Chapter 1
Total Programming

The definition of all the methods of a class must be complete. Each method must deal with all possible conditions under which clients can invoke them. As an example, consider a method for computing the greatest common divisor of two integer numbers. That method must not only return a correct result in case clients supply two positive numbers. The method must also deal with attempts to compute the greatest common divisor of negative numbers and of zero's. In this chapter, we introduce the paradigm of *total programming* as one of three alternatives to deal with such exceptional cases. We discuss the other two alternatives — *nominal programming* and *defensive programming* — in the following chapters.

Methods developed with the paradigm of total programming — *total methods* for short — always end in a normal way. A total method for computing the greatest common divisor might first of all work with the absolute values of the supplied numbers. Moreover, in case at least one of the supplied numbers is zero, it might return the value of the other number. Total methods are similar to total functions in mathematics. Indeed, a total function in mathematics is well-defined for all possible input values. Hence the term *total programming* for this approach to deal with exceptional cases.

This chapter also studies the successive steps in the overall definition of a class. The development of a class starts with a specification step, and proceeds with a representation step and an implementation step, and finishes with a verification step. The *specification* of a class results in its interface. It lists all the methods that can be invoked against the class itself or against individual objects of it. The class interface describes *what* clients can expect from the class and its objects. In establishing a *representation* for a class, the

focus is on how to store information. This step results in declarations of class variables and of instance variables. Class variables store information at the level of the class itself, whereas instance variables store information at the level of individual objects. In the *implementation* of a class, a body complements the specification of each method. That body is a sequence of statements. It must achieve the effects as stated in the specification of the method. Finally, the *verification* step results in a test suite for the class. Such a suite is a collection of tests for all the methods. Each test verifies whether a method behaves as expected in a particular case.

The order of steps in the development of a class is merely a conceptual one. We do not intend to suggest that we cannot proceed to the next step as long as we have not completely finished all preceding steps. Obviously, we can not work out the implementation of a method if we do not know what that method is supposed to do. Conceptually, the specification of a class must therefore precede its implementation. In practice, however, we do not work out the entire specification of a class, before we start working on its representation, on its implementation and on its verification. Instead, we rather specify a little, represent a little, implement a little, and verify a little. After several such iterations, we end up with a complete definition of the class. We may even reverse the order of some activities. As an example, once we have completed the specification of some methods of a class, we may already work out tests for those methods. In that strategy, verification precedes representation and implementation. In this chapter, we only want to describe what the different steps in the development of a class are all about, not the order in which we must perform them. Software development processes such as the *Unified Process* and *Extreme Programming* tackle that topic.

1.1 Class Interface

If we want to split software systems in a series of loosely coupled classes, we need a way to communicate what clients can expect from those classes. That information must have enough detail, yet not too much. It must also be precise and unambiguous. Computer languages — and imperative languages in particular – are not ideal to explain the semantics of software systems to human beings. Indeed, the implementation of a class written in some programming language is above all intended for the computer. Moreover, the implementation of a class often includes information that is irrelevant for its clients. The object oriented paradigm therefore distinguishes between the interface of a class and its

implementation. The *class interface* tells clients *what* they can expect from a class and its objects. The *implementation of a class* instructs the computer *how* to achieve the specified effects. In the literature, the interface of a class is also called the *specification* of that class.

1.1.1 Class heading

In Java, the definition of a class consists of a heading and a body. The body of a class defines properties and methods that either apply to the class itself or to individual objects. The *body* of a Java class is enclosed in braces ({…}). The class *heading* at least introduces a name for the class. A documentation comment typically complements the heading of a class. In this book, we use it to enumerate properties ascribed to the class itself and to its objects. We also include some bookkeeping information, such as a version number and the names of the authors of the class. Example 1 below lists the heading of a class of bank accounts. In Java, we must store the definition of a class in a file named after that class and complemented with a suffix `.java`. We therefore need a file named `BankAccount.java` to store the definition of the class of bank accounts.

```
/**
 * A class of bank accounts involving a bank code, a number,
 * a credit limit, a balance limit, a balance and a blocking
 * facility.
 *
 * @version 2.0
 * @author  Eric Steegmans
 */
class BankAccount {
  …
}
```

Example 1: Heading of a class of bank accounts.

Notational conventions for class names

We use identifiers to name ingredients of object oriented programs. Classes, methods and variables are examples of such ingredients. In Java, each *identifier* is a sequence of letters — both uppercase and lowercase —, digits, underscores (_) and dollar signs ($). Identifiers in Java may not start with a digit and are case sensitive. Java thus distinguishes between uppercase letters and lowercase letters. The identifier `abc` is not the same as the identifier `ABC`. Java basically supports the complete Unicode symbol set — http://www.unicode.org. This means that we can also use symbols such as ç, é and ñ in identifiers. In practice, not all programming environments fully support Unicode at all

possible points in Java programs. Therefore, we recommend not to use characters outside the ASCII-set in identifiers.

The name of a class must reflect as good as possible the nature of its objects. We further recommend to use a consistent style in *naming classes*. First of all, we suggest to use singular nouns as names for classes. In Example 1 above, we therefore use `BankAccount` as the name of the class – not `BankAccounts`. As another example, a software system for a public library may include a class reflecting loans of books. The name of that class will be the noun `Loan` – not the verb `Borrow`. We may add adjectives to the noun as in `GoldCard` or in `PaperBackBook`. We call such identifiers *noun phrases*, because their core is still a noun. No spaces, underscores, or other symbols separate successive words in noun phrases. Java further suggests — but does not impose — to use *pascal casing* for class names. In that style, each internal word starts with a capital letter. This explains the spelling `BankAccount` for the name of the class of bank accounts.

Coding Rule 1: Use singular nouns or noun phrases as names for classes. Use pascal casing for their spelling.

Throughout this book, we introduce similar rules for naming methods, variables and other ingredients of object oriented programs. Such *notational conventions* make it easier to read and understand object oriented programs. If identifiers for different kinds of ingredients adopt different notational conventions, we can recognize them merely from their spelling. Moreover, complex software systems are often developed by a team of programmers. If such a team agrees on a common set of notational conventions, software developed by that team will have a common look and feel. Members of the team then have less problems in getting familiar with software developed by other members of the team.

Documentation comments for classes

Java has no strong concepts for specifying the semantics of classes and their features. The language only offers plain *comments* to document class interfaces. Java distinguishes between implementation comments and documentation comments. *Implementation comments* serve to further explain internal aspects of classes and their features. Multi-line implementation comments are enclosed between `/* … */`. A single-line implementation comment starts with two consecutive slashes (`//`) and proceeds up

to the end of the line. *Documentation comments* are enclosed between `/** … */`. Each new line in a documentation comment typically starts with the symbol `*`. These extra stars only serve to turn documentation comments more visible in the entire definition of a class. In Java, a documentation comment immediately precedes the element to which it applies.

Documentation comments must include all the information needed to deal with classes and their objects. A *class documentation comment* briefly introduces all the characteristics ascribed to the objects of the class, as well as characteristics ascribed to the class itself. The class documentation comment in Example 1 above introduces the bank code, the number, the credit limit, the balance limit, the balance, and a blocking facility as characteristics that apply to individual bank accounts or to the class of bank accounts itself. The body of the class then offers methods for manipulating these characteristics, and variables to store their current value.

Coding Advice 1: A class documentation comment starts with an enumeration of all the characteristics ascribed to the class itself, as well as all the characteristics ascribed to individual objects of that class.

Java offers so-called *documentation tags* to structure the documentation of classes and their features. Tags can only occur at the start of each line in a documentation comment. In other words, tags must immediately follow the symbol `*` delineating the beginning of a new line of documentation. A documentation tag starts with the symbol `@` and is followed by an identifier clarifying its purpose. Some tags such as `@param` and `@author` are predefined. Additional tags can be defined. We recommend to add some bookkeeping information to the class documentation comment, such as a version number and information about the authors of the class. In Example 1 above, we use the predefined tags `@version` and `@author` to introduce information about the version, respectively about the author of the class of bank accounts.

Coding Advice 2: Include at least a version number and a list of author names in the class documentation comment.

The Java Development Kit (JDK) includes a documentation generation tool called *javadoc*. That tool processes definitions of classes and recognizes documentation comments. It produces one or more documentation files in HTML format. The generated files include all the documentation comments complemented with the signature of the

element to which they apply. Javadoc recognizes documentation tags and uses them in formatting the generated HTML files. Self-defined tags in documentation comments must be backed by so-called doclets. They tell javadoc how to process these tags in generated documentation files.

1.1.2 Methods

Java, forces us to define each method as an ingredient of some class. Pure object oriented programming languages — such as Java and C# — do not have a notion of global routines that operate on some global data. These constructs are typical for procedural languages such as Pascal and C. Hybrid programming languages also support the definition of global routines and global data. Such languages mix aspects of procedural programming with aspects of object oriented programming. An example of such a language is C++. In a pure object oriented programming language each method either applies to individual objects of its class or to the class itself. In Java, methods that apply to individual objects are called *instance methods*. In this book, methods that apply to classes rather than to individual objects are called *class methods*.

The definition of a method in Java consists of a heading and a body. The *body* of a method is enclosed in braces ({…}). It lists the instructions that are executed each time the method is invoked. The *heading* of a method instructs clients how to invoke that method. The heading of a method is also called the method's *signature*. Example 2 below lists the signature of methods defined in the body of the class of bank accounts.

- The heading of a method first of all introduces the method's name. The class of bank accounts offers methods named `getBalance`, `deposit`, `setCreditLimit`, `isBlocked`, … The first three methods in Example 2 are constructors. In Java, constructors are always named after their class.
- The heading of a method also specifies the formal arguments of the method and its return type. Constructors also have a list of formal arguments. However, they never return a result to their caller. The method `transferTo` in Example 2 involves the amount of money to be transferred and the destination bank account as formal arguments; this method does not return a result. The method `getBalance` returns an amount of money in the form of a long integer; this method does not involve any formal arguments.

- The signature of a method may start with some modifiers. One of these modifiers establishes *access rights* for methods. In Example 2, most methods are qualified **public**. Anyone who has access to a class, has access to all the public elements of that class. The method `setBalance` in the class of bank accounts is qualified **private**. Private elements are only accessible in the body of their class.

```
class BankAccount {

  public BankAccount
      (int number, long balance, boolean isBlocked) { … }
  public BankAccount(int number, long balance) { … }
  public BankAccount(int number) { … }

  public int getNumber() { … }

  public static int getBankCode() { … }

  public long getBalance() { … }
  public boolean hasHigherBalanceThan(long amount) { … }
  public boolean hasHigherBalanceThan(BankAccount other) { …
}
  public void deposit(long amount) { … }
  public void withdraw(long amount) { … }
  public void transferTo
      (long amount, BankAccount destination) { … }
  private void setBalance(long balance) { … }

  public static long getCreditLimit() { … }
  public static void setCreditLimit(long creditLimit) { … }

  public static long getBalanceLimit() { … }

  public boolean isBlocked() { … }
  public void setBlocked(boolean flag) { … }
  public void block() { … }
  public void unblock() { … }

}
```

Example 2: Signature of methods offered by a class of bank accounts.

In section 1.1.3, we discuss how to complement the heading of a method with a documentation comment. In this way, the definition of a method has the same structure as the definition of a class. They both involve a heading complemented with a documentation comment and a body. A method documentation comment describes what can be expected from that method whenever clients invoke it. Method documentation comments thus describe the semantics of methods.

Notational conventions for method names

The *name of a method* must reflect as good as possible its intention. Just by reading a method's name, clients must already have a good idea of what they can expect from it. As for class names, we recommend to use a consistent style in naming methods. First of all, we recommend to use a verb in active tense as the core of a method name. In Example 2 above, we use `deposit` as the name of the method for depositing some amount of money to a bank account. In the same way, we use `block` - not `beBlocked` - as the name of the method for blocking a bank account. We may add nouns, adjectives and adverbs to the verb, as in `transferTo`, `setBlocked` and `hasHigherBalanceThan`. Other possible names for the latter method are `exceedsInBalance` or `exceedsBalanceOf`. We refer to such identifiers as *verb phrases* because their core is always a verb. As for class names, we do not use spaces, underscores, or other symbols to separate successive words in verb phrases.

In Java, the name of a method is an identifier. If it is not a constructor, the name of a method must differ from the name of its class. We have discussed the rules for spelling identifiers in Java on page 7 in the context of class names. Java further suggests to use *camel casing* for method names. In this spelling style, each internal word starts with a capital letter, except for the first word. The instance method `transferTo` in the class of bank accounts illustrates camel casing. The first word `transfer` is written entirely in small letters. The second word `To` starts with a capital letter.

Coding Rule 2: Use verbs or verb phrases in active tense as names for methods. Use camel casing for their spelling.

In the end, we choose the name of a method such that invocations come close to sentences in a natural language. Some method invocations then read as commands; others read as questions. As an example, the Java statement `myAccount.transferTo-(100,yourAccount)` is close to the English sentence "Transfer an amount of 100 from my account to your account!". As another example, the Java expression `myAccount.-hasHigherBalanceThan(yourAccount)` is close to the English sentence "Has my account a higher balance than your account?".

Coding Advice 3: Choose method names in such a way that their invocations read as commands or questions in a natural language.

Grouping of methods

Java does not impose an order on the definition of the elements of a class. We recommend to start the definition of a class with its constructors. We further recommend to group elements of a class as much as possible according to the property to which they apply. In Example 2 above, methods that apply to the number of a bank account, respectively to the bank code follow the group of constructor methods. These groups just consist of a single method. A group of methods related to the balance of a bank account follows next. That group involves several methods. Next follows a pair of methods for manipulating the credit limit. The definition of the class proceeds with a single method for querying the balance limit, and ends with a group of methods related to the facility to block bank accounts. Within a single group, we order the elements according to their access rights. Each group of elements thus starts with the public elements, followed by its private elements. We use separate groups to introduce elements of a class that apply to several properties at the same time. Constructors are a typical example of such methods.

Coding Advice 4: Start the definition of a class with a definition of all its constructors.

Coding Advice 5: Group all elements of a class related to a single property. Start a group with its public elements; finish with its private elements.

1.1.2.1 Getters and setters

Since a few years, Java programmers have taken over some additional notational conventions that originate from Java Beans and Java Enterprise Beans. Broadly speaking, *Java beans* are reusable components. Programming environments offer facilities to manipulate beans in a visual way. *Java enterprise beans* are server-side components that encapsulate business logic. Both types of beans impose the definition of a *setter* and a *getter* for each characteristic or property ascribed to them. More and more people recommend to apply that same practice in developing *POJO's* – Plain Old Java Objects.

In this chapter, we only study *single-valued properties* or characteristics. A property ascribed to an object or to a class is single-valued if that property can have at most one value at each instance of time. As an example, the balance of a bank account is single-valued, because a single bank account cannot have several amounts of money as its balance at the same time. In Chapter 2, we discuss multi-valued properties. Such a property can have several values at the same time. As an example, a single person can

have several phone numbers. The property phone number ascribed to objects of a class of persons is therefore a multi-valued property.

The setter for a single-valued property α serves to change that property to a given value. The setter is a method with signature **void** `setα(T α)`. The getter for a single-valued property α serves to return the current value of that property. The getter is a method with signature `T getα()`. In Example 2 above, we introduce the setter **private** `void setBalance(`**long** `balance)` and the getter **public long** `getBalance()`. The setter serves to register a given amount of money as the new balance of a bank account. The getter serves to query the balance of a bank account. Notice that we do not offer `setBalance` as a public method. We do not want ordinary clients to set the balance of a bank account to some arbitrary value. We only want ordinary clients of a class to change the balance of a bank account by withdrawing and depositing money. In Example 2 above, we introduce in a similar way the methods `getCreditLimit` and `setCreditLimit` to manipulate the credit limit. The balance is a single-valued property ascribed to individual bank accounts. The credit limit, on the other hand, is a single-valued property shared by all bank accounts.

Coding Rule 3: Introduce a getter and a setter for each single-valued, mutable property. Introduce a getter for each single-valued, immutable property.

For *immutable properties* we only introduce a getter. A property is immutable if that property cannot change once its initial value has been set. In Example 2 above the number of a bank account is an example of an immutable property. Once a bank account has been created, its number can no longer change. We therefore only introduce a method `getNumber`, and not a method `setNumber`. The balance limit is another example of an immutable property. Once the balance limit is set, it can no longer change. Because the balance limit is ascribed to the class of bank accounts rather than to individual accounts, its initial value is set as soon as the class of bank accounts comes into play. We discuss the initialization of instance properties and of class properties in more detail on pages 67 and 69.

