get the following:

```
interface Ord_element<Employee> {
boolean leq(Employee e);
};
```

The F-bounded type constraint *Employee* <: *Ord_element* < *Employee* > is satisfied. The F-bounded constraint thus guarantees that the actual type parameter has the methods of the F-bound, as well as that those methods have correct signatures.

In order to add a parametric class $C < T >$ to a schema $\mathcal{S}$ with an F-bound $F < T >$ for its type parameter, a parametric class $F < T >$ must already be in $\mathcal{S}$. The method suit R of C must be well-formed, i.e., we must have $R <: R_{Object}$. In order to verify this property we must assume that the type parameter T that appears in R stands for a class in the schema ($T <: Object$). In addition, a recursive assumption that we must make is that occurrences of $C < T >$ in R also stand for a valid class. If under these assumptions we can prove that $R <: R_{Object}$, then the **interface** C<T **extends** F< T >> R is well-formed and may be added to the schema $\mathcal{S}$.

These considerations are summarized in the rule for adding a parametric interface with an F-bound to a schema:

$$\frac{\mathcal{S}\ schema, \quad C \notin dom(\mathcal{S}), \quad F < T > \in \mathcal{S}, \quad \mathcal{S}, T <: Object, C < T >: Object \vdash R <: R_{Object}}{\mathcal{S} \cup \{\textbf{interface}\ C < T\ \textbf{extends}\ F < T >>\ R\}\ schema}$$

In order to instantiate a parametric interface $C < T$ **extends** $F < T >>$ with an F-bound to a specific class $C < A >$, A must be available in the schema ($A <: Object$). In addition, A must satisfy the bound ($A <: F < A >$). Under these conditions adding an interface **interface** $C < A > R < A/T >$ to a valid schema $\mathcal{S}$ produces a valid schema. The formal rule follows:

$$\mathcal{S}\ \textit{schema},\quad \textbf{interface}\ C < T\ \textbf{extends}\ F < T >> R \in \mathcal{S},\quad \mathcal{S} \vdash A <: \textit{Object},\quad \mathcal{S} \vdash A <: F < A/T >$$

$$\textbf{interface}\ C < A > R < A/T > \in \mathcal{S}$$

Most viable database systems require indices on top of collections in order to handle queries efficiently. The interface *Index* is given below. It has two type parameters. T is the type of elements of the underlying collection. An index is built on top of a collection using the method *create_index*. *To* is the type of the indexing (search) attributes. *To* consists of a subset of features of T, hence the condition $T <: To$. To must be equipped with the ordering method, hence the F-bounded condition $To <: Ord\_element < To >$.

```
interface Index< To extends Ord_element < To >, T extends To > {
  void create_index(Collection< T > collection);
  Collection< T > select(To key);
  Iterator< T > create_iterator();
};
```

The method *select* returns a collection of elements with a given value of the indexing (search) attributes. The ordering of the underlying collection is reflected in the iterator object. Retrieving elements using the iterator delivers them in the ordering determined by the values of the indexed attributes (of type *To*).

7.2 Transactions

7.2.1 Transactions and Constraints

A transaction is a complex action that acts like a unit with respect to integrity. This means that if a transaction is started in a consistent database state, upon completion of the transaction execution the database will also be in a consistent state. A consistent database state is a state that satisfies the integrity constraints specified in the database schema. In addition, an object-oriented transaction will naturally be specified with a precondition and a postcondition. Transaction execution is illustrated in Fig. 7.1. Note that if a transaction is aborted, the assumption guaranteed by the underlying technology is that a consistent database state will be restored.

The current object technology has nontrivial problems in specifying even classical database integrity constraints, such as keys and referential integrity. No industrial database technology allows object-oriented schemas equipped with general integrity constraints. In addition to keys and referential integrity, such constraints include ranges of values or number of occurrences, ordering, constraints that apply to inheritance, and the integrity requirements for complex objects obtained by aggregation. More general constraints that are not necessarily classical database constraints come from complex application environments and they are often critical for correct functioning of those applications.

Object-oriented schemas are generally missing database integrity constraints because those are not expressible in type systems of mainstream object-oriented programming languages. Since the integrity constraints cannot be specified in a declarative fashion, the only option is to enforce them procedurally with nontrivial implications on efficiency and reliability. In a typed constraint-based database technology, the constraints would fit into the type systems of object-oriented languages and they should be integrated with reflective capabilities of those languages so that they can be introspected at run-time.

Most importantly, all of that is not sufficient if there is no technology to enforce the constraints, preferably statically, so that expensive recovery procedures will not be required when a transaction violates the constraints at run-time. The ability to verify statically that a transaction implemented in a mainstream object-oriented language satisfies the database integrity constraints has been out of reach for a long time. A pragmatic goal has been static automatic verification which hides completely the prover technology from the users.

A key observation is that if it is not possible to verify that transactions satisfy the schema integrity constraints, then it is not possible to truly guarantee the ACID properties of the transaction model. Recall that ACID stands for atomicity, isolation, consistency and durability. The classical transaction theory is based on serializable executions. A concurrent execution is serializable if it has an equivalent serial execution. The equivalence is defined with respect to the impact on database integrity. A serializable concurrent execution of a set of transactions has the property that it will maintain the schema integrity constraints only as long as the individual

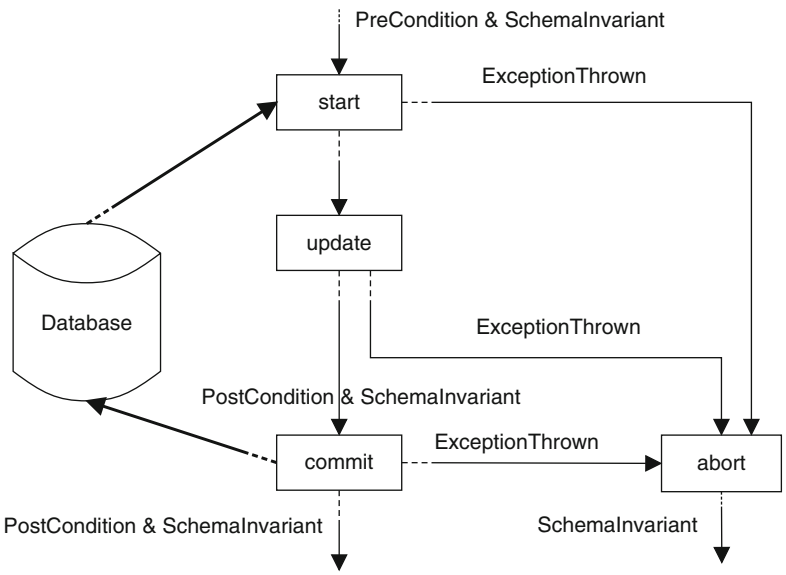


Fig. 7.1 Transaction execution

transactions by themselves (i.e., in isolation) satisfy those constraints. This and other related classical results on the locking protocols that guarantee serializability are actually independent of the particular form of the integrity constraints. But whatever those constraints are, the individual transactions must satisfy them in order for the serializability results to apply.

7.2.2 Schemas and Transactions

In addition to inheritance, the key abstraction technique for modeling complex applications is aggregation. This abstraction is well understood in semantic data models, but in the object-oriented model it has specific implications. A complex object defined by aggregation is represented by its root object called the owner along with references to the immediate components of the owner specified as its representation fields. References to other objects do not represent components of that object. This way a complex object is defined as a logical unit that includes all of its components. Constraints that apply to objects defined by aggregation may now be specified in such a way that they refer both to the owner object and to the components defined by its representation fields. In a flight scheduling application developed in this section, a flight scheduling object is defined as an aggregation of flights, planes and airports, as illustrated in Fig. 7.2.

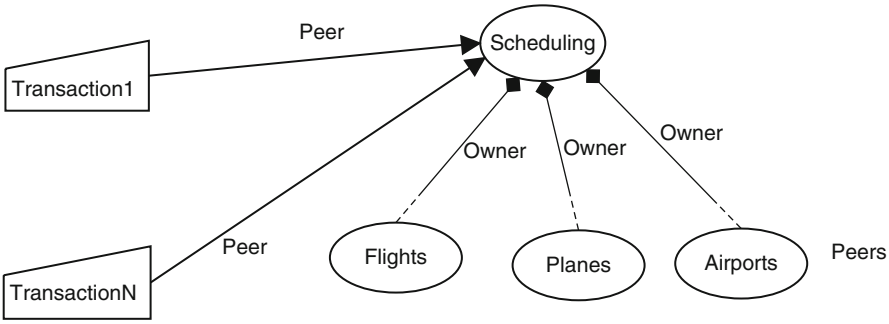


Fig. 7.2 Owners and peers in flight scheduling

The notion of ownership comes with a related semantic modeling notion. Objects that have the same owner are called peers. The relationship among objects *Flights*, *Planes* and *Airports* is clearly not the ownership relationship. These objects are peers as they have the same owner, the scheduling object.

The peer relationship has a role that may be even independent from the notion of ownership. Consider the relationship of a transaction object and its associated schema object. The relationship between a transaction object and a schema object is also modeled as a peer relationship. A transaction is not a component of a schema,

and a schema is not a component of a transaction. There are multiple transactions accessing the same schema object and all of them cannot own the schema object. An object can have at most one owner.

In the model of transactions presented in this section, the class *Transaction* is bounded parametric, where the bound type is the type of schema to which a specific transaction type is bound. This makes it possible for a particular transaction class to be compiled with respect to a specific schema type. The notation in the code given below follows Spec#. For presentation purposes, the notation in this section is more mathematical than the Spec# notation. However, there is a direct correspondence between this notation and the Spec# notation.

T! denotes a non-null object type, i.e., an object type that does not allow null references. The attribute [Peer] indicates that the relationship between a transaction object and its associated schema object is specified as the peer relationship. The attribute [SpecPublic] denotes private components that can be used as public only in specifications.

```
public interface Schema {...}
public class Transaction <T> where T: Schema {
   [SpecPublic][Peer] protected T! schema;
public Transaction(T! schema){ this.schema = schema;}
}
```

In addition to the above abstractions, inheritance is naturally an essential modeling abstraction which we do not show in the above diagram. The model of this application includes inheritance hierarchies of different aircraft types and different airport types, as well as an inheritance hierarchy of different transaction types. The interplay of inheritance and constraints is discussed in Sect. 7.2.7.

7.2.3 Levels of Consistency

The schema integrity constraints are typically violated during transaction execution and then the constraints are reinstated when the transaction is completed, so that the constraints should hold at commit time. The mechanism for handling correctly these situations is illustrated below by the structure of a transaction that closes an airport:

```
expose(flight scheduling){
close airport;
cancel all flights to or from the closed airport;
}
```

After the first action the referential integrity constraints are temporarily violated to be reinstated after the second action of cancelling all flights to or from the closed airport. The purpose of the **expose** block is to indicate that the schema object invariants may be violated in this block. Otherwise, the verifier will indicate violation of the schema invariants. In the **expose** block the object is assumed to be

in a mutable state and hence violation of the object invariants is allowed. Outside of the **expose** block, assignments that possibly violate the invariants will be static errors. Different situations that may occur with respect to the object state and its satisfaction of the object invariants are summarized below:

- **Valid** object state—object invariants hold, updates must satisfy the invariants.
- **Mutable** object state—object invariants are not required to hold, updates are allowed to violate them
- **Consistent** object state—the object is in a **valid** state and
  - the object does not have an owner or
  - the owner is in a **mutable** state
- **Committed** object state—the object is in a **valid** state and
  - the object has an owner
  - the owner is also in a **valid** state.

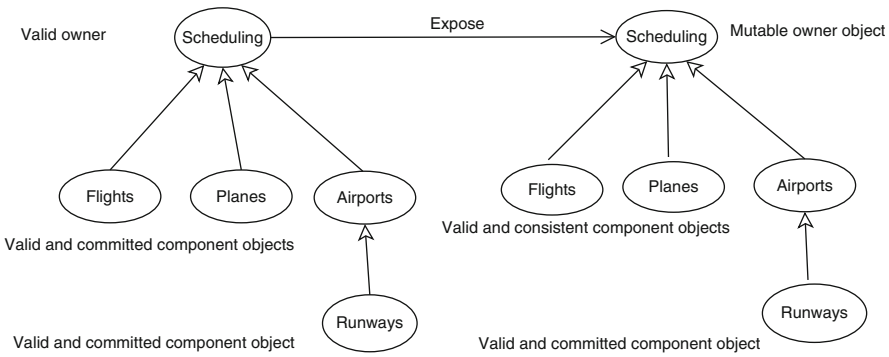


Fig. 7.3 Flight scheduling consistency states

When a transaction operates on an object, the implicit assumption is that the object is in a consistent state. This means that either the object does not have an owner to put restrictions on the object, or that the object has an owner, and the owner is in a mutable state, hence it allows update actions on the object. Since the object is in a consistent state, its state is valid and its components are thus in a committed state. In order to update the receiver and the states of its components, the receiver state must be changed to a mutable state using the *expose* block. This will also change the state of the components of the object from committed to consistent, so that methods can be invoked on them. The notions of valid, mutable, consistent and committed objects, and the effect of the expose statement, are illustrated in Fig. 7.3.

There is an obvious alternative to viewing a transaction and its associated schema as peers: just omit any ownership or peer attributes. But in fact, using the peer relationship has important implications for transaction verification. A transaction is verified under the assumption that the schema integrity constraints hold when the

transaction is started. If this condition is not satisfied, transaction verification makes no sense. So we really have to guarantee this condition.

Spec# adds an implicit precondition for peer consistency so that a transaction can assume this condition in its verification. This applies to in-bound parameters and the receiver of any method. The implicit postcondition for peer consistency also applies to all out-bound parameters and return values. The caller of a method is required to satisfy the peer consistency requirement. This means that an object and its peers must be valid, and their owner must be exposed first before an update is performed.

7.2.4 Constraints for Schemas

We now consider a specific schema in which the core object type is defined using the aggregation abstraction and the ownership model along with the associated integrity constraints. The *FlightScheduling* schema contains specification of three database collections: a list of airplanes, a list of airports, and a list of flights.

The schema *FlightScheduling* exhibits two cases of the aggregation abstraction as supported by the Spec# ownership model. The attribute [Rep] indicates that the lists of flights, airports and airplanes are components of the flight scheduling object which becomes their owner. The attribute [ElementsRep] indicates that list elements are also components of the flight scheduling object. These elements are then peers according to the Spec# ownership model. This has implications on invariants that can now be defined to apply to entire complex objects, i.e., including their components determined by the [Rep] and [ElementsRep] fields. These constraints are called ownership-based invariants.

```
public class FlightScheduling: Schema {
[SpecPublic][Rep] [ElementsRep] private List<Airplane!>! airplanes;
[SpecPublic][Rep] [ElementsRep] private List<Airport!>! airports;
[SpecPublic][Rep] [ElementsRep] private List<Flight!>! flights;
// constraints
}
```

In the collection of airplanes the key is *Id*, in the collection of airports the key is *Code*, and in the collection of flights the key is *FlightId*. The first referential integrity constraint specifies that each flight in the collection of flights refers to a unique airplane in the collection of airplanes. The remaining (omitted) referential integrity constraints specify that each flight in the collection of flights refers to a unique airport as its origin and a unique airport as its destination.

```
invariant ∀{int i ∈ (0: flights.Count), int j ∈ (0: flights.Count);
    flights[i].FlightId = flights[j].FlightId ⇒ flights[i].equals(flights[j])};
invariant ∀{int i ∈ (0: flights.Count);
    ∃ unique {int j ∈ (0: airplanes.Count); airplanes[j].equals(flights[i].Airplane)}};
```

A class is in general equipped with an invariant which specifies valid object states. The schema integrity constraints are specified above as class invariants. These assertions allow usage of universal and existential quantifiers as in the first-order predicate calculus, as well as combinators typical for database languages such as *min*, *max*, *sum*, *count*, *avg* etc. These constraints in the above schema refer to private components of the schema object. As explained earlier, the attribute [SpecPublic] means that these private components can be used as public only in specifications. Typically, such components will also be defined as public properties with appropriately defined *get* and *set* methods so that access to them can be controlled.