For single-valued properties of type boolean, we define inspectors of the form `isα()` or `hasα()`, instead of inspectors of the form `getα()`. In the definition of the class of bank accounts above, we introduce the inspector `isBlocked` reflecting whether or not a bank account is currently blocked. Other examples of such inspectors are `isReadable` for

objects of a class of files, and `hasChildren` for objects of a class of persons. Methods for setting boolean properties may still follow the general conventions, as illustrated by the method `setBlocked` in the class of bank accounts. An alternative is to introduce one method for setting the property true and another method for turning the property false, as illustrated by the methods `block` and `unblock`.

1.1.2.2 Mutators and inspectors

The paradigm of object oriented programming dictates to invoke methods against an object. In this book, we call that object the prime object of that invocation. As an example, we must invoke the method `getBalance` against an object of the class of bank accounts. The method then returns the balance of that bank account. As another example, we must invoke the method `getCreditLimit` against the class of bank accounts itself. That method then returns the credit limit shared by all bank accounts.

The methods offered by a class can serve different purposes. In this book, we distinguish between *constructors*, *destructors*, *mutators*, and *inspectors*. Constructors serve to initialize their prime object. Constructors may also change the state of other objects closely related to their prime object. We discuss constructors in more detail in section 1.1.2.6 on page 25. Destructors first of all serve to destroy their prime object. A destructor may also destroy or change the state of other objects directly or indirectly associated with its prime object. We introduce destructors in section 5.5 on page 366. Inspectors serve to return information about the state of some objects. The prime object must play a crucial role in computing that information. Finally, mutators serve to change the state of some objects. A mutator either changes its prime object, or uses its prime object in computing the actual objects to be changed. We discuss the definition of inspectors and mutators in more detail below.

In this book, we try to avoid the definition of methods that combine the semantics of mutators and inspectors. In other words, we do not define methods that change the state of some of the objects involved in it, and that at the same time return information about the state of those objects.

- A method for withdrawing some amount of money from a bank account does not return an indication reflecting whether or not the transaction was successful. The same is true for a method defined to remove an element from a collection, which does not return an indication whether or not the element was effectively removed

from the collection. Object oriented programming offers preconditions and exceptions as better alternatives to signal such exceptional cases.

- A method for withdrawing some amount of money from a bank account does not return the new balance of the bank account against which it is invoked. In most cases, clients of the method do not need that information. If they need to inspect the new balance, they can invoke the inspector `getBalance` against the bank account immediately after the withdrawal.

We believe that classes involving methods combining aspects of mutation with aspects of inspection are more difficult to understand. Moreover, if we do not allow mutators to return a result, we avoid the overhead of computing information that is most often simply ignored by their caller.

Coding Principle 1: Do not define a method that at the same time changes the state of some of the objects involved in it, and returns a result.

Java itself does not distinguish between mutators and inspectors. In fact, predefined classes in the Java Application Program Interface (API) regularly introduce methods that combine aspects of mutation with aspects of inspection. In most cases, there is no problem to reduce these methods to simple mutators. However, Java's concepts are not expressive enough to always avoid definitions of mutators that also return a result, respectively of inspectors that also change the state of some objects. Only in those rare cases that an elegant structure in Java is out of reach, we will set aside the mutator-inspector principle.

Inspectors

An *inspector* returns information about the state of the objects involved in it. An inspector never changes the observable state of any object. In Example 2 on page 11, the methods `getNumber`, `getBalance`, `hasHigherBalanceThan`, `isBlocked`, `getBankCode`, `getCreditLimit`, and `getBalanceLimit` are all inspectors. The first four methods return information concerning the bank account against which they are invoked. The inspectors `getBankCode`, `getCreditLimit`, and `getBalanceLimit` return information shared by all bank accounts. Inspectors may also inspect other objects in calculating their result. In Example 2, the inspector `hasHigherBalanceThan-(BankAccount)` compares the balance of its prime bank account with the balance of the bank account supplied as an explicit argument.

Coding Rule 4: Inspectors never change the observable state of any object.

In Java, the type of the *result* of a method is specified before the name of that method. The result type, also referred to as the *return type*, follows modifiers such as access modifiers that apply to the method. In Java, the information returned by an inspector can be a value of a primitive type or it can be an object of a class. A primitive type defines a set of values complemented with a series of operators involving such values. We discuss primitive types in Java in section 1.2.1. The inspector `getBalance` in the definition of the class of bank accounts specifies the primitive type **`long`** as its result type. That inspector therefore returns a long integer number as the balance of the bank account against which it is invoked. The inspector `isBlocked` returns a value of the primitive type **`boolean`**. So far, the class of bank accounts does not offer inspectors returning an object of a class. In Chapter 5, we extend the class of bank accounts with characteristics such as a holder and a bank card. The class then offers among others an inspector `getHolder` returning the holder of the bank account against which it is invoked. That inspector has the class `Person` as its result type.

Mutators

A *mutator* changes the state of some of the objects involved in it. A mutator never returns a result. Because we also consider classes to be objects of their own, mutators can also change information stored at the level of classes. In the definition of the class of bank accounts on page 11, the methods `deposit`, `withdraw`, `transferTo`, `setBalance`, `setBlocked`, `block`, `unblock`, and `setCreditLimit` are all mutators. Except for the method `setCreditLimit`, each of these methods at least changes the state of the bank account against which they are invoked. The mutator `setCreditLimit` changes information common to all bank accounts. Mutators may also change the state of other objects. As an example, the mutator `transferTo` in Example 2 not only changes the state of the bank account against which it is invoked. It also changes the state of the bank account that is available via the formal argument `destination`.

Coding Rule 5: Mutators never return a result.

In Java, methods use the keyword **`void`** to indicate that they do not return a result. In the definition of the class of bank accounts, the methods `withdraw`, `deposit`,

`transferTo`, `setBalance`, `setCreditLimit`, `setBlocked`, `block` and `unblock` all have **`void`** as their result type.

In Coding Advice 5 on page 13, we recommended to group all elements related to a single property. Within a group, public elements precede private elements. Because we distinguish between mutators and inspectors, we further strengthen this advice. In a group of elements with the same access right, inspectors for examining the property at hand precede mutators for changing that property. In Example 2, the inspectors `getBalance` and `hasHigherBalanceThan` for examining the balance of a bank account, precede the mutators `withdraw`, `deposit` and `transferTo` for changing that property.

Coding Advice 6: In a group of elements related to a single property, inspectors precede mutators with the same access right.

1.1.2.3 Instance methods and class methods

The methods of a class can have different targets. Most object oriented programming languages distinguish between instance methods and class methods. *Instance methods* apply to individual objects of their class. Existing objects of a class are also called *instances* of that class, hence the term *instance methods*. *Class methods* on the other hand apply to their class rather than to individual objects of that class.

Instance methods

In the definition of the class of bank accounts on page 11, the methods `getNumber`, `getBalance`, `hasHigherBalanceThan`, `deposit`, `withdraw`, `transferTo`, `setBalance`, `isBlocked`, `setBlocked`, `block` and `unblock` are all instance methods. In Java, a method is an instance method if that method is not explicitly qualified **`static`** in its heading. An instance method always has an *implicit argument* that references an effective object of its class. The instance method `deposit` thus has a bank account as its implicit argument. For that method, that bank account acts as the account to which the given amount of money must be deposited. As another example, the instance method `getBalance` returns the balance of the bank account implicitly involved in it.

Instance methods are always invoked against an object of their class. We refer to that object as the *prime object* of that method invocation. The prime object of an instance method invocation is bound to the implicit argument of that method. In other words, each time an instance method is invoked against an object, the implicit argument references the

prime object against which the method is invoked. Because of their tight bond, we often refer to the implicit argument of an instance method as its prime object.

The prime object of an instance method must have a crucial role in the semantics of that method. In Chapter 5, we will suggest a rule that instance methods must manipulate an object in the *span* of their prime object. The span of an object includes among others all objects that are reachable following references that start at that object. In the first part of this book, however, we do not yet deal with references between objects. At this point, the rule therefore simplifies to the requirement that each instance method must manipulate its prime object. Instance mutators must therefore change the state of their prime object. Instance inspectors must consult the state of their prime object in computing their result. Obviously, instance methods are free to manipulate other objects that are in reach via explicit arguments or via other channels.

A method for withdrawing some amount of money from a bank account is thus not defined as a method in which the target account is offered as an explicit argument. That method could then be an instance method or a class method of the class of bank accounts; it could also be a method of any other class. If defined as an instance method, it would have as signature `public void withdraw(BankAccount target, long amount)`. Obviously, by supplying the bank account as an explicit argument, the prime object of this method has no role to play.

The notion of a prime object for instance methods is typical for object oriented programming languages. The concept ensures that there is at least some reason to define a method as an instance method of its class. If the prime object has no role to play in an instance method, we suggest to restructure the method and to select another object as its target. However, this principle does not yield a unique class in which to define an instance method. Indeed, in case a method involves several objects, we must select one of them rather arbitrarily as the prime object of that method. Other objects then become explicit arguments. The instance methods `transferTo` and `hasHigherBalanceThan` in Example 2 on page 11 illustrate this problem to some extent. Both methods involve two bank accounts. We have chosen one of them rather arbitrarily as their prime object. We could, however, just as well define a method with signature `public void transferFrom(long amount, BankAccount source)` to transfer an amount of money between accounts.

As another example, consider the definition of a method for registering that some member of a public library has borrowed a book. We can define that method as the instance method **public void** `borrowBy(LibraryMember member)` of the class of library books. The borrowing member then becomes an explicit argument of that method. We can just as well define that method as the instance method **public void** `borrow(LibraryBook book)` of the class of library members. In that case, the borrowed book becomes an explicit argument. Finally, we can also offer the method as the constructor **public** `Loan(LibraryBook book, LibraryMember member)` of a class of loans, involving both the book and the member as explicit arguments.

Class methods

In Java, *class methods* are qualified **static** in their heading. Class methods are defined as an integral part of the class to which they apply. In Example 2 on page 11, the methods `getBankCode`, `setCreditLimit`, `getCreditLimit` and `getBalanceLimit` are all class methods. The second method is a mutator. The other methods are inspectors. By definition, all these methods apply to the class of bank accounts. In other words, the class of bank accounts is the prime object in each invocation of these methods. Contrary to instance methods, class methods do not have an implicit argument referencing their prime object. Obviously, there is no need for such an implicit argument, because a class method is always invoked against its own class.

Java uses a static qualification for class methods to emphasize that it imposes static binding on their execution. In the third part of this book, we explain the notions of static binding and of dynamic binding. The latter type of binding is imposed on the execution of instance methods. Because of their static qualification, most Java handbooks refer to class methods as *static methods*. In this book, we prefer the term "class method", because classes are their prime object. In the same way, the term "instance method" reflects that instances of classes are their prime objects.

Instance methods versus class methods

In general, we use instance methods if at least one object is involved in an operation. That object then acts as the prime object for that instance method. This explains why the method for withdrawing some amount of money from a bank account is an instance method. The bank account serves as the prime object for that method. Instance methods have some major advantages over class methods. In the third part of this book, we

explain that several versions of an instance method may exist throughout a hierarchy of classes. Each time we invoke an instance method against some object, the proper version is selected dynamically. This mechanism is known as *dynamic binding*. Class methods, on the other hand, cannot have several versions throughout a hierarchy. Invocations of class methods are handled in a static way. Dynamic binding of instance methods is one of the most powerful mechanisms in object oriented programming languages. It supports adaptability — one of the quality factors for software systems — in the sense that the same instance method can be tuned towards the particularities of different kinds objects.

Coding Principle 2: Prefer instance methods over static methods.

Static methods are close to procedures and functions in procedural languages such as Pascal and C. They all use static binding, and information stored at the level of a class corresponds to some extent with global variables in procedural languages. Programmers new to object oriented programming are tempted to use lots of static methods. This is especially so if they are switching from procedural programming to object oriented programming. As an example, the method for withdrawing money from a bank account can be defined as a static method. Its signature then becomes `public static void withdraw(BankAccount account, long amount)`. Notice that the bank account becomes another explicit argument of the static method. A procedure or a function in a traditional, procedural language would also include the bank account in its argument list.

We only see two cases in which we can use static methods. First of all, Java forces us to use a static method if no objects at all are involved in the operation. Indeed, Java distinguishes between primitive types and classes. Primitive types define sets of values of different kinds, complemented with operators for manipulating these values. Examples of primitive values are integer numbers, floating point numbers, characters and booleans. Java does not treat values of primitive types as objects. Whenever the definition of a method only involves a manipulation of primitive values, we must define that method as a static method. Otherwise, we break the principle introduced on page 19 stating that instance methods must manipulate at least one object in the span of their prime object.

The predefined class `Math` in the Java Application Interface (API) offers lots of static methods for manipulating values of primitive types. The class serves to complement language-defined operators for manipulating numeric values with all kinds of other operations involving such values. As an example, the class offers the method **public**

`static int` `abs(`**`int`** `a)`, returning the absolute value of the given integer `a`. Because this method only involves a single integer value, it is best defined as a static method. In fact, because no objects of the class `Math` can be constructed, the method must be defined as a static method in this particular case.

Coding Rule 6: Use a static method if only primitive values are involved in it.

We also use static methods to manipulate properties that are shared by all the objects of a class. In Example 2 on page 11, we ascribe six properties to bank accounts: a number, a bank code, a balance, a credit limit, a balance limit and a blocked state. The number, the balance and the blocked state may differ from bank account to bank account. However, in our example we assume that all bank accounts share the same bank code, the same credit limit and the same balance limit. The same bank code thus applies to all bank accounts, meaning that they all belong to the same bank. In the same way, the same credit limit and the same balance limit applies to all bank accounts. We further assume that these decisions are final. In other words, we do not expect the bank to change its policy concerning credit limits and balance limits in the future. We also assume that it must not be possible for our class of bank accounts to group bank accounts belonging to different banks.

In Java, properties that must at all times be shared by all the objects of a class are modeled as *static properties*. The property then becomes a property ascribed to the class itself, rather than to individual objects of that class. In this book, we prefer to call such properties *class properties*, for the same reasons we prefer the term *class methods* over *static methods*. In the same way, we refer to properties ascribed to individual objects as *instance properties*. Obviously, we use class methods to manipulate class properties. In the class of bank accounts, the methods `getBankCode`, `getCreditLimit`, `setCreditLimit` and `getBalanceLimit` are all class methods. They reflect our decision that the bank code, the credit limit and the balance limit are shared by all bank accounts, and that we do not expect these properties to become instance properties in the future.

Coding Rule 7: Use class methods if their behavior must at all times be the same for all the objects of a class. Use class methods among others to manipulate class properties.

It is an illusion to think that everything in a software system is easily changed. People often have that impression because software is soft – just a stream of bits and bytes loaded into some machine. As for all engineering products, the more adaptable a software system must be, the more complex – and thus the more pricey – its development will be. As an example, programmers often think that it is easy to change a class method into an instance method, and vice versa. This is definitely not the case. It seems that we only need to remove the keyword **static** from the heading of a class method. By definition, the former class method then becomes an instance method in Java. However, if we change a class method into an instance method, or vice versa, all the clients of the class must change their code as well. Everywhere clients did invoke a former class method against its class, they must now invoke that method against an object of that class. For heavily used methods, this can be a lot of work.

The signatures of the methods offered by a class are part of the *class interface*. Further on in this book, we will state that the class interface is a kind of contract between the clients of that class and its developers. In developing a class, we must consciously decide which aspects of that class must be adaptable, and which aspects must not be adaptable. The class contract must reflect those decisions. If we decide that a certain aspect of a class must not be adaptable, we can use that fact in writing down the class contract. The class of bank accounts on page 11 offers the credit limit as a class property complemented with class methods. This reflects our decision that all bank accounts will always have the same credit limit.

If a certain aspect of a class must be adaptable, we must anticipate future changes in the initial contract for that class. In such cases, we must write the class contract in the most general way. Let's assume for a moment that it must be possible to have different credit limits for different bank accounts in the long run. In that case, we must define the property from the very start as an instance property. In early versions of the class, the instance inspector `getCreditLimit` may still return the same credit limit for all bank accounts. However, we do not communicate that fact to our clients, such that they cannot use that knowledge in building their own code. In later versions, we can change the implementation of the inspector — and only its implementation — to return credit limits that differ from bank account to bank account.

1.1.2.4 Explicit arguments

In addition to their prime object, methods generally need more information to achieve their effects. A formal argument list therefore complements the signature of a method. Each *formal argument* in the list describes some piece of information that clients must supply each time they invoke the method. A formal argument specifies the type of the information to be passed, and a name that can be used to access that information in the definition of the method. As an example, the formal argument list of the instance method `withdraw` on page 11 reveals that clients must supply an amount of money. That amount must be a value of the primitive type **`long`**. In the definition of the method, that amount is known as `amount`.