Spec# constraints limit universal and existential quantification to variables ranging over finite intervals. The above constraints contain specifications of half open intervals. The limitation that quantifiers are restricted to integer variables ranging over finite intervals was a design decision to sacrifice expressiveness in order to allow automatic static verification. This limitation is no problem in the application considered in this section as the above schema shows.

The above schema contains non-null object types (indicated by the symbol *!*) that capture a very specific object-oriented integrity constraint. A frequent problem in object-oriented programs is an attempt to dereference a null reference. If this happens in a database transaction, the transaction may fail at run-time with nontrivial consequences. The Spec# type system allows specification of non-null object types. Static checking will indicate situations in which an attempt is made to access an object via a possibly null reference.

7.2.5 Sample Transactions

Each class that a schema refers to is also equipped with its constraints as illustrated below for the class *Flight*. The relationship between a flight object and the associated airplane object is defined as a peer relationship for the reasons explained in Sect. 7.2.2. The invariants include the obvious ones: the origin and the destination of a flight must be different and the departure time must precede the arrival time. If the current time is greater than the arrival time or the current time is less than the departure time, the status of the flight must be idle. If the current time is greater than the departure time and less than the arrival time the flight status must be either flying, landing or takeoff.

invariant to $\neq$ from;

invariant departureTime $<$ arrivalTime;

invariant DateTime.Now $>$ arrivalTime $\Rightarrow$ this.flightStatus = FlightStatus.Idle;

invariant DateTime.Now $<$ departureTime $\Rightarrow$
this.flightStatus = FlightStatus.Idle;

invariant DateTime.Now $\geq$ departureTime $\wedge$ DateTime.Now $\leq$ arrivalTime $\Rightarrow$

this.flightStatus = FlightStatus.TakeOff ∨
this.flightStatus = FlightStatus.Flying ∨
this.flightStatus = FlightStatus.Landing;

The constraints specified in this section include some classical database integrity constraints such as keys and referential integrity, and in addition constraints that are not typical for the existing database technologies, object-oriented in particular. In fact, we are not aware of a database technology that allows constraints of the above variety.

To make the job of the verifier possible, specification of methods that change the object state, such as database updates, requires specification of the frame conditions. This is done by the *modifies* clause, which specifies those objects and their components that are subject to change. The frame assumption is that these are the only objects that will be affected by the change, and the other objects remain the same. An attempt to assign to the latter objects will be a static error.

A sample instantiation of the class *Transaction* by the flight scheduling schema are given below.

```
public class ScheduleFlightTransaction:
                        Transaction<FlightScheduling> {
public Flight? scheduleFlight (String! flightId,
                String! toAirportCode, String! fromAirportCode,
                DateTime departure, DateTime arrival, Airplane! plane)
  // constraints
  {// transaction body }
}
```

Flight? in the above code is an explicit notation for a type that may contain a null value. The preconditions of the transaction *scheduleFlight* are that the flight id does not exist in the list of flights, that its origin (denoted *fromAirportCode*) and its destination (denoted *toAirportCode*) must refer to existing (valid) airport codes, and that the departure time precedes the arrival time. Valid airport codes are kept in a table *ValidCodes*. The transaction *scheduleFlight* modifies only the list of flights as specified in its **modifies** clause. The postcondition guarantees that the newly scheduled flight exists in the list of flights.

```
requires toAirportCode ≠ fromAirportCode;
requires ∀ {int i ∈ (0: schema.Flights.Count);
                schema.Flights[i].FlightId ≠ flightId };
requires ∃ unique {int i ∈ (0: schema.Airplanes.Count);
                schema.Airplanes[i].equals(plane)};
requires ∃ unique {String code ∈ ValidCodes.airportsCodes;
                        code = toAirportCode};
requires ∃ unique {String code ∈ ValidCodes.airportsCodes;
                        code = fromAirportCode };
```

requires departure < arrival;

modifies schema.flights;

ensures ∃ **unique** {int i ∈ (0: schema.Flights.Count);
schema.Flights[i].FlightId = flightId };

The first precondition of the transaction *cancelFlight* specifies that there exists a unique flight in the collection of flights with the given id of the flight to be deleted, denoted *flightId*. The second precondition specifies a requirement that the flight departure time is greater than the current time. The **modifies** clause specifies that this method modifies the collection of flights. The postcondition specifies that the cancelled flight does not appear in the list of flights.

requires ∃ **unique** {Flight! flight ∈ schema.Flights;
flight.FlightId = flightId};
requires ∀ {Flight! flight ∈ schema.Flights;
flight.FlightId = flightId ⇒
flight.departureTime > DateTime.Now };
modifies schema.flights;
ensures ∀ {Flight! flight ∈ schema.Flights;
flight.FlightId ≠ flightId };

The preconditions for the transaction *redirectFlight* are that the id of the flight to be redirected, denoted *flightId*, must exist in the list of flights, and that its status must not be landing. In addition, the new destination is required to be different from the flight origin. This transaction modifies just the list of flights. The postcondition ensures that the destination of the redirected flight has indeed been changed in the list of flights to *newDest*.

requires ∃ **unique** {Flight flight ∈ schema.flights;
flight.FlightId = flightId ∧ (flight.FlightStatus ≠ FlightStatus.Landing)};
requires ∀ {Flight flight ∈ schema.flights;
flight.FlightId = flightId ⇒ flight.from ≠ newDest };

modifies schema.flights;

ensures ∀ {Flight! flight ∈ schema.Flights;
flight.FlightId = flightId ⇒ flight.to = newDest };

7.2.6 Constraints and Queries

Queries are pure methods. Pure methods are functions that have no impact on the state of objects, database objects in particular. Interplay of constraints and queries is illustrated below. The attribute [Pure] indicates a pure method and *result* refers to its result.

An example of a query (hence pure) method is *flightsDepartureBetween* which returns a list of flights whose departure time is within a given interval. The preconditions require that the time interval is not empty (i.e. the initial time is less than the end time) and that the initial time is greater than the current time. The postcondition ensures that the flights that are returned by this method have the departure times within the specified bounds.

```
[Pure] public List<Flight!>? flightsDepartureBetween
                (DateTime beginDateTime, DateTime endDateTime)
requires beginDateTime < endDateTime;
requires beginDateTime > DateTime.Now;
ensures ∀ {Flight! f ∈ result;
                f.departureTime ≥ beginDateTime ∧
                f.departureTime < endDateTime };
{// method body }
```

The body of this method is specified as a LINQ query given below:

```
// open db
IEnumerable<Flight> flights =
from Flight flight ∈ db
where flight.departureTime ≥ beginDateTime ∧
      flight.departureTime < endDateTime
select flight;
   // close db;
```

A native query in Db4 Objects (details omitted) has the following form:

```
// open db
IList<Flight!>? flights =
db.Query<Flight!>(delegate(Flight! f) {
return (f.departureTime ≥ beginDateTime ∧
        f.departureTime < endDateTime); };
// close db;
```

7.2.7 Specification Inheritance

Specifications of constraints in a class are inherited in its subclasses. In addition, method postconditions and class invariants may be strengthened by additional constraints. Method preconditions remain invariant. This discipline with respect to inheritance of constraints is a particular case of behavioral subtyping. It guarantees that an instance of a subtype may be substituted where an instance of the supertype is expected with no behavioral discrepancies.

Consider the class *Airport* given below in which an airport object is the owner of its list of runways, as well as of the specific runways in that list.

```
public class Airport {

[SpecPublic] private String code;
[Additive] protected int numRunways;
[SpecPublic] [Rep] [ElementsRep] protected List<Runway!>! runways;
// methods and constraints
}
```

The invariants of this class specify that the number of runways must be within the specified bounds. In addition, there are ownership based invariants on flights in the take-off and landing queues in the runways. These are invariants that relate properties of the owner and its components and hence apply to the entire complex object of an airport. These constraints include a constraint that one and the same flight cannot be in two different queues belonging to different runways. In order to make it possible for subclass invariants to refer to the field *numRunways*, Spec# requires the attribute [Additive] in the specification of this field.

```
invariant numRunways ≥ 1 ∧ numRunways ≤ 30;
invariant runways.Count = numRunways;
invariant /* No multiple occurrences of the same flight in runways*/
```

Methods *addRunway* and *closeRunway* along with the associated constraints are specified as follows:

```
public virtual void addRunway(Runway! runway)

modifies runways, numRunways;
ensures ∃ {Runway! r ∈ runways; r.equals(runway)};
{//code }
public virtual void closeRunway (Runway! runway)
modifies runways, numRunways;

ensures numRunways > 0;
ensures numRunways = old(numRunways) - 1;
ensures ∀ { Runway! r ∈ runways; ¬ r.equals(runway)};
{// code }
```

Consider now a class *InternationalAirport* derived by inheritance from the class *Airport*. The class *InternationalAirport* inherits all the invariants from the class *Airport*. In addition, it adds new invariants that are conjoined with the inherited ones. These additional invariants require that the number of runways is higher than the minimum required by an airport in general. Furthermore, an additional requirement is that there exists at least one runway of the width and length suitable

for international flights. This is expressed using a model field *IntRunway*. The notion of a model field is explained in Sect. 7.2.8 that follows.

public class InternationalAirport: Airport {

invariant numRunways ≥ 10;
invariant ∃ {Runway! r ∈ Runways; r.IntRunway };
// IntRunway is a boolean model field in Runway
// constructor, methods

public override void closeRunway (Runway! runway)
ensures numRunways ≥ 10;
ensures ∃ {Runway! r ∈ runways; r.IntRunway };
{// code}
}

Overriding of the method *closeRunway* demonstrates the rules of behavioral subtyping. One would want to strengthen the precondition of this method by requiring that there is more than one international runway at an international airport or else the invariant for the international airport will be violated. But that is not possible by the rules of behavioral subtyping. Otherwise, users of the class *Airport* would see behavior of the method *closeAirport* that does not fit its specifications in the class *Airport*. This would happen if the airport object is in fact of the run-time type *InternationalAirport*. The *modifies* clause cannot be changed either for similar reasons. But the postcondition can be strengthened as in the above specifications. The postcondition now ensures that the number of runways is greater than or equal to ten and that there exists at least one international runway after the method execution. These are specific requirements for international airports.

Specification inheritance has implications on behavioral subtyping of parametric types that follow well-known typing rules for such types. For example, if we derive a schema *InternationalFlightScheduling* by inheritance from the schema *FlightScheduling*, *Transaction< InternationalFlightScheduling>* will not be a subtype of the type *Transaction<FlightSchedling>*, and hence not a behavioral subtype either.

A class frame is the segment of the object state which is defined in that class alone. A class frame does not include the inherited components of the object state. An invariant of a class will include constraints that apply to its frame, but it may also further constrain the inherited components of the object state. For example, an object of type *International Airport* has three class frames. These class frames correspond to classes *Object*, *Airport* and *InternationalAirport*.

The notions *valid* and *mutable* apply to each individual class frame. The notions *consistent* and *committed* apply to the object as a whole. So an object is consistent or committed when all its frames are valid. The *expose* statement changes one class frame from valid to mutable. The class frame to be changed is specified by the static type of the segment of the object state to be changed. For example, the body of the method *closeAirport* of the class *InternationalAirport* has the following form:

```
assert runways ≠ null;
additive expose((Airport)this){
        runways.remove(runway);
        numRunways–; }
```

7.2.8 Abstraction

Typical classes in this application have private fields that are made public only for specification purposes. Examples are fields *code* and *runways* in the class *Airport*. Users of this class would clearly have the need to read the code of an airport, and some users would have the need to inspect the runways of an airport. On the other hand, these fields are naturally made private as users are not allowed to access them directly in order to change them.

The basic mechanism for exposing a view of the hidden object state is to use public pure methods. A related technique is to use public properties. A public property is defined as a pair of *get* and *set* methods. The constraints in the *set* method control correctness of an update to a backing private field. A property *Runways* of the class *Airport* is specified below.

```
public List<Runway!>! Runways {
get { return runways;}
[Additive] set {
requires value ≠ null;
ensures runways = value;
ensures /*no multiple occurrences of the same flight in runways*/
//code
    }
}
```

Property getters are pure methods by default. Properties represent a general abstraction mechanism as the value of a property returned by the method *get* need not just be the value of a backing field, but it may be computed in a more complex way from the hidden (private) components of the object state.

Another abstraction mechanism is based on model fields. A model field is not an actual field and hence it cannot be updated. The model fields of an object get updated automatically at specific points in a transaction. An example of a model field is *IntRunway* of the class *Runway* specified below.

```
model bool IntRunway {
satisfies IntRunway = (length ≥ 80 ∧ width ≥ 10);}
```

Unlike pure methods, model fields do not have parameters. But they often simplify reasoning. The verifier checks that the *satisfies* clause can indeed be

satisfied, i.e., that there exists an object state that satisfies this clause. The *satisfies* clause may depend only on the fields of *this* and the objects owned by *this*. The *satisfies* clauses may be weaker in superclasses, and strengthened in subclasses.

7.2.9 Dynamic Checking of Constraints

Static verification of a transaction ensures that if the transaction is started in a consistent database state (the schema invariants hold) and the transaction precondition is satisfied, the schema invariants and the postcondition of the transaction will hold at the point of the commit action. The application program that invokes the transaction must satisfy the above requirements at the start of the transaction, and will be guaranteed that the postcondition and the schema invariants will hold at the end of the transaction execution.

Static verification does not say anything about what happens if the schema integrity constraints or the transaction precondition are not satisfied. What it says is that the code of a successfully verified transaction is correct with respect to the integrity constraints. Violation of constraints may still happen at run-time given the actual data. For example, a transaction may be invoked with arguments that do not satisfy the precondition and hence the verification results do not apply. This is why the dynamic checks that Spec# generates are essential. JML does the same, but it does not offer automatic static verification of code. Run-time tests will generate exceptions indicating violation of constraints. The transaction must handle these exceptions properly. Static verification guarantees that in the absence of such exceptions the results of transaction execution will be correct with respect to the integrity constraints. This extends to concurrent serializable executions of a set of transactions that have been statically verified.