In Java, the formal argument list of a method immediately follows the method's name in its signature. Comma's separate successive formal arguments. Parentheses enclose the entire formal argument list. The formal argument list of a method may be empty, as illustrated by the inspector `getBalance` in the class of bank accounts on page 11. The *name of a formal argument* is an identifier. The name of a formal argument is a noun or a noun phrase reflecting as good as possible the role of that argument in the method. Formal argument names and names of local variables further follow the notational conventions for method names, as described on page 12. The name of a formal argument is thus written in camel case. The first internal word is written in lower case. Subsequent internal words, if any, start with a capital letter.

Coding Rule 8: Use singular nouns or noun phrases as names for formal arguments and local variables . Use camel casing for their spelling.

As for the result type of a method, the *type of a formal argument* is either a primitive type or a class type. A formal argument of a primitive type specifies that clients must supply a value of the stated type. A formal argument of class type specifies that clients must supply an object of the stated class. In the definition of the class of bank accounts on page 11, the mutator `transferTo` specifies that clients must first of all pass a value of the primitive type **`long`**. That value acts as the amount of money to be transferred. The second formal argument in the mutator `transferTo` specifies that clients must also supply an object of the class of bank accounts each time they invoke the method. That bank account acts as the destination of the transfer.

1.1.2.5 Overloading

The methods of a class must not all have different names. Java fully supports the *overloading* of methods, meaning that several methods in the same class can have the same name. In Example 2 above, we introduce two different methods named `hasHigherBalanceThan`. The method `hasHigherBalanceThan(BankAccount)` serves to compare the balance of a bank account with the balance of another bank account. The method `hasHigherBalanceThan(long)` on the other hand serves to compare the balance of a bank account with a given amount.

In Java, overloaded methods must differ in their formal argument lists. Overloaded methods in Java differ if they have a different number of formal arguments. Overloaded methods also differ if at least one of their formal arguments at corresponding positions in the argument list have different types. In this way, the Java compiler can use the number of actual arguments and their type to match invocations of overloaded methods with the intended method. The overloaded methods `hasHigherBalanceThan` in the class of bank accounts differ in the type of their formal argument. Each time we invoke the method `hasHigherBalanceThan` against a bank account, the type of the actual argument — a long integer or a bank account — easily reveals which of both overloaded methods was intended.

1.1.2.6 Constructors

In addition to instance methods and class methods, a class also offers facilities for creating new objects. For that purpose, a series of *constructors* complement the definition of a class. The most important task of a constructor is to properly initialize each of the properties ascribed to the new object. In fact, constructors in Java only serve to initialize newly created objects. The actual creation of the new object precedes the invocation of the constructor. The term “constructor” is therefore a bit misleading.

In Java, constructors are named after their class. In other words, each constructor is a method whose name is identical to the name of its class. In this way, we can easily distinguish constructors from the other methods of a class. A class may introduce several constructors. In that case, the rules of overloading methods apply as explained on page 25. Example 2 on page 11 illustrates the definition of some constructors for the class of bank accounts. The most extended constructor serves to create a new bank account initialized with given number, given balance and given blocked state. The other

constructors use default values for the initial balance and/or for the blocked state of the new bank account.

A class typically offers several constructors. Each constructor reflects a particular protocol for creating new objects of its class. Java's decision to name constructors after their class sometimes restricts the cluster of constructors that a class can offer. Indeed different creation protocols may require the same amount of data and the same type of data. Java makes it impossible to work out different constructors for such protocols. As an example, consider a class of coordinates in a two-dimensional area. We can initialize a coordinate in the Cartesian way by supplying a horizontal and a vertical displacement. We can also initialize a coordinate in the Polar way by supplying an angle and a radius. Both initializations require two numeric values. A Java class of coordinates cannot offer both constructors as such. The class may only support a single initialization protocol, for instance to initialize coordinates in a Cartesian way. An alternative is to offer a single constructor that combines both initialization protocols. In that case, we must add a rather artificial flag to the formal argument list. That flag indicates whether the supplied numeric values must be interpreted as Cartesian coordinates or as Polar coordinates.

The signature of a constructor may start with some modifiers. It always includes the name of the constructor and its formal argument list. All the constructors in the class of bank accounts are qualified **public**, meaning that all clients of the class can use them to initialize new bank accounts. By definition, constructors in Java never return a result. Contrary to mutators, we do not explicitly specify the keyword **void** as the result type of a constructor. As for instance methods, constructors also have an implicit argument that refers to an object of their class. This time, the implicit argument references the newly created object that is initialized by the constructor.

A class may introduce a constructor without any formal arguments. In Java, such a constructor is called the *default constructor* of its class. Java implicitly complements the definition of each class with a default constructor, if that class does not explicitly introduce at least one constructor. The body of the implicitly added default constructor is empty. That constructor therefore does not change any explicit or implicit initializations of properties ascribed to newly created objects. As soon as a class explicitly introduces at least one constructor, Java no longer adds an implicit definition of a default constructor to that class. If clients of a class must nevertheless be able to initialize objects with a constructor involving no arguments, the definition of that class must include an explicit

definition of such a constructor. Let us illustrate these rules with all possible cases in offering constructors for the class of bank accounts:

1. The class of bank accounts does not explicitly introduce any constructor. In that case, Java complements the definition of the class with a default constructor. In section 1.2.1, we discuss how explicit or implicit initializations of instance variables then establish the initial values for the number, the balance and the blocked state of the newly created bank account. These rules see to it that the newly created bank account is unblocked, and both its balance and its number are initialized to `0`.
2. The class of bank accounts introduces several constructors all of them involving at least one formal argument. In that case, Java does not implicitly complement the definition of the class with a default constructor. Because the class of bank account does not offer a constructor involving no arguments, clients must supply at least some specific information each time they create a new bank account. This is the case in Example 2.
3. The class of bank accounts introduces several constructors one of which does not involve any arguments. Again, Java does not implicitly complement the definition of the class with the definition of a default constructor. In this case, that implicit constructor would conflict with the explicitly defined constructor without any arguments. Because of that explicit constructor, clients of the class can initialize new bank accounts without supplying any specific information.

1.1.3 Documentation

The signature of a method reveals syntactical information. It instructs clients of a class *how* to invoke the method. The signature first of all reveals whether clients must invoke the method against an individual object of the class or against the class itself. The formal argument list describes additional information that clients must supply each time they invoke the method. Finally, the result type describes the kind of result that clients can expect from the method. Besides syntactical information, clients also need semantic information about methods. They need to know *what* they can expect when they invoke constructors, instance methods and class methods. The *specification of a method* reveals its semantics.

The specification of a method has a declarative nature. It describes what clients can expect when they invoke the method. The implementation of a method, on the other hand, has an operational character, at least in imperative programming languages. It reveals how

the method achieves its effects. We can use natural languages to describe the effects of methods in a declarative way. Natural language descriptions are rather simple to write, and they are easily understood by human beings. However, specifications in natural languages are often ambiguous and incomplete. More formal notations such as first-order logic offer an alternative. Formal specifications are accurate and unambiguous. The problem is that most programmers are not trained to read and write specifications in first-order logic. In this chapter, we only specify the semantics of methods in an informal way. Starting from Chapter 2, however, we specify each method both in a formal way and in an informal way. We believe that programmers write better informal specifications, if they have been trained to write formal specifications. In practice, we only use natural languages to document our classes. Now and then, we may clarify some aspects in a formal way.

In Java, the specification of a method is worked out in a documentation comment preceding the method's heading. We have worked out a specification for all the methods offered by the class of bank accounts in Example 3 below. We discuss all the ingredients that we use in documentation comments after the example. At this point, we have completed the class interface. We are ready to publish the definition of the class, such that clients can already start using its methods. At the same time, we can proceed with the implementation of the class.

```
/**
 * A class of bank accounts involving a bank code, a number,
 * a credit limit, a balance limit, a balance and a blocking
 * facility.
 *
 * @version 2.0
 * @author  Eric Steegmans
 */
class BankAccount {

  /**
   * Initialize this new bank account with given number, given
   * balance and given blocked state.
   *
   * @param   number
   *          The number for this new bank account.
   * @param   balance
   *          The balance for this new bank account.
   * @param   isBlocked
   *          The blocked state for this new bank account.
   * @post    If the given number is not negative, the initial
   *          number of this new bank account is equal to the
   *          given number. Otherwise, its initial number is
   *          equal to 0.
   * @post    If the given balance is not below the credit
   *          limit and not above the balance limit, the
```

```
     *          initial balance of this new bank account is equal
     *          to the given balance. Otherwise, its initial
     *          balance is equal to 0.
     * @post    The initial blocked state of this new bank
     *          account is equal to the given flag.
     */
    public BankAccount
        (int number, long balance, boolean isBlocked) { … }

    /**
     * Initialize this new bank account as an unblocked account
     * with given number and given balance.
     *
     * @param   number
     *          The number for this new bank account.
     * @param   balance
     *          The balance for this new bank account.
     * @effect  This new bank account is initialized with the
     *          given number as its number, the given balance as
     *          its balance, and false as its blocked state.
     */
    public BankAccount(int number, long balance) { … }

    /**
     * Initialize this new bank account as an unblocked account
     * with given number and zero balance.
     *
     * @param   number
     *          The number for this new bank account.
     * @effect  This new bank account is initialized with the
     *          given number as its number and zero as its
     *          balance.
     */
    public BankAccount(int number) { … }

    /**
     * Return the number of this bank account.
     *   The number of a bank account serves to distinguish it
     *   from all other bank accounts.
     */
    @Basic @Immutable
    public int getNumber() { … }

    /**
     * Return the bank code that applies to all bank accounts.
     *   The bank code identifies the bank to which all bank
     *   accounts belong.
     */
    @Basic @Immutable
    public static int getBankCode() { … }

    /**
     * Return the balance of this bank account.
     *   The balance of a bank account expresses the amount of
     *   money available on that account.
     */
    @Basic
    public long getBalance() { … }
```

```
  /**
   * Check whether this bank account has a higher balance than
   * the given amount of money.
   *
   * @param   amount
   *          The amount of money to compare with.
   * @return  True if and only if the balance of this bank
   *          account is greater than the given amount.
   */
  public boolean hasHigherBalanceThan(long amount) { … }

  /**
   * Check whether this bank account has a higher balance than
   * the other bank account.
   *
   * @param   other
   *          The bank account to compare with.
   * @return  True if and only if the other bank account is
   *          effective, and if this bank account has a higher
   *          balance than the balance of the other bank
   *          account.
   */
  public boolean hasHigherBalanceThan(BankAccount other) { …
}

  /**
   * Deposit the given amount of money to this bank account.
   *
   * @param   amount
   *          The amount of money to be deposited.
   * @post    If the given amount of money is positive, and if
   *          the old balance of this bank account incremented
   *          with the given amount of money is not above the
   *          balance limit, the new balance of this bank
   *          account is equal to the old balance of this bank
   *          account incremented with the given amount of
   *          money.
   */
  public void deposit(long amount) { … }

  /**
   * Withdraw the given amount of money from this bank account.
   *
   * @param   amount
   *          The amount of money to be withdrawn.
   * @post    If the given amount of money is positive, and if
   *          this bank account is not blocked, and if the old
   *          balance of this bank account decremented with the
   *          given amount of money is not below the credit
   *          limit, the new balance of this bank account is
   *          equal to the old balance of this bank account
   *          decremented with the given amount of money.
   */
  public void withdraw(long amount) { … }

  /**
   * Transfer the given amount of money from this bank account
```

```
    * to the given destination account.
    *
    * @param   amount
    *          The amount of money to be transferred.
    * @param   destination
    *          The bank account to transfer the money to.
    * @effect  If the given destination account is effective and
    *          not the same as this bank account, and if this
    *          bank account is not blocked, and if the old
    *          balance of this bank account decremented with the
    *          given amount of money is not below the credit
    *          limit, and if the old balance of the given
    *          destination account incremented with the given
    *          amount of money is not above the balance limit,
    *          the given amount of money is withdrawn from this
    *          bank account, and deposited to the given
    *          destination account.
    */
   public void transferTo(long amount, BankAccount
destination)
     { … }

   /**
    * Set the balance of this bank account to the given balance.
    *
    * @param   balance
    *          The new balance for this bank account.
    * @post    If the given balance is not below the credit
    *          limit and not above the balance limit, the new
    *          balance of this bank account is equal to the
    *          given balance.
    */
   private void setBalance(long balance) { … }

   /**
    * Return the credit limit that applies to all bank accounts.
    *   The credit limit expresses the lowest possible value for
    *   the balance of any bank account.
    */
   @Basic
   public static long getCreditLimit() { … }

   /**
    * Set the credit limit that applies to all bank accounts to
    * the given credit limit.
    *
    * @param   creditLimit
    *          The new credit limit for all bank accounts.
    * @post    If the given credit limit is not above the credit
    *          limit that currently applies to all bank
    *          accounts, the new credit limit that applies to
    *          all bank accounts is equal to the given credit
    *          limit.
    */
   public static void setCreditLimit(long creditLimit) { … }

   /**
    * Return the balance limit that applies to all bank
```

```
    * accounts.
    *   The balance limit expresses the highest possible value
    *   for the balance of any bank account.
    */
   @Basic @Immutable
   public static long getBalanceLimit() { … }

   /**
    * Check whether this bank account is blocked.
    *   Some methods have no effect when invoked against blocked
    *   bank accounts.
    */
   @Basic
   public boolean isBlocked() { … }

   /**
    * Set the blocked state of this bank account according to
    * the given flag.
    *
    * @param   flag
    *          The new blocked state for this bank account.
    * @post    The new blocked state of this bank account is
    *          equal to the given flag.
    */
   public void setBlocked(boolean flag) { … }

   /**
    * Block this bank account.
    *
    * @effect  The blocked state of this bank account is set to
    *          true.
    */
   public void block() { … }

   /**
    * Unblock this bank account.
    *
    * @effect  The blocked state of this bank account is set to
    *          false.
    */
   public void unblock() { … }

}
```

Example 3: Interface of the class of bank accounts.

1.1.3.1 Method summary

The specification of a method best starts with a brief introduction. That one line *summary* must enable clients to get a first, rough impression of what they can expect from the method. Java assumes that the first sentence in a documentation comment – all the words up to the first dot – introduces the element to which the comment is attached. Javadoc derives an HTML file from the entire definition of a class. That HTML file includes an alphabetic overview of all the methods offered by the class. The overview includes the

signature of each method together with its one-line summary. A hyperlink to the full documentation of each method complements that overview. Figure 1 on page 47 lists a small part of the documentation generated for the class of bank accounts. It illustrates how javadoc produces overviews of constructors and methods offered by a class.

Coding Rule 9: Start documentation comments attached to methods with a one line summary of the method's semantics.

In Example 3, the summary for the instance method `transferTo` states that the method serves to "transfer the given amount of money from this bank account to the given destination account". We use "this class name" to refer to the prime object of instance methods in documentation comments. The summary for the class method `getCreditLimit` states that it will "return the credit limit that applies to all bank accounts". In documentation comments attached to class methods we use "all class names" to stress that they have the same effect for all the objects of their class. Finally, the summary for the most extended constructor states that it will "initialize this new bank account with given number, given balance and given blocked state". In documentation comments attached to constructors, we use "this new class name" to refer to their prime object.

The introduction to a method may take more than just a single line. Other lines of text may follow the one-line summary. Javadoc does not show these extra lines in the method overview. It only shows them in the method details. In Example 3, the summary for the class method `getBankCode` states that the method serves to "return the bank code that applies to all bank accounts". An additional sentence follows the one-line summary detailing that "the bank code identifies the bank to which all bank accounts belong".

1.1.3.2 Parameters

Documentation comments attached to methods also clarify the role of each formal argument. In Java, we describe the role of each formal argument in a separate clause that starts with the predefined tag `@param`. The name of the documented argument follows the tag. A brief informal description of the formal argument's role in the method complements the clause. In Example 3 above, two successive clauses clarify the roles of the formal arguments `amount` and `destination` in the instance method `transferTo`. By convention, clauses clarifying formal arguments immediately follow the introductory

lines to the method. Figure 1 on page 47 on page illustrates how javadoc processes documentation related to formal arguments of methods.

Coding Rule 10: Clarify in a method's documentation the role of each formal argument in a clause headed by the predefined tag @param.

1.1.3.3 Postconditions

The most important part in the documentation of a method is a detailed specification of its effect. For that purpose, we formulate conditions that hold upon exit from the method. We refer to them as *postconditions* because they must hold at the end of the execution of the method. Postconditions take different forms depending on the kind of method to which they apply.

Basic inspectors

In formal and informal specifications, we must be able to talk about the state of objects involved in methods. The *state of an object* is the set of current values of all properties ascribed to that object. The state of a bank account as defined in Example 3 involves its number, its balance and its blocked state. The *state of a class* is the set of current values of all properties ascribed to that class. The state of the class of bank accounts itself involves its bank code, its credit limit and its balance limit. In the documentation of a class, we designate a number of inspectors as basic inspectors. A *basic inspector* serves to return part of the state of an object or a class. In the class of bank accounts, the inspectors `getNumber`, `getBalance`, `isBlocked`, `getBankCode`, `getCreditLimit`, and `getBalanceLimit` are basic inspectors. The first three inspectors are sufficient to return all the information concerning the state of an individual bank account. The last three inspectors are sufficient to get all the information concerning the current state of the class of bank accounts itself.

Since Java 5, we can annotate specific elements of object oriented programs. *Annotations* provide a formalized way to add information to specific ingredients of Java programs. They are also referred to as meta-data. We can attach annotations to ingredients such as constructors, instance variables, static variables, methods, formal arguments and classes. Java supports the development of annotation processors to manipulate programs involving annotated constructs. An annotation processor is some kind of preprocessor that operates on annotations in programs.

Annotations immediately precede the element to which they apply. They follow documentation comments attached to the annotated element. An annotation consists of the symbol @ followed by the name of the annotation. Annotations may carry information complemented with dedicated methods for manipulating that information. In that sense, annotations have a lot in common with objects of classes. Java only defines a few annotations itself. An example is the annotation `@Deprecated`. It indicates that clients must no longer use the entity to which the annotation applies. In the long run, authors of a class may remove deprecated ingredients.

In Java, we can define our own annotations. We use several such self-defined annotations in this book. Annotations must be defined as interfaces that obey a set of well-defined rules. We do not discuss the definition of annotations in this book. At this point, we use the self-defined annotation `@Basic` to expose the basic inspectors of a class. The annotation serves no other purpose than to differentiate ordinary methods of a class from its basic inspectors. In Example 3 above, the inspectors `getNumber`, `getBalance`, `isBlocked`, `getBankCode`, `getCreditLimit`, and `getBalanceLimit` all have the `@Basic` annotation.

Coding Rule 11: Add the annotation `@Basic` to each basic inspector of a class.

We cannot further specify the result of a basic inspector. We can only complement the one line summary with a more detailed explanation concerning the result of a basic inspector. For the inspector `getBalance` in Example 3 above, we further clarify that the balance of a bank account expresses the amount of money currently available on that account. In the same way, the documentation for the basic inspector `getBalanceLimit` reveals that the balance limit is the highest possible value for the balance of any account. Especially for basic inspectors for which there is no generally accepted name, such additional descriptions are crucial. Otherwise, clients of a class have difficulties in understanding their definition.

Coding Rule 12: Complement documentation comments for basic inspectors with a further explanation of their result, is its name is not self-explanatory.

Another self-defined annotation that we use in Example 3 above is `@Immutable`. This annotation only applies to inspectors without formal arguments. The result returned by an

immutable inspector does not change over time. Clients of a class can safely cache results returned by immutable inspectors. An immutable class inspector always returns the same result, even if properties ascribed to its class have changed. In the same way, an immutable instance inspector always returns the same result whenever it is invoked against the same object, even if the state of that object has changed. However, immutable inspectors may return a different result at times they are invoked against objects that are not in a steady state. We illustrate this issue in Chapter 2 on page 174. In the class of bank accounts, the instance inspector `getNumber` is an immutable inspector. This means that the number ascribed to any bank account cannot change over time. The annotation `@Immutable` also applies to the class inspectors `getBankCode` and `getBalanceLimit`. This means that the bank code and the balance limit shared by all bank accounts do not change over time.

Coding Rule 13: Add the annotation `@Immutable` to inspectors without formal arguments, whose result does not change as long as the application is executing.

We use basic inspectors in the specification of other methods. In specifying the effect of the mutator `deposit` for instance, we state that the balance of the prime bank account increases with the given amount of money. For the least extended constructor, we state that the balance of the new bank account is initialized to `0`. In both cases, we describe how these methods influence the state of their prime bank account. We use the basic inspector `getBalance` to describe changes to the balance. If we create a new bank account using the least extended constructor, we know that the basic inspector `getBalance` applied to that bank account returns `0`. If we subsequently deposit `200` to that bank account, the specification of the mutator `deposit` reveals that the basic inspector `getBalance` returns `0+200 == 200`, ⋯

For the class of bank accounts in Example 3, there is only one possible set of basic inspectors. However, for more complex classes several candidate sets may exist. We must choose the basic inspectors of a class with enough care, because that choice has an impact on the specification of all other methods of the class. First of all, the set of basic inspectors must be minimal. This means that it may not be possible to leave out one of the basic inspectors from the set, and still be able to get all information about the state of individual objects of the class and about the class itself. If we still have a choice between

several such minimal sets, we usually opt for the set with the smallest number of basic inspectors.

Derived inspectors

In addition to basic inspectors, a class may also introduce other inspectors. We refer to such inspectors as *derived inspectors*, because their result can be described in terms of basic inspectors. In Example 3 above, the two inspectors named `hasHigherBalanceThan` are both derived inspectors. We could add lots of other derived inspectors to the class of bank accounts. Examples are (1) an inspector to check whether the balance of a bank account is positive, zero, or negative, (2) an inspector returning the largest amount of money that we can withdraw from a bank account, (3) an inspector returning the largest amount of money that we can deposit to a bank account, (4) an inspector to check whether two bank accounts have the same blocked state, ··· Especially for more complex classes, the list of candidate derived inspectors soon grows exponentially. In the definition of a class, we only offer those inspectors that we expect our clients to use heavily. Clients can always themselves compute the result of missing inspectors by means of the basic inspectors of a class.

We specify the result of a derived inspector in one or more return clauses. A return clause starts with the predefined tag `@return`. A specification of the result of the inspector immediately follows that tag. In Example 3 on page 32, the return clause for the derived inspector `hasHigherBalanceThan(long)` states that it returns "true if and only if the balance of this bank account is larger than the given amount". Notice that we use the basic inspector `getBalance` in this phrase. The specification indeed states that the derived inspector returns "true if the result obtained from the basic inspector `getBalance` invoked against the implicit bank account is larger than the given amount, and false otherwise". Figure 1 on page 47 illustrates how javadoc processes return-clauses in documentation comments.

Coding Rule 14: Specify the result of derived inspectors in one or more clauses headed by the predefined tag `@return`.

The specification of the derived inspector `hasHigherBalanceThan(BankAccount)` in Example 3 is a bit more complicated. The inspector has a formal argument of class type, and there is no guarantee that such arguments reference an effective object of that class.

In Java, formal arguments and variables of class type have reference semantics, meaning that they can store a reference — a pointer — to an object of their class. We further explain in section 1.2.2.2 that such entities can also store the *null reference* to indicate that they currently do not reference an effective object. The documentation of the derived inspector `hasHigherBalanceThan(BankAccount)` reveals among others that it returns false, if the method must compare the balance of a bank account with the balance of a non-effective bank account. Formal arguments and variables of primitive type, on the other hand, have value semantics in Java. Such variables always store an effective value of their type. This explains why we do not have to bother about a non-effective value in the specification of the derived inspector `hasHigherBalanceThan(long)`.

Another interesting facet in the specification of the inspector `hasHigherBalanceThan(BankAccount)` is the way we specify its result in case an effective bank account is supplied. The specification then states that the inspector returns "true if and only if this bank account has a higher balance than the balance of the other bank account". If we read this phrase carefully, we see that we specify the result of this derived inspector in terms of the other derived inspector `hasHigherBalanceThan(long)`. Indeed, the specification states that we obtain the result by applying that derived inspector with the balance of the other bank account — a long integer — as its actual argument. Specifying derived inspectors in terms of other derived inspectors is no problem. Indeed, we can always expand such specifications to specifications involving only basic inspectors. We only have to substitute occurrences of derived inspectors by their specification for that purpose. In the example, such a substitution would reveal that the inspector returns "true if and only if the balance of this bank account is larger than the balance of the other bank account".

We can spread the specification of the result returned by a derived inspector over several return clauses. In that case, each return clause typically specifies the result of that inspector in a particular case. In the code snippet below, we spread the specification of the derived inspector `hasHigherBalanceThan(BankAccount)` over two return clauses. The first return clause covers the case in which the other bank account is not effective. The second return clause covers the opposite case. It is a matter of personal taste and style, whether or not to spread result specifications over several return clauses.

```
/**
 * Check whether this bank account has a higher balance than
 * the other bank account.
 *
```

```
 * @param   other
 *          The bank account to compare with.
 * @return  If the other bank account is not effective, the
 *          method returns false.
 * @return  If the other bank account is effective, the
 *          method returns true if and only if this bank
 *          account has a higher balance than the balance of
 *          the other bank account.
 */
public boolean hasHigherBalanceThan(BankAccount other) { … }
```

In the specification of the derived inspector `hasHigherBalanceThan(BankAccount)`, we take care of the exceptional case in which the other bank account is not effective. In specifying a method, it is of utmost importance that we not only describe how the method behaves in the normal case. It is of equal importance that we also describe how a method behaves in all possible exceptional cases. In Example 3 on page 32, we specify the derived inspector `hasHigherBalanceThan(BankAccount)` in such a way that it returns a result in all possible cases, including exceptional cases. We refer to such methods as *total methods*, because they are similar to total functions in mathematics.

Coding Rule 15: If you choose to handle an exceptional case in a total way, deal with it in a return-clause in case of a derived inspector, and in a postcondition in case of a mutator or a constructor.

In this book, we discuss several strategies — *paradigms* — to deal with exceptional cases. The approach that we follow in this chapter is known as *total programming*. This paradigm states to handle exceptional cases in return clauses, respectively in postconditions of methods. This means that we use the same constructs to handle both exceptional cases and normal cases. The border line between normal cases and exceptional cases is therefore rather vague in this paradigm. This must not be a problem. The important thing is that the specification reveals how the method behaves in each possible case. In Chapter 2, we discuss the paradigm of *nominal programming*. This paradigm uses preconditions to tell clients not to invoke methods under exceptional conditions. Finally, Chapter 3 discusses the paradigm of *defensive programming*. This paradigm uses exceptions to signal invocations of methods under exceptional conditions. We cannot give any guidelines concerning when to handle an exceptional case in a total way, in a nominal way or in a defensive way. This is a matter of personal taste and style. Again, however, how we handle exceptional cases is not the million dollar question; the most important thing is that we deal with all of them in the definition of each method.

Coding Principle 3: Deal with all exceptional cases in the definition of a method, such that clients always know exactly how it behaves under all possible circumstances.

Mutators

We can specify the semantics of mutators in two different ways. The most common approach is to use inspectors to specify how mutators change the state of some of the objects involved in them. A mutator then specifies conditions that must hold upon exit. We refer to such conditions as *postconditions*. Java itself does not support the notion of postconditions. In this book, we therefore use the self-defined tag `@post` to outline specifications of postconditions in documentation comments. A description of the condition that must hold upon exit from the method follows the tag. We may include several clauses headed by the tag `@post` in the documentation of a single method. Each clause then describes a postcondition that must hold upon exit from the method. Figure 1 on page 47 illustrates how javadoc processes post-clauses in documentation comments. Because we use a self-defined tag in this case, we must supply a taglet to javadoc telling it how to process postconditions in documentation comments.

In Example 3 on page 32, the mutators `deposit`, `withdraw`, `setBalance`, `setCreditLimit`, and `setBlocked` all use postconditions in their specification. As an example, the postcondition for the basic mutator `setBlocked` specifies that "the new blocked state of this bank account is equal to the given flag". The postcondition uses the basic inspector `isBlocked` to specify how the state of the prime object changes. Indeed, the postcondition states that an invocation of the basic inspector `isBlocked` against the implicit bank account upon exit from the mutator `setBlocked`, yields the same value as the value of the given flag.

Coding Rule 16: Specify changes to the state of objects or classes involved in mutators and constructors in one or more clauses headed by the self-defined tag `@post`.

The specification of the mutator `setCreditLimit` is a bit more complicated. The policy of our bank states that the credit limit for all bank accounts can never be a positive value. Moreover, the bank guarantees its clients that it will never raise the credit limit to a less negative value. Because the credit limit is initialized to `0`, it suffices to state that in changing the credit limit, the new value may not exceed the current value for the credit limit. Restrictions also influence the definition of the mutator `setBalance`. That method

must deal with the exceptional case in which the supplied balance is not between the credit limit and the balance limit.

At first sight, we do not specify how these methods behave in that case. Indeed, the postcondition of `setCreditLimit` only states that "the new credit limit that applies to all bank accounts is equal to the given limit, if that limit is not above the credit limit that currently applies to all bank accounts". The postcondition of the method `setBalance` only states that "the new balance of its prime bank account is equal to the given balance, if that balance is not below the credit limit and not above the balance limit". It seems that these postconditions do not tell us how the credit limit, respectively the balance change in case the given values are not legal. However, in the next paragraph we explain that these postconditions implicitly state that the credit limit, respectively the balance remain unchanged in those cases. The message here is that setters for a property must deal in one way or another with the exceptional case in which illegal values are supplied to them. As mentioned before, this chapter deals with such exceptional cases in a total way, turning methods such as `setCreditLimit` and `setBalance` into total methods. Both methods implicitly state in their postcondition that they have no effect if the supplied values are illiegal.

Coding Rule 17: The setter `setα` for a property `α` must deal with illegal values in a nominal way, in a total way or in a defensive way.

The so-called *inertia axiom* applies to specifications of mutators and constructors. This axiom states that, if a method does not explicitly specify changes to some of the properties ascribed to objects or classes, that method leaves those properties untouched. Because of the inertia axiom, the specification of the mutator `setCreditLimit` thus implies that the credit limit is left untouched if clients supply a limit that exceeds the current limit. In the same way, the specification of the mutator `setBlocked` implies that this method leaves the number and the balance of the implicit bank account untouched.

Because of the inertia axiom, specifications of mutators and constructors are limited in size. Indeed, in specifying mutators, we must only focus on describing the things that change. This is even more true because the inertia axiom not only applies to properties of objects or classes that are directly involved in methods. The axiom extends to all other objects and classes involved in the running application. In other words, if a method does not specify any changes at all to some objects or classes, that method does not affect the

state of those objects and classes. The inertia axiom for the mutator `setCreditLimit` thus implies that it does not change the state of any existing bank account. In the same way, the specification of the mutator `setBlocked` implies that it only affects the blocked state of its prime bank account. The method leaves the state of all other bank accounts and of the class of bank accounts itself untouched.

The mutators `setBlocked` and `setBalance` perform state changes that are independent of the current state. As an example, the mutator `setBalance` registers the given amount of money as the new balance of its prime bank account, regardless of the balance of that account upon entry to the mutator. The mutators `deposit` and `withdraw`, on the other hand, compute the new balance of their prime bank account in terms of their old balance. In such cases, we must be able to distinguish the value of a property upon entry to a method from the value of that property upon exit. In informal specifications, we refer to the value of a property upon entry to a method as the *old* value of that property. The *new* value of a property then refers to the value of that property upon exit from the method. The postcondition for the mutator `deposit` therefore states that "the *new balance* of this bank account is equal to the *old balance* of this bank account incremented with the given amount of money". If we do not qualify the value of a property in the specification of a postcondition, we refer to the new value of that property.

Notice once more the exceptional cases in the definitions of the mutators `deposit` and `withdraw`. The first exceptional case concerns a non-positive amount of money. In both methods we specify that the changes to the balance of the prime bank account do not apply in that case. Indeed, in that case we do not specify an explicit change to the balance of the prime bank account. This means that the inertia axiom applies, and that the balance of the prime bank account does not change. A withdrawal is also not possible if the prime bank account is blocked. The final exceptional case deals with possible overflows in the balance of the prime bank account. The balance of each bank account must at all times be in the range established by the credit limit and the balance limit. Again, the specification of both methods reveals that the balance of their prime bank account is left untouched, if a withdrawal or a deposit would cause an overflow in the balance.