Explicit dynamic checks may be used to verify that the constraints hold at run-time. This is illustrated below with a dynamic check of the precondition of the transactions *addAirport* in which *a* denotes the airport that should be added. The precondition of *addAirport* is that an airport with the code of the new airport does not exist in the database. This can be checked only dynamically by querying the database and asserting that this condition is satisfied.

```
IList<Airport!>? airports =
db.Query<Airport!>(delegate(Airport! arp){
return (arp.Code = a.Code);};
assert airports = null;
```

Ensuring that a new airport has been added to the database is accomplished by querying the database and asserting that the list of airports in the database with the new code is not empty and that the newly inserted airport is indeed in the database.

IObjectSet? airportsSet =
db.Query(typeof(Airport!));
assert ∃ **unique** {Airport! arp ∈ airportsSet; arp.equals(a)};

7.3 Bibliographic Remarks

The type system developed in this chapter is a formal specification of the notions of the Object Oriented Database Management Group (ODMG), the industrial standard for object-oriented databases [7]. This type system comes from Alagić [1]. See also [3].

The classical paper on consistency of database transactions is [8]. The first attempt of formal transaction verification based on computational logic was published in [11]. Some subsequent results are [6, 12]. More recent results are [4, 5]. The presentation of the transaction specification and verification material is from [2]. The constraints are based on the Spec# notation [9, 10].

7.4 Exercises

1. Specify the typing rule for LINQ comprehension queries.
2. Specify the typing rule for LINQ lambda queries.
3. Specify the precondition and the postcondition of the transaction *RedirectFlight* of the *FlightScheduling* application.
4. Specify the invariants for the following schema:

   class Stock {
   String stockId();
   float price();
   }
   class Broker {
   String brokerId();
   String name();
   Set<Stock> stocks();
   }
   class StockMarket: Schema {
   [SpecPublic][Rep] private Set<Stock> stocks;
   [SpecPublic][Rep] private Set<Broker> brokers;
   // invariants
   // public methods for insertions, updates, and deletions of stocks and brokers
   }
5. Specify the preconditions and the postconditions of the public methods for insertion, deletion and update of objects specified in the stock market schema.

6. Specify a transaction *MergeStocks* for the stock market schema.
7. Extend the previously specified ODMG *Publication* schema in C# by the schema constraints specified as invariants.
8. Specify a collection of representative transactions for the *Publication* schemas in terms of their preconditions and postconditions. Specify the expose blocks in the code of the above transactions.
9. Extend the previously specified airport schema in C# with schema constraints expressed as invariants.
10. Extend the collection of previously implemented transactions for the airport schema with their preconditions and postconditions.
11. Specify the expose blocks in the code of the above transactions.
12. Specify the precondition and the postcondition of the transaction action *commit*. This specification should include the schema invariant.
13. Specify the precondition and the postcondition of the transaction action *abort*. This specification should include the schema invariant and the transaction postcondition.
14. Specify formally the following transaction property: If a verified transaction is invoked, it will eventually either commit or abort. Assume that a transaction is equipped with pure boolean methods *atStart*, *atCommit* and *atAbort*. Make use of the operator *eventually* so that *eventually(p)* evaluates to true if the predicate *p* holds some time in the future.
15. Specify formally the following transaction integrity property: If a verified transaction is invoked in a consistent database state and its precondition is satisfied, upon commit of the transaction the database will be in a consistent state.
16. Specify formally the following transaction correctness property: If a verified transaction is invoked in a consistent database state and its precondition is satisfied, upon commit of the transaction the transaction postcondition will hold.
17. Specify formally the following transaction property: A verified transaction is allowed to violate database integrity constrains only in expose blocks. Outside of the expose blocks the integrity constraints will hold.
18. Specify formally the following transaction aggregation property: A verified transaction cannot permanently violate the integrity constraints associated with complex objects defined by aggregation by updating components of such complex objects.

References

1. S. Alagić, Type checking OQL queries in the ODMG type systems. ACM Trans. Database Syst. **24**(3), 319–360 (1999)
2. S. Alagić, A. Fazeli, Verifiable object-oriented transactions, in *Proceedings of COB 2012 (Concurrent Objects and Beyond)*, ed. by G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, K. Taura. Lecture Notes in Computer Science, vol. 8665 (Springer, Berlin, 2014), pp. 251–275

3. S. Alagić, M. Royer, Genericity in Java: persistent and database systems implications. VLDB J. **17**(4), 847–878 (2007)
4. S. Alagić, P. Bernstein, R. Jairath, Object-oriented constraints for XML schema, in *Proceedings of International Conference on Objects and Databases (ICOODB 2010)*, ed. by A. Dearle, R.V. Zicari. Lecture Notes in Computer Science, vol. 6348 (Springer, Berlin, 2010), pp. 101–118
5. I.G. Baltopoulos, J. Borgstrom, A.G. Gordon, Maintaining database integrity with refinement types, in *Proceedings of Object-Oriented Programming (ECOOP) 2011*, ed. by M. Mezini. Lecture Notes in Computer Science, vol. 6813 (Springer, Berlin, 2011), pp. 484–509
6. V. Benzanken, X. Schaefer, Static integrity constraint management in object-oriented database programming languages via predicate transformers, in *ECOOP'97 — Object-Oriented Programming*, ed. by M. Akşit, S. Matsuoka. Lecture Notes in Computer Science, vol. 1241 (Springer, Berlin, 1997), pp. 60–84
7. R.G.G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, F. Velez, *The Object Data Standard: ODMG 3.0* (Morgan Kaufmann, San Francisco, 2000)
8. K.P. Eswaran, J.N. Grey, R.A. Lorie, I.L. Traiger, The notions of consistency and predicate locks in a database system. Commun. ACM **19**, 624–633 (1976)
9. K.R. Leino, P. Muller, Using Spec# language, methodology, and tools to write bug-free programs, Microsoft Research (2010), http://research.microsoft.com/en-us/projects/specsharp/
10. Microsoft Corp., Spec#. http://research.microsoft.com/specsharp/
11. T. Sheard, D. Stemple, Automatic verification of database transaction safety. ACM Trans. Database Syst. **14**, 322–368 (1989)
12. D. Spelt. S. Even, A theorem prover-based analysis tool for object-oriented databases, in *Tools and Algorithms for the Construction and Analysis of Systems*, ed. by W.R. Cleaveland. Lecture Notes in Computer Science, vol. 1579 (Springer, Berlin, 1999), pp. 375–389

Index

A
Abort method, 148, 152, 160, 201
Abstract
  class, 17–20, 37, 43, 56, 68, 69, 120, 121, 155, 158, 160
  data type, 1
ACID properties, 148, 155, 187
Active object, 105
Aggregation, 188, 191, 201
Ambient, 105, 120–124, 131
Argument, 2, 3, 8, 10, 20–22, 34, 37, 38, 42, 45, 51–56, 59–61, 63–65, 67, 72, 77, 83, 86, 87, 93, 94, 98, 99, 101–103, 106, 113–116, 120, 125, 126, 128, 130, 138, 140–142, 145–147, 153, 156, 158, 159, 169, 170, 174, 178, 181, 182, 184, 199
Array, 16, 21, 26, 29, 30, 32–34, 47, 49, 51, 52, 54, 56–59, 61, 64–67, 69, 91–93, 99, 102, 114, 130, 168
ArrayLoad, 65–67, 78
  operation specification, 65–67
ArrayStore, 67, 78
  operation specification, 66, 67
Assertion
  class invariant, 31, 125, 192
  loop invariant, 25, 28, 29, 33–34, 36, 47
  object-invariant, 25, 35–37, 40, 43–45
  postcondition, 25, 26, 29–33, 35, 37, 38, 40, 41, 45, 47, 125–127
  precondition, 25, 26, 31–33, 35, 36, 38, 40, 41, 44, 45, 47, 125–127
Assignment
  reasoning rule, 25
  typing rules, 81, 82, 92
Association, 134, 163
Asynchronous message, 105, 113, 115, 118, 119, 121, 122, 126
Atomicity, 148, 187
Attribute, 43, 44, 57, 58, 113, 114, 134, 135, 143, 166–170, 172, 173, 186, 189–192, 194, 196
Automatic static verification, 192, 199