Another way to describe the semantics of mutators is to specify their effect in terms of other mutators. The specification then expresses that the effect of that mutator is at all times equivalent with the combination of effects of other mutators. We use another self-

defined tag `@effect` to specify that a mutator has the same effect as other mutators. A description of how other mutators achieve the effect of the derived mutator follows the tag. A single method may use both effect clauses and clauses involving postconditions in its documentation. This means that we specify parts of the semantics of the method directly in terms of postconditions, while we specify other parts in a more indirect way as some combination of effects of other mutators.

In Example 3 on page 32, we specify the mutators `transferTo`, `block` and `unblock` in this way. We use the mutators `deposit` and `withdraw` to specify the effect of the derived mutator `transferTo`. Indeed, the specification states that "if the given destination is effective and ⋯, the given amount of money is withdrawn from this bank account, and deposited to the given destination account". If the given destination account and the given amount of money satisfy all stated conditions, the method thus has the combined effects of invoking the mutator `withdraw` against the prime bank account, and of invoking the mutator `deposit` against the given destination account. Both invocations use the given amount of money as their actual argument.

Coding Rule 18: Specify the effect of mutators and constructors in terms of other mutators, respectively constructors, if their effect must at all times be equivalent with the combined effect of the latter methods. Use one or more documentation clauses headed by the self-defined tag `@effect` for that purpose.

Notice that we also describe *how* to combine the effects of mutators that we use in effect clauses. In the specification of the mutator `transferTo`, we state that it must achieve the effects of both the withdraw *and* the deposit. Other methods could specify that they achieve either the effect of one mutator *or* the effect of another mutator. This also explains why we handle a transfer a money from a bank account to the same bank account in a separate way. Indeed, if we specify that also in this exceptional case the effect of a transfer is the combined effect of a withdrawal and a deposit, we state that the balance of that account must at the same time be incremented with the given amount of money and decremented with that same amount. Obviously, this is impossible to achieve for any non-zero amount of money to be transferred. As an example, given a bank account with a balance of `500`, a transfer of `100` from that account to that very same account cannot result in a balance for that account that is `400` and `600` at the same time.

In specifications of most mutators, we can choose to specify their semantics with postconditions or with effect clauses. The code snippet below shows a specification for

the mutator `transferTo` in terms of postconditions. In fact, this is the specification that results from substituting the occurrences of the mutators `withdraw` and `deposit` by their respective postconditions in the original specification. This leads to a first advantage of effect clauses over postconditions. If we specify more complex mutators in terms of other, more basic mutators, we re-use specifications of the latter mutators. If we specify more complex mutators directly in terms of postconditions, on the other hand, we often work out copies of postconditions of less complex mutators.

Specifications involving postconditions really start to differ from specifications involving effect clauses, if we take evolution into account. The specification of Example 3 expresses that a transfer of money is *at all times* equivalent with a withdrawal combined with a deposit. If the methods for withdrawing or depositing money change over time - e.g., by imposing a cost, respectively a bonus on each transaction -, these changes automatically propagate to the method `transferTo`. In the specification below, we do not have such a tight bond between the mutators `withdraw` and `deposit` on the one hand, and the mutator `transferTo` on the other hand. Changes to the specification of the mutators `withdraw` and `deposit` do not have an impact on the semantics of the mutator `transferTo` as worked out below. Obviously, both specifications are correct. We must decide to which extent changes to the specification of mutators must propagate to the specification of other mutators.

```
/**
 * Transfer the given amount of money from this bank account to
 * the given destination account.
 *
 * @param   amount
 *          The amount of money to be transferred.
 * @param   destination
 *          The bank account to transfer the money to.
 * @post    If the given destination account is effective and
 *          not the same as this bank account, and if the
 *          given amount of money is positive, and if this bank
 *          account is not blocked, and if the old balance of
 *          this bank account decremented with the given amount
 *          of money is not below the credit limit, and if the
 *          old balance of the given destination account
 *          incremented with the given amount of money is not
 *          above the balance limit, the new balance of this
 *          bank account is equal to the old balance of this
 *          bank account decremented with the given amount of
 *          money.
 * @post    If the given destination account is effective and
 *          not the same as this bank account, and if the given
 *          amount of money is positive, and if this bank
 *          account is not blocked, and if the old balance of
 *          this bank account decremented with the given amount
 *          of money is not below the credit limit, and if the
```

```
     *          old balance of the given destination account
     *          incremented with the given amount of money is not
     *          above the balance limit, the new balance of the
     *          given destination account is equal to the old
     *          balance of the given destination account
     *          incremented with the given amount of money.
     */
    public void transferTo(long amount, BankAccount destination)
        { … }
```

Constructors

The specification of constructors is very similar to the specification of mutators. The only difference is that a constructor specifies among others the initial state of the newly created object, while a mutator specifies among others the next state of its prime object. Both mutators and constructors may specify changes to the state of other objects involved in them. In Example 3 on page 32, we specify the state of the newly created bank account resulting from the most extended constructor directly in terms of basic inspectors. The documentation comment attached to the most extended constructor therefore includes three postconditions. Notice once more how we apply the principles of total programming in the definition of that constructor. In the exceptional case that the given number is not positive, the first postcondition states that the number of the newly created bank account is set to `0`. In the same way, the second postcondition states that the initial balance of the newly created bank account is set to `0` if the given balance is out of range.

We specify the two other constructors in the definition of the class of bank accounts in terms of the most extended constructor. This is similar to specifying the semantics of mutators in terms of other mutators. The documentation comments attached to both constructors therefore both include an *effect clause*. The specification of the constructor involving only a number and a balance in terms of the most extended constructor states that it has the same effect as "the initialization of a newly created bank account involving the given number as its number, the given balance as its balance, and false as its blocked state". We subsequently specify the least extended constructor in terms of the constructor involving only a number and a balance.

Upon entry to a constructor, we assume that each of the properties ascribed to the newly created object is initialized to the *default value* of its type. In other words, we assume that each basic inspector returns the default value of its type, whenever it is invoked against the newly created object upon entry to a constructor. When invoked

against the newly created bank account upon entry to a constructor offered by the class of bank accounts, the basic inspectors `getNumber` and `getBalance` thus return `0` – the default value for integer types. At that same point, the invocation of the basic inspector `isBlocked` against the newly created object returns **`false`** – the default value for the type **`boolean`**.

A constructor must not explicitly describe in its specification how it initializes all the properties of its prime object. Because of the *inertia axiom*, a constructor leaves the initial value of a property untouched if it does not explicitly mention any changes to that property. This means that in Example 3 a single postcondition would suffice to specify the entire effect of the least extended constructor. That postcondition would then state that the number of the newly created bank account would be equal to the given number. In that case, the constructor would not mention changes to the balance and the blocked state of its prime account. Both properties would then stick to their initial values.

`class` `BankAccount`

A class of bank accounts involving a bank code, a number, a credit limit, a balance limit, a balance and a blocking facility.

Constructor Summary

`BankAccount(`**`int`** `number)`
Initialize this new bank account as an unblocked account with given number and zero balance.

Method Summary

`void`	`deposit(`**`long`** `amount)` Deposit the given amount of money to this bank account.
`long`	`getBalance()` Return the balance of this bank account.
`boolean`	`hasHigherBalanceThan(`**`long`** `amount)` Check whether this bank account has a higher balance than the given amount of money.

Constructor Detail

`public` `BankAccount(`**`int`** `number)`
Initialize this new bank account as an unblocked account with given number and zero

balance.

Parameters:

`number` - The number for this new bank account.

Effect:

This new bank account is initialized in the same way a new bank account would be initialized involving the given number as its number and zero as its balance.

Method Detail

`@Basic` **`public long`** `getBalance()`

Return the balance of this bank account. The balance of a bank account expresses the amount of money available on that account.

`public boolean` `hasHigherBalanceThan(`**`long`** `amount)`

Check whether this bank account has a higher balance than the given amount of money.

Parameters:

`amount` - The amount of money to compare with.

Result:

True if and only if the balance of this bank account is greater than the given amount.

`public void` `deposit(`**`long`** `amount)`

Deposit the given amount of money to this bank account.

Parameters:

`amount` - The amount of money to be deposited.

Postcondition:

If the given amount of money is positive, and if the old balance of this bank account incremented with the given amount of money is not above the balance limit, the new balance of this bank account is equal to the old balance of this bank account incremented with the given amount of money.

Figure 1: Partial documentation file for the class of bank accounts as it is generated by javadoc.

1.1.4 Class usage

As soon as we have finished the interface of a class, clients can already start writing code in which they use that class. Indeed, the class interface supplies all the information clients need to manipulate individual objects of that class, as well as the class itself. In fact, the interface of a class is the only source of information intended for the clients of the class. Internal details that are irrelevant for clients may not pollute class interfaces. Obviously, client code can only execute as soon as we also finish the implementation of a class. However, if we first work out interfaces for all the key classes involved in an

application, we can ask different members of the team to work out their implementation. This is one of the strengths of the object oriented paradigm. It offers concepts for partitioning a software system in a set of loosely coupled modules. Each of these modules – classes – can be developed independent of each other. Moreover, changes to the implementation of one module should not have an impact on other modules.

1.1.4.1 Construction of new objects

In Java, the operator **new** serves to create new objects of classes. The expression **new** `ClassName(`a_1`, `a_2`, …, `a_n`)` results in a new object of the named class. The actual arguments a_1, a_2, …, a_n serve to initialize the newly created object. The evaluation of the expression involves the following steps:

1. The Java Virtual Machine (JVM) allocates a sufficient amount of memory to store the state of a new object of the named class. In section 1.2, we will explain that the body of a class includes definitions of instance variables that describe the kind of information to be stored in each of its objects. Java derives the amount of memory to be allocated for an object from the type of those instance variables. As an example, the memory allocated to each object of the class of bank accounts must be large enough to store its number in the form of an integer number, its balance in the form of a long integer number, and its blocked state in the form of a `boolean`. In addition to a sufficient amount of memory to register object-specific information, the memory allocated to objects also includes room to store some bookkeeping information. The creation of a new object fails if no more memory is available. The Java Virtual Machine then throws an exception of type `OutOfMemoryError`.
2. The Java Virtual Machine *initializes* all the properties ascribed to the newly created object to the default value of their type. Java defines the following *default values* for its types: `0` for integer types, `0.0` for floating point types, **`false`** for the boolean type, the symbol `NUL` - the symbol with internal code `0x0000` - for the character type, and the null reference for class types. The balance of a newly created bank account is thus initialized to `0`; its blocked state is initialized to **`false`**.
3. The Java Virtual Machine invokes a *constructor* of the named class against the newly created object. The formal argument list of that constructor must match with the actual arguments supplied in the creation of the new object. This means that each assignment of the actual argument a_i to the corresponding formal argument f_i must

be a legal assignment. If no matching constructor is defined in the given class, the Java compiler refuses the code.

4. The Java Virtual Machine returns a reference – a pointer – to the newly created object. Typically, but not necessarily, that reference is assigned to some variable.

The code snippet below illustrates the creation of some new bank accounts. The snippet starts with the declaration of some variables intended to store references to objects of the class of bank accounts. Hereafter, we create a first new bank account using the most extended constructor offered by the class of bank accounts. From the documentation of that constructor, we know that it initializes the number of the new bank account to `1234567`. The constructor further initializes the balance of the new bank account to `1000`, at least if that amount does not exceed the balance limit that applies to all bank accounts. Finally, the constructor initializes the blocked state of the new bank account to **`false`**. In the code snippet below, we assign the reference to the newly created bank account to the variable `myAccount`.

In creating the second bank account, we use the constructor involving only a number and an initial balance. The documentation of that constructor reveals among others that it initializes the balance of this new account to `0`, at least if a value of `-30000` is below the credit limit for all accounts. In creating the third bank account, we use the least extended constructor of the class of bank accounts. This new bank account therefore has a balance of `0`, and **`false`** as its blocked state.

The Java compiler rejects the final instruction in the code snippet below. Indeed, the class of bank accounts has no constructor involving a boolean as its single formal argument.

```
BankAccount myAccount;
BankAccount yourAccount;
BankAccount hisAccount;
BankAccount herAccount;

myAccount   = new BankAccount(1234567,1000,false);
yourAccount = new BankAccount(1111111,-30000);
hisAccount  = new BankAccount(9876543);
herAccount  = new BankAccount(true);
```

1.1.4.2 Invocation of instance methods

As soon as we have created some objects of some class, we can start invoking *instance methods* offered by that class against those objects. Object oriented programming languages emphasize the role of the prime object in method invocations.

Syntactically, the prime object precedes the instance method invoked against it. This originates from the *message paradigm* that was popular in the early years of object oriented programming. That paradigm states that the execution of an object oriented program proceeds by sending messages to objects. A receiving object responds to an incoming message by returning a result to the sender of the message. In preparing its response, a receiving object may in turn send messages to other objects. Java and other modern object oriented programming languages no longer use the terminology of sending messages to objects. They rather talk about invoking methods against objects.

Syntactically, the invocation of an instance method has the following form: `ObjectExpression.instanceMethodName(`a_1`, `a_2`, …, `a_n`)`. The object resulting from the evaluation of the object expression is the object against which the instance method is invoked. If the evaluation of that expression does not result in an effective object, the Java Virtual Machine throws an exception of type `NullPointerException`. Notice that the object against which an instance method is invoked, precedes the name of that method. A dot separates the object expression from the instance method. A list of actual arguments enclosed in parentheses follows the name of the instance method.

The code snippet below illustrates the invocation of instance methods against objects of the class of bank accounts. The first two statements following the declarations illustrate how to invoke the simple methods `deposit` and `withdraw` against the bank account referenced by the variable `myAccount`. According to the documentation of both methods, the balance of that account changes to `6000` as a result of both invocations, at least if the balance limit that applies to all bank accounts is sufficiently high. Because both methods do not return a result, they both are Java statements.

In the next statement, we invoke the inspector `getBalance` against that same bank account. We assign the resulting value – `6000` – to the local variable `myBudget`. Because the instance method `getBalance` returns a result, the method invocation itself is a Java expression. In the code snippet below, that expression is the right-hand side of an assignment statement. Notice, by the way, that we can mix statements and variable declarations in a Java program. Notice also that Java allows us to initialize variables in their declaration.

In the final instructions in the code snippet below, we invoke the instance method `transferTo` against different bank accounts. According to the documentation of that method, the first transfer will be successful. As a result, the balance of my account drops

to `0`, and the balance of your account raises to `6000`. The second invocation of the instance method `transferTo` has no effect, because the destination account is not effective. The final invocation results in a `NullPointerException`, because the variable `hisAccount` is not referencing an effective bank account at that point.

```
BankAccount myAccount = new BankAccount(1234567,1000);
BankAccount yourAccount;
BankAccount hisAccount = null;

myAccount.deposit(10000);
myAccount.withdraw(5000);

long myBudget = myAccount.getBalance();

yourAccount = new BankAccount(9876543);
myAccount.transferTo(myBudget, yourAccount);
yourAccount.transferTo(100,hisAccount);
hisAccount.transferTo(100,myAccount);
```

Java is a language with strong typing. This means that we must specify a type for each entity – variable, formal argument, return type, ⋯ – in a Java program. The Java compiler consistently checks whether values and objects that we assign to variables or formal arguments are consistent with the declared type of those entities. We refer to the declared type of an entity as that entity's *static type*. The static type of an entity can be a primitive type. In that case, the entity can only store values of that primitive type. As an example, in the code snippet above, the variable `myBudget` is declared to store long integer values. The static type of an entity can also be a class type. In that case, the entity can only store references to objects belonging to that class. In the code snippet above, the variable `myAccount` is declared to store references to bank accounts.

The notion of a static type not only applies to entities. Java associates a static type with each expression. Obviously, the static type of an expression depends on its form. If the expression consists of a variable, the static type of that expression is the static type of that variable. If the expression is an invocation of a method, its static type is the return type of that method. We discuss some other, more complex forms of expressions in section 1.2.3.

Method resolution

The Java compiler uses static types in *resolving* method invocations. Given an invocation of an instance method against some object expression, the Java compiler searches for a method with the given name in the class corresponding to the static type of that object expression. If no such instance method exists in that class, the Java compiler

refuses the code. In the code snippet above, we invoke the instance method `withdraw` against `myAccount`. Because the static type of the latter expression is `BankAccount`, the Java compiler searches for an instance method named `withdraw` in the class of bank accounts. If we would invoke an instance method named `marry` against `myAccount`, the Java compiler would signal us that no such method exists in the class of bank accounts.

In resolving method invocations, the Java compiler further examines the actual argument list. Each argument in the formal argument list of the invoked method must match with the corresponding actual argument supplied in the invocation. This means that each assignment of the actual argument a_i to the corresponding formal argument f_i must be a legal assignment. In other words, we must be able to assign elements of the static type of each actual argument to the corresponding formal argument. As an example, consider the last invocation of the instance method `transferTo` in the code snippet above. The first actual argument is of static type **`int`**, which is assignable to the formal argument `amount` of type **`long`**. The second actual argument is of static type `BankAccount`, which corresponds to the type of the second formal argument of the method. The Java compiler therefore accepts that invocation.

Because of *overloading*, a class may define several instance methods with the same name. In searching for the actual instance method to be invoked, the Java compiler uses the static types of actual arguments to resolve method invocations. As an example, consider the Java expression `myAccount.hasHigherBalanceThan(200)`. In searching for a matching method, the Java compiler finds two instance methods named `hasHigherBalanceThan` in the class of bank accounts. Because the static type of the supplied actual argument is **`int`**, the compiler selects the instance method with a formal argument of type **`long`**. Indeed, values of type **`int`** are assignable to entities of type **`long`**. Values of type **`int`** are not assignable to entities of type `BankAccount`.

1.1.4.3 Invocation of class methods

The rules for *invoking class methods* are similar to those for invoking instance methods. This time, however, the name of the class precedes the class method to be invoked. The code snippet below illustrates the invocation of class methods. In the first statement, we invoke the class method `setCreditLimit` against the class `BankAccount`. The documentation of that method reveals that this method changes the credit limit for all bank accounts to `-1000`. In the second statement, we invoke the class method `getCreditLimit` against the class of bank accounts. The documentation of that

method reveals that it returns the credit limit that applies to all bank accounts. In the snippet below, the value `-1000` is thus assigned to the local variable `creditLimit`.

```
BankAccount.setCreditLimit(-1000);
long creditLimit = BankAccount.getCreditLimit();
```

Java also allows to invoke class methods against individual objects of their class. Assume that we declare a variable `myAccount` to reference a bank account. The invocation `myAccount.getCreditLimit()` via that variable is then fully equivalent with the invocation `BankAccount.getCreditLimit()`. In fact, the variable `myAccount` only serves to identify the class against which we want to invoke the class method `getCreditLimit`. More in particular, the static type of that variable determines the class against which the method is actually invoked. Most books on programming in Java advice against invocations of static methods via individual objects. The resulting code is confusing because programmers may have the impression that they invoke an instance method. We follow that advice.

Coding Advice 7: Invoke class methods against their classes. Do not invoke class methods against objects of their class.

1.1.4.4 Application Program Interface

Software systems must be able to communicate with their end users. They must be able to read data from input devices and to display information on output devices, they must be able to store information on disks and other storage devices, they must be able to send data across network connections, … Operating systems manage all these resources. They offer software systems running under their supervision facilities to manipulate such resources. An *Application Program Interface* (API) therefore complements the definition of a programming language. An API for a language specifies how programs written in that language can invoke services offered by the underlying operating system The Java API is a huge collection of predefined classes. It supports input/output from command-driven terminal windows, from graphical user interfaces, from web browsers, … The Java API also supports network connectivity, facilities for managing files and directories, multi-threading, … Moreover, the Java API offers additional facilities that are not related to the underlying operating system. As an example, Java offers a collections framework supporting all kinds of predefined data structures. We discuss that framework in more detail in Part II.

Most operating systems offer an Application Program Interface in the procedural language C. However, that interface differs from operating system to operating system. As an example, operating systems may use different names for identical facilities. They may also use different structures and protocols to communicate with devices they manage. Programs that directly appeal to the Application Program Interface offered by a particular operating system are therefore not portable. In Java, the Application Program Interface is an integral part of the definition of the language. The language guarantees that its API is identical on all platforms. Each time Java must be installed on a new platform supporting a new operating system, all the classes in the Java API must be implemented on that platform. In this way, the burden shifts from the ever growing group of Java programmers to a small group of software engineers building and managing operating systems.

Apart from being a platform-independent interface to the operating system, the Java Application Program Interface is inline with the principles of object oriented programming. Devices managed by an operating system are presented as instances of classes. Operations for manipulating these devices then become methods offered by their class. Unfortunately, the structure of the Java API is not always as good as it could be. A typical example is the way the API struggles with dates and calendars. Sometimes the documentation is not accurate enough, although the quality has improved considerably over the last releases. Graphical user interfaces do not always look as fancy as they could be. Still, the mere existence of an object oriented, portable and uniform Application Program Interface is one of the major strengths of Java.

The Java Application Program Interface is structured in *packages*. Each package groups a series of related classes. As an example, the package `io` groups a large number of classes supporting input and output. We discuss packages in more detail in section 5.6. At this point, we take a quick look at the predefined classes `String` and `PrintStream`, because we need them further on in this chapter. At other points in this book, we study other classes in the Java API. However, we do have not included a complete overview of the Java API in this book. We strongly suggest to turn to the Java homepage at java.sun.com. There, Java programmers can browse the current version of the Java Application Program Interface.

Class `String`

Strings are sequences of characters. In Java, strings are instances of the predefined class `String`. That class is part of the Java API, and belongs to the package `java.lang`. Strings are immutable, meaning that a string cannot change once it has been created. This explains why the class `String` does not offer any mutators for changing the state of individual strings. The class offers lots of methods for examining individual characters of a string, for comparing strings, for searching strings, for extracting substrings, ···

The Java API offers the predefined classes `StringBuffer`, whose instances are mutable strings. Since Java 5, the API includes another class `StringBuilder` that is identical in nature to `StringBuffer`. Contrary to string buffers, different threads cannot use objects of the class `StringBuilder` safely. However, because they are not thread-safe, the methods offered by `StringBuilder` are more efficient in time. Both classes offer equivalent methods for inspecting and changing their objects.

Because strings are heavily used in programs, Java offers a special notation to denote *string literals* in program texts. Each sequence of characters enclosed between double quotes denotes an object of the class of strings. The enclosed sequence is the immutable contents of that string. As an example, the Java expression `"abc"` results in a reference to an object of the class of strings, whose first symbol is `a`, whose second symbol is `b`, and whose last symbol is `c`. In fact, string literals are just a shorthand notation for creating strings by means of the operator **`new`**, using one of the constructors offered by the class of strings. Java further guarantees that different occurrences of the same string literal in a program text all reference the same string object. For that purpose, The Java Virtual Machine manages a pool of string objects during the execution of a Java program. The JVM can then 'intern' string objects, meaning that it adds them to the pool if they are not already in it. By definition, each string literal in the program text is interned.

In addition to a shorthand notation to create strings, the Java language offers the operator `+` as an elegant way to *concatenate strings*. As an example, the Java expression `"this is " + "a string"` results in a reference to a string with contents `"this is a string"`. If we would not use string concatenation, we must wrap at least one of the strings in a string buffer or a string builder. We must then invoke the instance method `append` against that object, supplying the other string as the actual argument. That method extends the prime string buffer or the prime string builder with the given

string. Finally, if we need the contents of the prime string buffer or prime string builder as a string, we must invoke the method `toString` against it.

Class `PrintStream`

Operating systems offer different ways to communicate with end users. One of the simplest mechanisms is inspired on the operating system Unix. It involves three streams: the *standard input stream*, the *standard output stream* and the *standard error stream*. A *stream* is just a sequence of characters without any deeper structure. Programs can read data from the standard input stream, they can write their results on the standard output stream, and they can write error diagnostic messages on the standard error stream.

Typically, all three standard streams are mapped on the *command line interface* in which we can start applications. However, end users may redirect any of these streams to other windows or files. If the environment in which a program executes does not support command line interfaces, that environment usually offers a solution for the standard output stream and the standard error stream. A typical example is a web browser that is executing a *Java applet*. In that context, there is no support for the standard input stream. However, if the running applet writes some information on the standard output stream or on the standard error stream, the web browser will pop up a window displaying that information.

In the Java Application Program Interface, the standard input stream, the standard output stream and the standard error stream are all objects of predefined classes. We use instance methods of those classes to read data from the standard input stream, respectively to write information on the standard output stream and on the standard error stream. Public class variables reference the objects representing the standard input/output streams. These variables are part of the predefined class `System`. That class is definitely not a prototypical example of a class definition in an object oriented programming language. It is a rather weird mix of variable declarations and method definitions.

The public class variable `out` of the predefined class `System` references the standard output stream. The public class variable `err` references the standard error stream. Both streams are objects of the predefined class `PrintStream`. That class belongs to the package `java.io` and offers a large set of instance methods for displaying information of different type. Examples are the overloaded methods `print` and `println` for writing primitive values and objects on the stream against which we invoke them. The method

`print` simply writes out its actual argument. The method `println` turns to a new line after it has written out its argument. As an example, the Java statement `System.out.println("Hello!")` displays the string `"Hello!"` on the standard output stream, and then turns to a new line. Because the variable `out` is a class variable, we must select that variable from the class to which it belongs. This is similar to the way we invoke class methods against the class to which they belong. The statement above thus consists of an invocation of the instance method `println` against the object referenced by `System.out`.

The public class variable `in` of the predefined class `System` references the standard input stream. The static type of this variable is `InputStream`, which is another predefined class that is part of the package `java.io`. The class `InputStream` only offers a small set of rather primitive methods for reading data. In fact, an input stream in Java is just a raw stream consisting of bytes. For that reason, we can only invoke methods for reading bytes against such streams. As an example, the instance method `read` returns the next byte available on the input stream against which we invoke the method. The method returns that byte as a value of type **`int`**. If we have reached the end of the input stream, the method returns the value `-1`.

Java uses a philosophy of wrapping more powerful instruments around primitive streams. Before Java 5, things were rather complex. The first half of the code snippet below illustrates how to read a simple integer value from the standard input stream in Java 1.4. First, we wrap an input stream reader around the standard input stream. Objects of the predefined class `InputStreamReader` read bytes from their underlying input stream, and convert them into characters. Hereafter, we wrap a buffered reader around the input stream reader. Objects of the predefined class `BufferedReader` buffer characters supplied by their underlying reader, such that characters and lines can be read in a much more efficient way. The code snippet then reads an entire line from the buffered reader using the instance method `readLine` applicable to buffered readers. Finally, we use the static method `parseInt` to transform all the characters in that line into an integer value. The predefined wrapper class `Integer` offers the static method `parseInt`.

Since Java 5, the API includes a predefined class `Scanner` whose objects can parse values of primitive types and strings. We can wrap such scanners directly around the standard input stream. The class offers a rich variety of instance methods for reading data of different types. Examples are the instance method `hasNextInt` that returns a boolean

indicating whether the next token in the underlying input stream is an integer, and `nextInt` that returns the value of the next integer token in the underlying input stream. The bottom half of the code snippet below illustrates how to read an integer from the standard input stream using the facilities offered by Java 5. First, we wrap a scanner around the standard input stream. Then, we invoke the instance method `nextInt` against that scanner to return the value of the integer in the first line of the standard input stream.

```
// Old-fashioned code to read an integer from the standard
// input stream.
BufferedReader inputStreamReader =
    new BufferedReader(new InputStreamReader(System.in));
String firstLine = inputStreamReader.readLine();
int myValue = Integer.parseInt(firstLine);

// Recommended code to read an integer from the standard input
// stream since Java 5.
Scanner inputStreamScanner = new Scanner(System.in);
int yourValue = inputStreamScanner.nextInt();
```

1.1.4.5 Main method

A Java program is a collection of classes such as our class of bank accounts. Each class may offer a number of instance methods and class methods. Java has no special construct similar to a main program or a main function in procedural languages such as Pascal and C. We can execute a collection of Java classes as a standalone *application*. In that case, one of the classes must offer a main method. Java also offers facilities to execute a piece of code as an *applet*. Applets run under control of a web browser. In that case, one of the classes must work out some dedicated methods. We do not discuss the development of applets in this book.

If a piece of Java software must run as a standalone application, we must develop a collection of classes one of which defines a main method. Example 4 below shows the definition of a main method to perform some experiments with bank accounts. Notice that we do not pollute our class of bank accounts with a main method. That would jeopardize its applicability in other applications. In the example, the main method therefore belongs to a separate class named `BankApplication`. The body of the main method is kept simple. By now, it must be straightforward to understand. The main method communicates with the end user via the standard input stream and the standard output stream. As soon as we have learned a lot more about Java, we will be able to set up graphical user interfaces. That gives end users more flexibility and more support in communicating with running applications. The Java API contains a large number of classes to develop graphical user

interfaces. The package `java.awt` – the abstract windowing toolkit – is the oldest package of classes for developing graphical user interfaces. These classes are rather primitive and often difficult to use. The package `javax.swing` offers an alternative collection. It builds upon classes in `java.awt`, and tries to simplify the construction of graphical user interfaces.

In Java, the *main method* must obey the following rules. Otherwise, the Java Virtual Machine does not recognize the method as a main method, and the application does not start up.

- The main method must be named `main`. Recall that identifiers in Java are case sensitive. This means that the Java Virtual Machine does not recognize a method named `Main` as a main method.
- The main method must be a static method. At the time an application starts up, there are no objects against which we can invoke methods. Because the main method is the very first method that we invoke, Java requires it to be a static method.
- The main method must have exactly one formal argument. That argument serves to pass information to the running application via the command line interface. The formal argument of a main method must be an array of strings. We discuss arrays in much more detail in Chapter 2. For the moment, we only need to know that entities of array type use one or more pairs of square brackets (`[]`) in their declaration.
- The main method may not return a result. Its result type must be **`void`**. If we define a method named `main` with another result type or with other formal arguments, the Java Virtual Machine does not recognize it as a main method.
- The main method must be a public method of its class.

```
public class BankApplication {

  public static void main(String args[]) {
    BankAccount myAccount;
    Scanner inputStreamScanner = new Scanner(System.in);
    // Create a new bank account with given initial balance.
    System.out.println("Enter initial balance.");
    long initialBalance = inputStreamScanner.nextInt();
    myAccount = new BankAccount(1234567,initialBalance);
    // Deposit a given amount of money.
    System.out.println("Enter deposit amount.");
    long amount = inputStreamScanner.nextInt();
    myAccount.deposit(amount);
```

```
    // Withdraw a given amount of money.
    System.out.println("Enter withdraw amount.");
    amount = inputStreamScanner.nextInt();
    myAccount.withdraw(amount);
    // Display the balance.
    System.out.println("Final balance: "
        + myAccount.getBalance());
  }
}
```

Example 4: Definition of a main method in Java.

Without the support of an Integrated Development Environment (IDE) such as Eclipse for developing your Java programs, we must store the definition of the class of bank accounts and of the class introducing the main method in the same directory. We can then compile both classes using the *Java compiler* that is distributed as an element of the Java Development Kit (JDK). In a command line interface we enter the command `javac BankAccount.java BankApplication.java` for that purpose. As a result we get the files `BankAccount.class` and `BankApplication.class`. These files store the translated versions of both classes as a sequence of bytecode instructions. Hereafter, we can execute the program using the *Java Virtual Machine* that is also part of the Java Development Kit. We must enter the command `java BankApplication` for that purpose. The virtual machine then knows that it must search for a main method in the class `BankApplication`.

1.2 Class Representation

The interface of a class informs its clients how to invoke instance methods and class methods. It further describes what clients can expect when they invoke those methods against individual objects of that class, respectively against the class itself. In the next step in the development of a class, we decide on the information to be stored at the level of individual objects of a class, as well as at the level of the class itself. We also decide *how* to store that information. We therefore say that this step establishes a *representation* for the information to be stored. For the class of bank accounts as specified in Example 3 on page 32, we decide on how to store information concerning the number ascribed to individual accounts, concerning their balance and concerning their blocked state. We also decide how to store information concerning the bank code, the credit limit, and the balance limit at the level of the class itself.

Choosing a proper representation for information to be stored at the level of individual objects and at the level of classes is not an easy task. First of all, the representation must be rich enough to accommodate the implementation of all the methods defined in the interface of the class. The actual choice for a representation may also have a huge impact on *run-time efficiency*. Especially for multi-valued properties, various alternatives for their representation may exist. Each alternative typically has a different impact on performance. As an example, if we decide to collect a set of integer values in a sorted set, insertions are costly but searches are efficient. If we decide to use a sequential list instead, insertions are simple and efficient, but it takes more time to find elements in such a list.

Redundancy is another factor influencing run-time efficiency. We may decide to store information even though we can compute it in terms of other pieces of information. As an example, if we store a series of integer numbers, we may also store redundantly the largest and the smallest number in the series. Obviously, it is much more efficient to return information that we have stored redundantly, especially if the algorithm to compute that information consumes a lot of time. The downside is that adding, removing and updating information takes more time. Each time some piece of information changes, we must also change redundantly stored information derived from it.