B
Bag ordered, 19
Begin method, 70, 147–151, 160
Behavioral subtyping, 25, 40–41, 46, 47, 105, 119–120, 130, 131, 195, 197
Binding
  dynamic, 1, 9–12, 113, 176
  static, 1, 9–12
Bind method, 140, 141, 145–147, 159, 183
Bounded parametric polymorphism, 158, 160

C
C#, 5, 6, 10–12, 18, 21, 26, 32, 38, 40–45, 47, 49, 50, 53, 58, 77, 78, 91, 99, 102, 103, 110, 113, 141, 143, 152, 157, 163, 201
C++, 172, 176, 179–184
Catch clause, 14, 55–57, 70, 72, 73, 101, 111, 112, 149–151, 156
Class
  Class, 6, 11, 49, 51, 68, 69, 78, 89, 90, 125
  ClassLoader, 52, 56, 57
  constructor, 3, 6, 54, 55, 69
  extent, 49, 56, 166, 167, 172, 173

S. Alagić, *Object-Oriented Technology*, DOI 10.1007/978-3-319-20442-0

Class (*cont.*)
field, 6, 53, 54
file, 49, 51, 52, 56–59, 67
inner, 160
invariant, 31, 125, 131, 192, 195
loader method, 49, 51, 52, 56, 57, 59, 89
Collection
Bag, 38, 168
OrderedBag, 19
OrderedCollection, 16, 17, 21, 86, 90, 99, 103, 109, 111–113, 136, 141, 158, 165, 167, 184–186
OrderedSet, 16
Set, 144, 168
specification, 13, 15, 17
types, 13, 32, 95, 100
Commit method, 155, 160
Committed object state, 190
Common Language Runtime (CLR), 58, 126, 131
Complex object, 71, 72, 74–76, 79, 110, 133, 145, 154, 186, 188, 191, 201
Component, 2–4, 6, 22, 36, 37, 49, 52, 58, 65, 67, 70–72, 75, 82, 124, 145, 152, 154, 155, 157, 169–171, 188–193, 196–198, 201
Concurrent, 148, 187, 199
models, 105–131
object, 105, 120–129, 131
threads, 105–113, 130
ConcurrentObject class, 122, 123
Conditional
reasoning rule, 25
statement, 25, 27, 28, 82
typing rules, 81
Conjunction, 27, 31, 32, 41, 119, 120, 158
Consistency, 148, 165, 187, 189–191, 200
Consistent
database state, 148, 186, 199, 201
object state, 35, 190
Constrained genericity. *See* Bounded parametric polymorphism
Constraints. *See* Assertion
Constructor, 3, 36, 42, 44, 49, 51, 54–55, 84, 85, 87, 89, 106, 110, 113–116, 124, 125, 130, 146, 147, 156, 183, 197
Contract, 25, 33, 42–46
[ContractInvariantMethod], 43, 44
Contracts class, 42, 43, 45, 46
Contravariance, 8, 120
Covariance, 8
Create method, 131, 140, 186

D
Database
constraints, 158–163, 186
object-oriented, 70, 133, 149, 151, 152, 162, 166, 200
relational, 135, 143, 144
schema, 134, 148, 151, 160, 186
type system, 165–186
Database class, 159, 168
DataContext class, 143
DBag class, 140, 144, 146, 147, 181, 183
Db4 Objects, 133, 151–155, 162, 195
DCollection class, 140, 181
DefineClass method, 56, 57
Delete method, 153, 157
Dictionary, 21
Disjunction, 27
Dispatch, 10–12
Distinct clause, 136, 167, 169, 174, 175, 179
Do loop, 29, 30, 47, 83
DSet class, 181
Durability, 148, 187
Dynamic
binding, 1, 9–12, 113, 176
checking, 10, 14, 21, 33, 42, 46, 77, 88, 89, 91–94, 99, 146, 177, 199–200

E
[ElementsRep], 191, 196
Ensures clause, 31, 32, 38
Enumerable interface, 141
Equality, 10, 26
Equals method, 180
Exception, 12, 14, 33, 41, 53–57, 72, 73, 93, 97–101, 110, 112, 127, 131, 138, 149–151, 156, 199
Execute method, 147, 160, 183
Existential quantification, 39, 45, 192
Exists, 36, 73, 75, 93, 106, 134, 137, 138, 157, 171, 193, 194, 196, 197, 199
Exists method, 45
Expose block, 36, 189, 190, 201
Expression, 2, 26–28, 34, 42, 45, 49, 69, 81–84, 86, 88, 89, 91–93, 113, 115–118, 124–126, 128, 129, 131, 138, 142, 146, 158, 166, 167, 171, 173–175, 178, 182
Extended virtual machine, 67, 69
Extends, 3–6, 9, 14, 16–20, 31, 38, 51, 53–55, 57, 69, 71, 76–78, 85, 90, 97, 102, 103, 105–107, 109, 110, 112, 115–121, 123, 124, 127, 129, 131,

135, 138, 144, 156, 158–162, 170, 171, 181, 184–186, 199, 201
Extent, 5, 135, 165–169, 172, 173

F
File system, 59, 75, 147
Final, 6, 11, 51, 53–55, 69, 114, 125–127, 158–160, 162
ForAll method, 45
Foreach statement, 47, 102
Formula, 26, 68, 69, 78, 127, 131
ForName, 52, 78, 89
For statement, 13, 15, 33, 47, 102
Frame constraint, 37, 61, 67, 78

G
Genericity. *See* Parametric polymorphism
GetConstructor, 51, 52
GetDeclaringClass, 53, 54
GetExceptionTypes, 54, 55
GetInterfaces, 52
GetMethod, 51, 52, 69, 70, 92–94, 101, 122, 125, 192, 198
GetModifiers, 52, 54, 55
GetName, 52–55, 93, 135, 139, 146, 147
GetParameterTypes, 54–56, 93, 94
GetReturnType, 54
GetSuperClass, 6, 51
GetType, 53, 55, 69, 92

H
Heap, 57–67, 74–76, 78, 105, 107, 123, 124, 127–129
Hierarchy, 6, 22, 50, 51, 75, 78, 105, 114, 117, 144, 155, 189
Horn clause, 32, 82, 158

I
IEnumerable interface, 141, 144
IEnumerator interface, 141
Implements, 2, 4, 6, 16, 17, 19, 21, 22, 46, 52, 58, 71, 75, 78, 97, 106, 110–112, 114–116, 130, 141, 144, 161, 163, 164
Implication, 10, 11, 21, 25, 27, 31, 33, 50, 75, 95, 98, 99, 101, 102, 105, 117, 138, 146, 158, 182, 184, 187, 188, 190, 191, 197
Inheritance
  diamond, 5
  multiple, 5, 18, 23, 110, 111, 114
  single, 18, 110
Integrity constraint
  general, 155, 165, 186
  key, 160, 186, 187, 191, 193
  referential, 160, 186, 189, 193
Interface, 1–6, 13–19, 22, 31, 32, 45, 46, 52, 57, 71, 74–76, 78, 97, 102, 103, 106, 110, 114–116, 124, 126, 127, 129–131, 135, 137, 138, 140, 141, 144, 145, 147, 152–155, 157, 159–162, 165, 167–172, 174, 178, 180, 181, 184–186, 189
Interpreter, 49, 58, 147
Interpretive technique, 147
Interrupt method, 111
Intersection, 14, 15, 18, 19, 31, 38–40, 178
Invariant
  class, 31, 125, 131, 192, 195, 196
  loop, 25, 28, 33–34, 36, 47
  object, 25, 35–37, 40, 43–45, 189, 190
Invoke
  method, 17, 41, 49, 51, 54–56, 61, 62, 78, 101, 109, 112, 115
  operation specification, 61, 78
  typing rules, 93–94
IObjectContainer class, 152
IsInterrupted method, 112
Isolation, 148, 187, 188
Iterator, 19, 101, 138, 181, 186
  variable, 141