The quest for an adequate representation often also involves more technical issues. A typical example is the precision we use to store numerical information. Java offers four different types to store integer values, as well as two floating point types. The Java Application Program Interface offers some additional classes such as `BigInteger` and `BigDecimal`. Moreover, in the class interface we have already decided how to expose information to the clients of your class. The precision we use to store that information cannot be lower than the precision under which we expose it to clients. As an example, we have decided to expose the balance of a bank account as a long integer number. A consequence of that decision is that we cannot use ordinary variables of type **`int`** for its representation.

1.2.1 Instance variables and class variables

Java offers instance variables and class variables to store the information ascribed to individual objects of a class, respectively to the class itself. As for class methods, class variables are qualified **`static`** in their declaration. Instance variables have no such qualification. Example 5 below illustrates the definition of instance variables and class

variables in the class of bank accounts. The declaration of each variable at least includes a name and a type. Access rights, final qualifications and explicit initializations can further complement declarations of instance variables and class variables. We discuss all these aspects in detail in this section.

```
class BankAccount {

      // Definition of method(s) related to the number.

  /**
   * Variable registering the number of this bank account.
   */
  private final int number;

      // Definition of method(s) related to the bank code.

  /**
   * Variable registering the bank code that applies to all
   * bank accounts.
   */
  private static final int bankCode = 123;

      // Definition of methods related to the balance.

  /**
   * Variable registering the balance of this bank account.
   */
  private long balance = 0L;

      // Definition of method(s) related to the credit limit.

  /**
   * Variable registering the credit limit that applies to all
   * bank accounts.
   */
  private static long creditLimit = 0L;

      // Definition of method(s) related to the balance limit.

  /**
   * Variable registering the balance limit that applies to all
   * bank accounts.
   */
  private static final long balanceLimit = Long.MAX_VALUE;

      // Definition of methods related to the blocked state.

  /**
   * Variable registering the blocked state of this bank
   * account.
   */
  private boolean isBlocked = false;

}
```

Example 5: Instance variables and class variables for the class of bank accounts.

In Java, we can freely mix definitions of methods and declarations of variables in the body of a class. In Coding Advice 5 on page 13, we have recommended to group all elements related to a single property. For a typical property, we complement mutators and inspectors for manipulating that property with one or more variables for storing its current value. We further recommend to start a group of elements related to some property with the definitions of all its methods. Declarations of all the variables related to that property then follow the definition of its methods. Example 5 above illustrates this schematically.

Coding Advice 8: In a group of elements related to a single property, definitions of methods precede declarations of variables.

1.2.1.1 Instance variables

In Java, *instance variables* describe information that is stored at the level of each individual object of their class. The declaration of an instance variable at least introduces a name and a type. The name of the variable serves to access that piece of information in the implementation of methods. The type of an instance variable describes the kind of information it stores. In Java, the type of an instance variable is either a primitive type or a class type.

Example 5 above illustrates the declaration of instance variables for the class of bank accounts. The class introduces instance variables to store the number of a bank account, its balance and its blocked state. Notice that a short documentation comment describing the kind of information stored in these variables complements their declaration. However, all these instance variables are private elements of their class. Documentation comments attached to them are therefore far less important than documentation comments complementing public methods.

The types of the instance variables in the class of bank accounts reveals that the number of a bank account is stored in the form of an integer number. The balance of a bank account is stored as a long integer number, and its blocked state in the form of a boolean. The memory allocated to an object of the class of bank accounts thus includes sufficient room to store these three pieces of information. In addition, memory allocated to objects of a class includes room to store some bookkeeping information.

Graphical notation

The *state of an object* reflects the current contents of all its instance variables. Figure 2 illustrates the state of a bank account. We use the *Unified Modeling Language* (UML) to represent objects in a graphical way. UML is a widespread notation for designing complex software systems. It offers all kinds of diagrams and has graphical notations for classes, objects, associations, inheritance, ··· In UML, the *graphical representation of an object* consists of at least two compartments. The top compartment displays the name of the class to which the object belongs. The compartment may also include a name for the object itself.

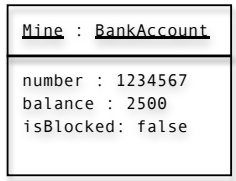


Figure 2: A bank account.

A colon separates the object name in that case from the class name. In graphical representations of objects we must underline both their own name and the name of their class. The bottom compartment displays additional information concerning the object. Although this is not supported in the UML, we use it in this book to display the contents of variables – called attributes in UML – ascribed to the object along with their current value. The compartment must not show the current value of all variables. We may display only those variables that have a relevant role to play in the overall picture.

Notational conventions

The *name* of an instance variable is an identifier, which must reflect as good as possible the information it registers. Java suggests to use the same notational conventions for method names and for variable names. Names of instance variable thus use camel casing that we have discussed on page 12. This explains the spelling for the instance variable `isBlocked` in Example 5 above. The first word "`is`" is written in small letters. The second word "`Blocked`" starts with a capital letter.

Coding Rule 19: Choose names of variables in such a way that they reflect as good as possible the information they store. Use camel casing for their spelling.

Encapsulation

In section 1.1, we have argued that the interface of a class serves to inform its clients what they can expect from the class itself and from its objects. One of the most basic

principles of object oriented programming imposes to hide all details concerning the internals of a class from its clients. This principle is known as *information hiding*: definitions of classes hide irrelevant information from their clients. In the area of object oriented programming, we often use *encapsulation* as a synonym for information hiding. Object oriented programmers *encapsulate* all kinds of technical details in the definition of their classes.

Coding Principle 4: Encapsulate all technical details concerning the internal working of a class in its definition.

We must never expose the representation of objects to the clients of their class. In other words, we never tell the clients of a class how we represent the information stored in each of the objects of that class. This explains the private qualification for all the instance variables in the definition of the class of bank accounts as worked out in Example 5. On page 11 we have already explained that we can only access private elements in the body of their class. Clients of the class of bank accounts are thus not able to manipulate in a direct way the instance variables `number`, `balance` and `isBlocked`.

A first reason to encapsulate the representation of objects is *adaptability*. If we hide all information concerning the representation of objects, we are able to change that representation without affecting client code. Practice has shown that we often need to change the representation of objects. A first kind of change in representation concerns the medium that we use to effectively store the information. We can store information concerning the state of an object in memory. We can also store it on more permanent storage media such as files and databases. As an example, instead of storing the state of all bank accounts in memory, we may decide to store their state in a relational database. Because we do not tell clients of our class that we currently store bank accounts in memory, we can switch to a relational database just by changing aspects in the body of the class of bank accounts. Connecting to relational databases from within Java programs is not simple. We need to study Java Database Connectivity (JDBC) for that purpose. However, it can be done in such a way that the interface of the class of bank accounts is left untouched. If the interface does not change, no changes are needed in client code.

A second kind of change in representation concerns changes in the type of the stored information. Primitive types such as integers and booleans do not suffice to store all kinds of information concerning objects. In Chapter 6 we will discuss how to store information

concerning savings accounts attached to bank accounts or concerning people that have a grant over bank accounts. We then need more complex data structures such as sets, lists and trees to collect savings accounts, respectively grantees. Data structures that we initially choose in such cases are often not adequate to meet certain non-functional requirements. If we initially choose a linked list to collect all grantees, we need a sequential search to find out whether some person has a grant over some account. If a sequential search is not fast enough, we must switch to another data structure, such as a sorted set. Again, if we do not tell clients of our class how we store the grantees for a bank account, we can switch from a linked list to a sorted set just by changing aspects in the body of the class of bank accounts.

Robustness is a second reason not to expose the representation of objects. If we grant clients of our classes direct access to the instance variables of a class, clients are able to register in those instance variables any value that fits their type. Indeed, only the type of a variable restricts its contents. As an example, the instance variable `balance` in Example 5 on page 62 is of type **`long`**. Consequently, Java does not allow us to register other values than long integer values in that variable. In other words, Java programs are robust enough to withstand attempts to register values of the wrong type in a variable. Moreover, if we read the contents of a properly initialized variable, Java guarantees us a value that belongs to the type of that variable. As an example, if we compute the sum of the balances of two bank accounts, we know that we add two long integer values - and not things such as strings or characters.

In addition to their type, we often want to impose additional restrictions on the contents of variables. In the class of bank accounts as worked out in Example 5, we want the balance of each bank account to always be in the range defined by the credit limit and the balance limit. We also want all bank accounts to have positive numbers. If we expose instance variables to the clients of our classes, such additional restrictions are hard, if not impossible to enforce. Indeed, clients can then register any value or the correct type as the new contents of instance variables. We cannot reasonably expect clients of our class to stick to additional restrictions that we wish to impose on the state of the objects of our classes. If we hide instance variables from the clients of our classes, we as designers are in full control. We then only offer clients public methods for manipulating objects of our classes. In the specification and implementation of these methods, we see to it that objects of our classes always satisfy additional restrictions we want to impose on them. As illustrated in Example 3 on page 32, we only offer the instance methods `deposit`,

`withdraw`, and `transferTo` to manipulate the balance of bank accounts. According to their specification, these methods never result in a bank account whose balance is out of range.

Coding Rule 20: Use private instance variables to encapsulate all aspects concerning the representation of objects of a class.

Initialization

Java does not allow us to inspect the contents of a variable before we have given that variable its first value. Java imposes different initialization strategies for different kinds of variables. On page 48 we have discussed the successive steps involved in the creation of a new object. We then stated that the Java Virtual Machine initializes all the properties of a newly created object to the default value of their type. This actually means that the JVM initializes all the instance variables that apply to the newly created object to the default value of their type.

Java offers several alternatives to further *initialize* the instance variables of a class. One of these alternatives is to initialize instance variables in their declaration. Such an initialization actually corresponds to an assignment that the Java compiler embeds at the start of the body of constructors of the class. Java guarantees that compilers embed initializations in the lexical order in which they occur in the definition of the class. Upon entry to each constructor, all the instance variables of the newly created object are thus initialized to the default value of their type. Explicit initializations in declarations of instance variables only get their effect during the execution of constructors.

In Example 5 on page 62, we explicitly initialize the instance variables `balance` and `isBlocked` in their declaration to `0`, respectively to **`false`**. These initializations are not really needed, because they coincide with the default value of their type. However, we believe that it is important to communicate whether or not a proper initial value for an instance variable is available. If we have such a value, we explicitly state it in the declaration of the variable, even if that value happens to coincide with the default value of its type. This rule also applies to class variables that we discuss in section 1.2.1.2 below. In Example 5 we do not explicitly initialize the instance variable `number` in its declaration. This expresses that we do not have a proper initial value for that variable. In section 1.3.3 we discuss how to initialize that variable in the body of a constructor of the class of bank accounts.

Coding Advice 9: Explicitly initialize an instance variable or a class variable, if a proper initial value exists, even if that value coincides with the default value of its type.

Selection

During the execution of an object oriented program we need to read and write the contents of instance variables. *Selecting* instance variables from objects of a class is very similar to invoking instance methods against those objects. Syntactically, the selection of an instance variable has the form `ObjectExpression.instanceVariableName`. The construct stands for the named instance variable of the object resulting from the evaluation of the object expression. As for instance methods, the Java compiler refuses the selection if the class corresponding to the static type of the object expression does not declare a variable with the stated name. The compiler also rejects the code if that instance variable is not accessible in the context in which we try to select it. Finally, if the evaluation of the object expression does not result in an effective object, the Java Virtual Machine throws an exception of type `NullPointerException`.

If we select an instance variable at the left-hand side of an assignment statement, that selection denotes the location allocated to that instance variable. *Assignment statements* register the value at their right-hand side as the new contents of the variable at their left-hand side. If we select an instance variable as part of some expression, that selection stands for the current value of that instance variable. As an example, assume that the variables `myAccount` and `yourAccount` both reference an effective bank account. The assignment statement `myAccount.balance = yourAccount.balance + 100;` then changes the contents of the instance variable `balance` of the bank account referenced by `myAccount`. The new contents is the current contents of the instance variable `balance` of the bank account referenced by `yourAccount`, incremented with `100`.

1.2.1.2 Class variables

In section 1.1.2.3, we saw that class methods or static methods apply to the class itself rather than to individual objects of that class. *Class variables* or *static variables* serve to specify information that is stored at the level of the class itself, rather than at the level of individual objects. Information stored in class variables is shared by all objects of the class. Technically, the memory allocated to individual objects of a class does not include room to store the contents of class variables. Instead, some separate chunk of storage is allocated once and for all to store the contents of all the static variables of a class.

As for declarations of instance variables, the declaration of a class variable involves at least a name and a type. In Coding Rule 19 on page 64 we already stated that identifiers for class variables follow the notational conventions imposed on instance variables. As for class methods, class variables in Java are qualified **`static`** in their declaration. The class of bank accounts in Example 5 on page 62 introduces the class variables `bankCode`, `creditLimit` and `balanceLimit`. All objects of the class of bank accounts thus share information concerning these properties. Notice that we hide all three variables from the clients of our class. They must use public methods such as `getBankCode` and `setCreditLimit` to manipulate these properties.

In discussing instance variables, we used the Unified Modeling Language to denote objects in a graphical way. UML also has a graphical notation for classes. However, that notation displays attributes (variables) and operations (methods) defined at the level of the class. The notation does not serve to display information concerning the current value of class variables. We use graphical representations for classes and other ingredients of software systems to outline their overall structure. We discuss this issue in Part II.

Because all objects of a class share all class variables of their class, we can display information concerning the value of class variables as part of the state of objects of their class. We illustrate this in Figure 3 for the class of bank accounts. The example shows an unnamed object of the class of bank accounts. That object displays the current value of the class variables defined to store the bank code, the credit limit and the balance limit shared by all bank accounts. Notice that UML asks us to underline names of static variables in *graphical representations of objects*.

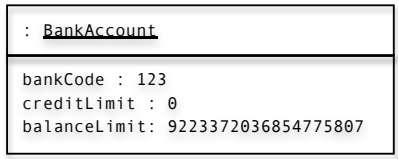


Figure 3: Class variables.

Initialization

As for instance variables, Java offers several constructs to *initialize class variables*. The simplest construct is to initialize class variables in their declaration. If we do not initialize a static variable in its declaration, the default value of its type serves as the initial value for that variable. We again recommend to initialize static variables explicitly in their declaration, if a proper initial value is available. Java further offers *static initialization blocks*

to work out more complex initializations of static variables. We do not discuss this concept in more detail in this book. The Java Virtual Machine initializes class variables as soon as the running application becomes aware of their class. The creation of the first object of a class or the first invocation of a static method offered by a class are examples of events that trigger the initialization of the static variables of that class.

Selection

In Java, we *select* class variables ascribed to a class in the same way we invoke class methods against their class. The construct `ClassName.classVariableName` stands for the named class variable of the given class. The compiler refuses such a selection if that class does not declare the named class variable. The compiler also refuses the selection if the class variable is not accessible at the point of the selection. As an example, the assignment statement `BankAccount.creditLimit = -1000;` registers the value `-1000` as the new contents of the class variable `creditLimit` ascribed to the class `BankAccount`. Java also allows to select class variables via an object of their class. However, as for class methods, we advice against using this facility. In that way, it is a lot easier to distinguish selections of instance variables from selections of class variables.

1.2.1.3 Final qualifications

In section 1.1.2.1 on page 13, we have distinguished between mutable properties and immutable properties. Once an immutable property is initialized, we can no longer change its value during the execution of the application. Java offers *final qualifications* to express that we can no longer change the contents of a variable, once that variable has been given its first and final value. In Example 5 on page 62, we attach a final qualification to the instance variable `number` and to the class variables `bankCode` and `balanceLimit`. Once a bank account has been created and its number is set, it is no longer possible to change the contents of the final instance variable `number` ascribed to that bank account. In the same way, Java does not allow us to change the contents of the final class variables `bankCode` and `balanceLimit` once these variables have been initialized.

Technically, Java refuses assignments to final instance variables outside the bodies of constructors and of instance initialization blocks. Instance initialization blocks are another construct to initialize instance variables. They are similar in nature than static initialization blocks. Assignments to final class variables are illegal outside the scope of static initialization blocks. Java forces us to explicitly assign a value to each final instance variable

and to each final class variable. Moreover, the language only allows us to assign once to final variables. For each final instance variable, we must therefore choose between an explicit initialization in its declaration, or a single assignment to that variable in the body of a constructor or in an instance initialization block. For a final static variable we must either add an explicit initialization to its declaration, or we must initialize that variable in a single assignment in a static initialization block.

Coding Rule 21: Add a final qualification to the declaration of all instance variables and class variables that serve to register immutable properties.