J
Java, 5, 6, 10, 12, 13, 18, 21, 26, 32, 38, 41, 45–47, 49–53, 56–59, 71, 74, 75, 77–79, 81, 84, 85, 87, 88, 91, 93, 95, 97–103, 105, 106, 110–113, 130, 131, 133, 134, 137–141, 144–149, 155, 162, 163, 170, 172, 173, 175–177, 180–184
Java binding (ODMG), 133, 137, 140, 144–149, 172, 175–177, 181–183
Java Core Reflection (JCR), 51, 53, 68, 77, 93, 101
Java OQL, 78, 133, 137–141, 162, 165, 181–184
Java Virtual Machine (JVM), 49, 58, 59, 70, 78, 99, 126, 131

K
Key, 1, 16, 21, 30, 133, 143, 148, 157, 160, 165, 186–188, 191, 193

L
Lambda expression, 45, 142
Language Integrated Query (LINQ), 133, 141–144, 162, 163, 181, 195, 200
List, 37, 42, 118, 136, 154, 157, 168, 180, 184, 191, 194–196, 198
LoadClass method, 56, 57
Loader, 49, 51, 52, 56, 57, 59, 67, 77, 89
Locking, 188
Lookup method, 146, 159
Loop
  Do, 29, 30, 47, 83
  invariant, 25, 28, 29, 33–34, 36, 47
  reasoning rule, 28
  typing rule
    For, 30
    While, 28, 34, 83

M
Match method, 154, 157
Message
  arguments, 174
  asynchronous, 105, 113, 115, 118, 119, 122, 126
  broadcast, 113, 119
  create, 115, 116, 118, 119, 121
  mutator, 113, 115, 125
  receiver, 3
  signature, 178
  sustained, 113, 116, 121
  term, 69
  transient, 113, 116, 121
  twoway, 115, 122, 128
  type, 105, 113, 114, 116, 117
Method
  arguments, 10
  get, 43, 51, 53, 153, 157, 198
  introspection, 6, 11, 54, 81, 92, 93
  invocation, 32, 33, 61, 93, 94, 98
  mutator, 51
  parametric, 18, 22
  pure, 37, 38, 44, 194, 195, 198
  receiver, 11, 54, 191
  result, 170
  set, 43, 53, 55, 153, 192, 198
  static, 12, 142
Mobile object, 123, 130
Model field, 197, 198
Modifies clause, 33, 37, 193, 194, 197
Multiple inheritance, 5, 18, 22, 23, 110, 114
Mutable object state, 190

N
Native query, 157, 164, 195
Negation, 17, 27
NewArray, 64, 78
  operation specification, 64, 65, 78
NewInstance
  method, 51, 54, 55
  operation specification, 60, 78
Nonnull type, 34, 189, 192
NotifyAll method, 107, 109, 130

O
Object
  committed, 190
  concurrent, 105, 120–129
  consistent, 35, 190, 197
  database, 133–164, 194
  identity, 2, 7, 10, 129
  invariant, 25, 35–37, 40, 43–45, 189, 190
  message, 114–116, 118, 121
  mobile, 123, 130
  mutable, 190
  persistent, 49, 50, 71–78, 100, 133, 144, 145, 148, 153, 155
  serialization, 50, 75–77
  service, 120, 121, 123, 124
  state, 2–4, 6, 7, 31, 35–37, 45, 105, 107, 113, 119, 128, 129, 139, 146, 152, 157, 190, 193, 197, 198
  valid, 190, 192
Object class, 50
Object Definition Language (ODL), 133–135, 162, 165–167, 172, 173
ObjectInput class, 75
Object Oriented Database Management Group (ODMG), 70, 133–135, 137–141, 144–149, 151, 152, 158, 159, 162–165, 167–170, 172–177, 179–182, 184, 200, 201
ObjectOutput class, 75
Object Query Language (OQL), 133–135, 137, 138, 140, 142, 143, 146, 163, 165–167, 169, 172–184
Open method, 152
Operation, binary, 63, 64, 83
Operator, 32, 47, 70, 83, 84, 142, 201
OQLQuery class, 140, 183

Order by clause, 142, 165, 167, 179–181
Orthogonal persistence, 50, 71–75, 145
Overload, 10, 26
Overloading, 87, 96, 97, 103, 170
Override, 11, 18, 22, 38, 41, 109–111, 197
Overriding, 10, 11, 86, 109, 197
Owner, 188–191, 196
Ownership model, 191

P
Parameter
  actual, 18, 36, 54, 55, 96, 110, 174, 183
  formal, 3, 87, 96, 98, 102, 118, 140, 146, 183
Parametric polymorphism, 1, 15, 16, 20, 21, 146, 160, 172, 178, 179, 181, 183, 184
  bounded, 158, 160
  class, F-bounded, 16, 19, 22, 184
  interface, 16, 31, 178, 185
  method, 18, 22
  subtype, 7, 9, 12, 14, 15, 146
  universal, 16, 90, 94
Peer, 188–192
[Peer], 189
Persistence
  architecture, 50, 71, 74–75, 78, 145
  orthogonal, 50, 71–75, 145
  by serialization, 99–101
Persistent
  collection, 75, 100, 133, 138, 144, 149–151
  object, 49, 50, 71–78, 100, 133, 144, 145, 148, 153, 155
PJama interface, 159
PJStore interface, 71, 159
Portability, 49
Postcondition, 25, 26, 29–33, 35, 37, 38, 40, 41, 45, 47, 61, 63, 64, 66–70, 78, 118, 120, 125–128, 130, 131, 159, 160, 186, 191, 193–195, 197, 199–201
Precondition, 25, 26, 29–33, 35, 36, 38, 40, 41, 44, 45, 47, 60–64, 66–70, 78, 115, 118, 120, 122, 125–128, 131, 159, 160, 186, 191, 193–195, 197, 199–201
Predicate, 47, 119, 121, 126, 137, 138, 142, 153, 154, 157, 181, 182, 192, 201
Predicate type, 153
Private, 2, 4, 10, 11, 17, 18, 36, 37, 41, 43, 72, 79, 84, 94, 106, 107, 109, 111, 112, 122, 127, 139, 146, 150–152, 156, 157, 189, 191, 192, 196, 198, 200
Property, 12, 16, 43, 45, 47, 74, 109, 148, 157, 174, 176, 185, 187, 198, 201
Protected, 18, 37, 41, 56, 155, 156, 160, 189, 196
Prototype object, 153
Public, 2, 4–6, 9–19, 31, 34–45, 51–58, 68, 69, 71–77, 84, 94–101, 103, 106–112, 138–148, 150–152, 154–162, 175, 176, 189, 191–198, 200
[Pure], 37–40, 44, 195
Pure method, 37, 38, 44, 194, 195, 198
PutField
  method, 62
  operation specification, 62, 63, 78

Q
Query
  comprehension, 142, 163, 200
  by Example, 153, 156
  inner, 136, 137
  lambda, 142, 163, 200
  languages, 133–144, 153, 163, 181
  methods, 138, 139, 154
  native, 157, 164, 195
  nested, 136, 137
  object-oriented, 133, 159
Queryable type, 144
Queue, 21, 22, 110, 123, 124, 127, 128, 130, 196

R
Reachability, 71
ReadObject method, 76, 77, 79, 100
Reasoning rule, 25–30
Receiver object, 3, 10–12, 41, 55, 56, 61, 78, 113–116, 126, 173
Recovery, 187
Redefine
  class invariant, 10, 32, 123, 170
  method, 10, 32, 170
  postcondition, 38
  precondition, 32
  reference, 10, 32, 123, 170
  type, 32, 170
Referential integrity, 160, 186, 189, 191, 193
Reflection, 49–58, 67, 68, 70, 77, 78, 81, 91–102, 105, 116, 124–127, 130, 140, 151
Region, 123, 124
Relation, 71, 117, 143, 170
Relational database, 135, 143, 144

Relationship, 5, 6, 28, 52, 85, 105, 134–136, 138, 139, 148, 149, 159, 165, 170, 176, 188–190, 192
[Rep], 191, 196, 200
Requires clause, 31, 32
Result, 2, 8, 19, 20, 28, 33–35, 37–40, 50, 54, 56, 58, 61–66, 70, 72, 77, 83, 87, 89, 91, 93, 94, 100, 102, 107, 111–113, 115, 117, 118, 121, 122, 126, 128, 129, 136, 138, 142, 146, 147, 150, 153, 154, 158, 167, 169, 170, 174, 175, 178–181, 183, 184, 188, 194, 195, 199
Rollback method, 155
Root
  of persistence, 72, 145, 146, 159
  of type hierarchy, 78
Run method, 111
Runnable interface, 106, 130
Runtime
  checking, 99
  stack, 58, 64, 78, 127, 129

S

Schema (database), 134, 148, 151, 160, 186
Search tree, 21
Select
  clause, 173
  method, 11, 12, 138, 186
  operator, 142
SelectElement method, 138
Self, 3, 42, 181
Semantics, 1, 15, 31, 32, 49, 67, 70, 77, 113, 159, 188
Sentence, 68, 69, 78, 127
Serial execution, 187
Serializable execution, 187, 199
Serializable interface, 71, 75, 110
Serializable object, 71–77, 110
Service object, 120, 121, 123, 124
Set
  method, 192, 197
  ordered, 16, 21
Signature
  constructor, 6, 84
  field, 6, 84
  method, 6, 8, 74, 85, 87, 117, 168–170, 173, 174
Sleep method, 112
Spec#, 26, 32–39, 46, 126, 189, 191, 192, 196, 199
Specification, 3–5, 12, 13, 16, 18, 20, 25–33, 36–39, 45–47, 49, 50, 60, 61, 67, 77, 78, 81, 103, 114, 117, 125, 129, 131, 134–137, 139–144, 153, 158, 160, 165, 166, 168, 169, 173, 178, 182, 189, 191–193, 195–198, 200
  inheritance, 195–198
[SpecPublic], 37, 41, 189, 191, 192, 196, 200
SQL, 135, 136, 141–143, 167, 181
Start method, 106, 160
Statement
  assignment, 26, 28, 29, 82, 85–86, 88, 97
  conditional, 25, 27, 28, 82–83
  loop, 3–35, 47, 81, 83
Static
  method, 12, 142
  structure, 169–170
  typing rules, 81
Subclass, 6–10, 18, 22, 40, 47, 82, 86, 119, 120, 160, 195, 196, 199
Substitution, 7, 15, 21, 69, 89, 96, 103, 118, 119, 174, 178
Subtyping
  behavioral, 25, 40–41, 46, 47, 105, 119–120, 130, 131, 195, 197
  function, 8, 10, 87
  typing rules, 10, 85, 88, 99, 102, 103
Superclass, 5–8, 20, 40, 41, 47, 51, 57, 75, 86, 109, 117, 119, 120, 199
Swizzelling, 75
Synchronization, 105, 107–110, 130
Synchronized
  method, 108–110, 112, 130, 131
  object, 107–109, 111
Synchronize statement, 111, 130

T

Table, 57, 82, 83, 143, 144, 193
Term, 1, 7, 8, 19, 31, 69, 76, 78, 126, 201
Termination
  of a loop, 25, 30
  verification, 34, 47
This, 2, 3, 19, 31, 36, 37, 39, 41, 57, 68, 69, 106, 111, 115, 121, 123, 139, 159, 161, 181, 182, 189, 192, 193, 199
Thread class, 106, 112
Thread object, 106
Transaction
  abort, 70, 71, 148, 150–152, 160, 186, 201
  commit, 155
  execution, 186, 187, 189, 199
  specification, 200
  start, 148
  verification, 190, 191, 200
Transaction class, 160, 168, 189

Transitive, 9, 85, 154
persistence, 154
Transparency, 71, 73–75, 145
Type cast, 13, 15, 73, 77, 79, 88, 91, 99, 100, 146, 150, 151, 163, 169, 176, 177, 182, 183
Type checking. *See also* Typing rules
dynamic, 1, 10, 16, 53, 81, 88–89, 91, 92, 99, 102, 146, 176, 177, 182, 183
static, 16, 81, 91, 92, 95–100, 173, 179–181, 183, 184
Type erasure
dynamic violations, 95
reflection, 81, 101, 102, 151
serializability, 81
static violations, 81
Type signature
class, 51, 67, 78, 84, 89, 90, 95, 103
constructor, 51, 84
field, 51, 67, 78, 84, 90
message, 87
method, 18, 51, 67, 78, 84, 87, 90, 95, 100, 117, 154
Type systems, 6, 11, 32, 34, 51, 54, 55, 81–103, 116, 130, 134, 139, 146, 151, 165–187, 192, 200
object-oriented, 32, 81, 84–88, 102, 173
Type violation, 53, 54, 81, 93, 95, 98, 99
Typing environment, 82, 83, 85, 86, 88–91, 93, 116, 118
Typing rules
arrays, 99
assertions, 116, 118, 119
assignment, 92
behavioral subtyping, 105, 130, 131, 197
blocks, 102
bounded, 90
class, 173
class extent, 173
conditional, 81
constructor, 84, 116–119
Do loop, 83
dynamic, 21, 81, 92, 99
expression, 81–84, 92, 116–118
extent, 173
field access, 81
field update, 92
inheritance, 7, 81, 197
interface, 103
JavaOQL, 181–184
messages, 81, 103, 105, 116–119
method introspection, 81
method invocation, 81, 103
methods, 7, 20, 81, 99, 102, 103, 116–119, 183
OQL queries, 173, 183
with order by clause, 165
overloading, 87, 96, 97
overriding, 86
parametric types, 81
records, 82, 83
reflection, 81, 91–94, 116
static, 81
structures, 102
subsumption, 86, 88
subtyping, 10, 85, 88, 99, 102, 103
universal, 90
While loop, 83

U

Union, 14, 15, 18, 19, 31, 38, 39, 118, 178
Unique, 2, 42, 114, 191, 193, 194, 200
Universal quantification, 45, 178
Update
field, 55
transaction, 96, 98, 99, 103

V

Valid object state, 190, 192
Variable
bound, 119, 120, 126, 127, 174, 183
control, 13, 33
free, 68, 69, 120, 126, 127
Variant function, 34, 35
Verification
dynamic, 33, 46, 199
static, 33, 42, 46, 192, 199
Virtual
machine, 49, 50, 58–67, 77, 78, 99, 105, 122–124, 126–129, 131
method, 11, 49–58, 61, 62, 67–70, 72–79, 99, 126, 128, 131
platform, 1, 49–79

W

Wait method, 107, 112, 130, 131
Where
clause, 137
operator, 142
While loop, 28–29, 83
WriteObject method, 76, 77, 79, 100

Y

Yield method, 112