In Java, *final qualifications* also apply to formal arguments and local variables. As an example, we may use final qualifications in the signature of the mutator `transferTo`. The heading of that method would then become `public void transferTo(final long amount, final BankAccount destination)`. As for instance variables, we cannot change final arguments once they have been initialized. This means that Java does not allow any assignments to a final formal argument in the body of its method. The final formal argument `amount` for the above method `transferTo` thus registers the amount of money to be transferred during the entire execution of the method. In a similar way, the final formal argument `destination` refers to the destination account for the entire execution of the method. Final qualifications for formal arguments do not affect clients in any way. Indeed, in section 1.2.2.4 on page 79 we will explain that Java uses "call by value" in binding arguments. In that parameter binding mechanism, a final qualification for a formal argument only has an effect on the local copy of the actual argument, and not on the actual argument itself.

Symbolic constants

Procedural languages such as Pascal and C support the definition of symbolic constants. A *symbolic constant* serves to give a name to a constant literal. In Java, final class variables are close to symbolic constants. Several classes in the Java Application Interface introduce such symbolic constants. `Math.PI` is an example of such a predefined symbolic constant. It introduces the symbolic name `PI` for the double value closest to the mathematical constant π. It is declared in the predefined class `Math` as `public static double PI = …;`. Another example of a predefined symbolic constant is `Long.MAX_VALUE`. This final class variable introduces the symbolic name `MAX_VALUE` for

the largest positive value in the primitive type **long**. It is declared in the predefined wrapper class `Long` as **public static long** `MAX_VALUE = …;`.

Symbolic constants improve the readability of code. An expression such as `(myValue <= Math.PI)` is easier to understand than `(myValue <= 3.1415…)`. The first expression explicitly states that `myValue` is compared with the mathematical constant π. In the second expression, readers of our code can only guess whether it is our intention to compare `myValue` with that mathematical constant. Notice the notational conventions that Java suggests for symbolic constants. The name of a symbolic constant is fully written in upper case. Underscores (_) separate successive words in the name of a symbolic constant as illustrated by `MAX_VALUE`.

Coding Rule 22: Use all upper case in spelling identifiers naming symbolic constants. Separate successive words in such names by underscores.

At first sight, symbolic constants and immutable class properties serve the same purpose. They both denote something that cannot change once their initial value is set. We interpret symbolic constants as names for values that never change. Immutable properties, on the other hand, may change their value from one execution to the next. In other words, symbolic constants are timeless. Immutable properties are only bound to their value until the running application stops executing. Using this interpretation, we can be sure that the symbolic constant `Math.PI` always denotes the same value of primitive type **double**. Replacing all occurrences of that constant in a program with its literal value will not influence the behavior of that program. In the definition of the class of bank accounts of Example 3 on page 32, the bank code shared by all bank accounts is an immutable property. This means that the bank code does not change during the execution of an application that uses our class of bank accounts. However, the bank code may change over time. Successive executions of the same application may thus get different bank codes. This means that we cannot replace all invocations of the inspector `getBankCode` by the literal value it currently returns.

Coding Rule 23: Use symbolic constants for timeless properties. Use immutable properties for values that are constant during a running application, but that may change over time.

1.2.2 Value semantics and reference semantics

In Java, primitive types differ in semantics from class types. Primitive types adopt *value semantics*, whereas class types adopt *reference semantics*. We refer to a type adopting value semantics as a *value type*, and to a type adopting referencing semantics as a *reference type*. Assignments and comparisons have different semantics if they involve elements of value types, respectively of reference types. In Java, the borderline between value semantics and reference semantics is language-defined. Procedural languages such as C and Pascal, as well as some object oriented languages such as C++, have an explicit notion of pointers. In these languages, programmers themselves decide whether a variable has value semantics or reference semantics. Java's decision simplifies programming in a considerable way. The downside is a loss in flexibility for the Java programmer. Other object oriented programming languages offer more flexibility still without introducing an explicit notion of pointers. Eiffel and C# for instance offer two different kinds of classes or types. Ordinary classes in these languages imply reference semantics, whereas so-called expanded classes (Eiffel) or structs (C#) imply value semantics.

Formal arguments of methods in Java act as variables that are initialized each time we invoke their methods. In fact, the binding of an actual argument to its formal argument is identical in semantics to an assignment of that actual argument to the corresponding formal argument. This results in a parameter binding mechanism known as "*call by value*". Java supports no other parameter binding mechanisms. Other object oriented programming languages offer additional parameter binding mechanisms. As an example, both C# and C++ support the parameter binding mechanism known as "call by reference".

1.2.2.1 Value semantics

A variable declared to store elements of a type adopting *value semantics* directly stores a value of that type. In other words, for variables with value semantics, compilers allocate sufficient room to store a value of their type. In Java, value semantics applies to all primitive types. Locations allocated to variables of primitive type **`int`** are thus big enough to store 32-bit integer values. Locations allocated to variables of primitive type **`double`** are big enough to store 64-bit floating point values. For variables of type **`boolean`** a 1-bit location would suffice. Typically, however, Java compilers allocate bigger locations for boolean variables such that the locations allocated to all the variables in a program line up well.

The code snippet below declares and manipulates three variables of primitive type **int**. The top half of **Error! Reference source not found.** shows the contents of the locations allocated to these variables after the Java Virtual Machine has processed all three declarations. Because all three variables are of primitive type **int**, each of these locations serves to store a 32-bit integer value. They each store the initial value supplied in the declaration of the corresponding variable.

```
{
  int myValue   = 10;
  int yourValue = 15;
  int hisValue  = 20;

  myValue  = hisValue;
  hisValue = hisValue + yourValue;
  yourValue = 2*yourValue + 5;

  System.out.println
      ("Expecting true: " + (yourValue == hisValue));
}
```

Assignments involving elements of value types have copy semantics. Given an *assignment* `a = e` of some expression `e` to some variable `a` with value semantics. That assignment registers a copy of the value resulting from the evaluation of the expression `e` at its right-hand side as the new contents of the variable `a` at its left-hand side. The first assignment in the code snippet above thus registers a copy of the contents of the variable `hisValue` as the new contents of the variable `myValue`. From that point on, both copies evolve in a separate way. The next assignment registers `35` – the sum of the variables `hisValue` and `yourValue` – as the new contents of `hisValue`. That assignment does not influence in any way the contents of the variable `myValue`. The bottom half of Figure 4 displays the state of all three variables after the Java Virtual Machine has executed all assignments in the code snippet above.

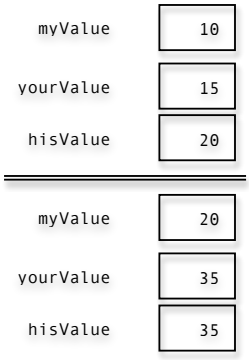


Figure 4: Value semantics.

Comparisons involving elements of value types use the actual values at both sides of their operator. Java offers the operators `==` and `!=` to check whether two expressions evaluate to the same result, respectively to a different result. For expressions of value types, the comparison operators thus directly compare the values resulting from the evaluation of their left-hand

side and their right-hand side operands. In the code snippet above, the comparison `yourValue == hisValue` thus results in comparing the integer values `35` and `35`.

1.2.2.2 Reference semantics

A variable declared to store elements of a type adopting *reference semantics* stores a reference – a pointer – to an object of that type. In other words, for variables with reference semantics compilers just allocate room to store the address of a memory location. In general, objects themselves may occupy several successive memory locations. The address registered in a variable with reference semantics is the address of the first memory location allocated for the referenced object. In Java, reference semantics applies to all class types. Locations allocated to variables of type `BankAccount` thus register a reference to an object of the class of bank accounts. In the same way, a variable of type `Date` – another predefined class in the Java API – registers a reference to a date-object. If a variable with reference semantics does not reference an object, the value **`null`** is registered in its location. We then say that the variable is *not referencing an effective object*.

The code snippet below declares and manipulates three variables of type `BankAccount`. The top half of Figure 5 shows the contents of the locations allocated to these variables after the Java Virtual Machine has processed all three declarations. Because all three variables have reference semantics, each of these locations serves to store the address of a memory location. The variable `myAccount` registers the predefined value **`null`**, which means that it is not referencing an effective bank account at that time. The locations allocated to the variables `yourAccount` and `hisAccount` register the addresses with hexadecimal value `0x100`, respectively `0x108`. These addresses are the starting address of the memory allocated for both bank accounts created in initializing these variables. At this point of the code, the balance of the bank account referenced by `hisAccount` is `20`. In the remainder of this book, we do not illustrate the contents of variables with reference semantics in a numerical way. Instead, as illustrated in Figure 5, we use arrows pointing to objects referenced by such variables.

```
{
  BankAccount myAccount   = null;
  final BankAccount yourAccount = new BankAccount(1234567,15);
  BankAccount hisAccount  = new BankAccount(7654321,20);

  myAccount  = hisAccount;
```

```
  hisAccount.balance =
      hisAccount.balance + yourAccount.balance;

  System.out.println
      ("Expecting true: " + (myAccount == hisAccount));
  System.out.println
      ("Expecting false: " + (myAccount == yourAccount));
}
```

Assignments involving elements of reference types have sharing semantics. Given an *assignment* `a = e` of some expression `e` to some variable `a` with reference semantics. That assignment registers a copy of the memory address – the pointer – resulting from the evaluation of the expression `e` at its right-hand side as the new contents of the variable `a` at its left-hand side. The first assignment in the code snippet above thus registers a copy of the address registered in the variable `hisAccount` as the new contents of the variable `myAccount`. From that point on, both variables thus reference the same bank account. We say that both variables now *share* the same bank account.

The next assignment registers `35` – the sum of the balances of the bank accounts referenced by `hisAccount` and `yourAccount` – as the new balance of the bank account referenced by the variable `hisAccount`. That assignment has value semantics, because it involves elements of primitive type **`long`**. Because the variables `hisAccount` and

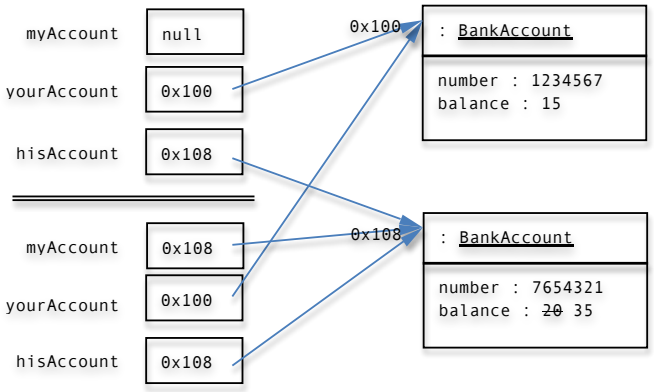


Figure 5: Reference semantics.

`myAccount` share the same bank account at the time of that assignment, the balance of the bank account referenced by `myAccount` also changes to `35`. The bottom half of Figure 5 displays the state of all three variables after the Java Virtual Machine has processed both assignments.

Comparisons involving elements of reference types use the references – the memory addresses – at both sides of their operator. Java offers the same operators `==` and `!=` to compare objects resulting from the evaluation of expressions of reference types. In the code snippet above, the comparison `myAccount == hisAccount` thus results in comparing the addresses `0x108` and `0x108`. When supplied with expressions of reference types, comparison operators never compare the internal states of the referenced objects. The comparison `myAccount == yourAccount` at the end of the code snippet above thus yields **`false`**. Even if the bank accounts referenced by both variables would have the same number, the same balance and the same blocked state, the comparison would still yield **`false`**.

In the code snippet above, we have attached a final qualification to the variable `yourAccount`. We already explained in section 1.2.1.3 on page 70 that such a

qualification prohibits assignments to that variable following its declaration. Java compilers would thus reject the assignment `yourAccount = myAccount` in the code snippet above. They would even reject the assignment `yourAccount = yourAccount`. However, a final qualification for a variable with reference semantics does not prohibit changes to the referenced object via that variable. In the code snippet above, the assignment `yourAccount.balance = 1000` would thus be perfectly legal. That assignment does not change the contents of the variable `yourAccount` itself. It changes the state of the bank account referenced by that variable.

1.2.2.3 Boxing and unboxing

In the previous section, we explained that values of primitive types adopt value semantics. However, the Java API offers a set of so-called *wrapper classes* that offer support to approach integer numbers, floating point numbers, characters and booleans by means of reference semantics. Each primitive type in Java therefore has a corresponding wrapper class in the Java API. The wrapper class for the primitive type **`int`** is `Integer`, the wrapper class for **`float`** is `Float`, ··· Each object of a wrapper class holds a value of the corresponding primitive type as its state. We say that such objects *wrap* – enfold – values of the corresponding primitive type. In the code snippet below, a new object of the wrapper class `Integer` is created registering the value `20` of type **`int`** as its state. We then use that object to initialize the variable `myWrappedValue`. Because this variable has reference semantics, it references the newly created wrapper object.

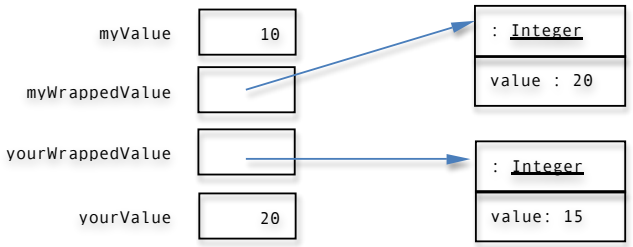
```
{
  int myValue = 10;
  Integer myWrappedValue = new Integer(20);

  // Boxing
  Integer yourWrappedValue = myValue+5;

  // Unboxing
  int yourValue = myWrappedValue;
}
```

Since version 1.5, Java supports the concepts of boxing and unboxing. These concepts simplify transitions from values of primitive types to objects of their corresponding wrapper class, and vice versa. *Boxing* applies when we assign a value of a primitive type to an entity declared to reference objects of the corresponding wrapper class. The execution of such an assignment implies the creation of a new object of that wrapper class. The Java Virtual Machine automatically creates that object and initializes it

with the primitive value to be assigned. We say that the JVM puts the primitive value in a box, hence the term *boxing*. In the code snippet above, the first assignment illustrates the concept of boxing. As a result of that assignment the variable `yourWrappedValue` references a newly created object of the wrapper class `Integer`. That object internally



stores the value 15. Figure 6 illustrates the result of that assignment.

Unboxing is the reverse of boxing. Unboxing applies when we assign an object of a wrapper class to an entity declared to store values of the corresponding primitive type. In this case, the Java Virtual Machine actually assigns the value stored in the object of the wrapper class to the variable of primitive type. We say that the JVM takes the value stored in the wrapper object out of its box, hence the term *unboxing*. In the code snippet above, the last assignment illustrates the concept of unboxing. That statement assigns the value `20` – the value wrapped in the object referenced by `myWrappedValue` - to the

Figure 6: Boxing/unboxing.

variable `yourValue` of primitive type **`int`**. Figure 6 illustrates the result of this assignment.

1.2.2.4 Call by value

Each time we invoke a method, the Java Virtual Machine must bind all actual arguments involved in that invocation to the formal arguments of that method. Hereafter, the JVM can properly execute the body of the invoked method. Java only supports the simplest parameter binding mechanism in programming languages known as "*call by value*". This binding mechanism initializes each formal argument with a copy of the corresponding actual argument. In fact, the binding of an actual argument to the corresponding formal argument in Java fully corresponds with the semantics of an assignment of the actual argument to the corresponding formal argument.

For formal arguments of primitive type, "call by value" means an initialization of the formal argument with a copy of the primitive value resulting from the evaluation of the corresponding actual argument. The class of bank accounts from Example 3 on page 32 introduces the instance method `transferTo` involving a formal argument `amount` of primitive type **`long`**. In the code snippet below, we invoke that method supplying `myWeeklyBudget` as the corresponding actual argument. For that invocation, the Java Virtual Machine thus initializes the formal argument `amount` with a copy of the contents of `myWeeklyBudget`. **Error! Reference source not found.** illustrates that binding.

```
{
  BankAccount myAccount = new BankAccount(1234567,1500);
  BankAccount yourAccount  = new BankAccount(7654321,2000);
  long myWeeklyBudget = 600;

  myAccount.transferTo(myWeeklyBudget,yourAccount);
}
```

For formal arguments of class type, "call by value" means an initialization with a copy of the reference to the object resulting from the evaluation of the corresponding actual argument. The instance method `transferTo` involves a formal argument `destination` of type `BankAccount`. For the invocation of the method in the code snippet above, the Java Virtual Machine thus initializes that formal argument with a copy of the reference stored in the variable `yourAccount`. The formal argument `destination` and the actual argument `yourAccount` then share the same bank account.

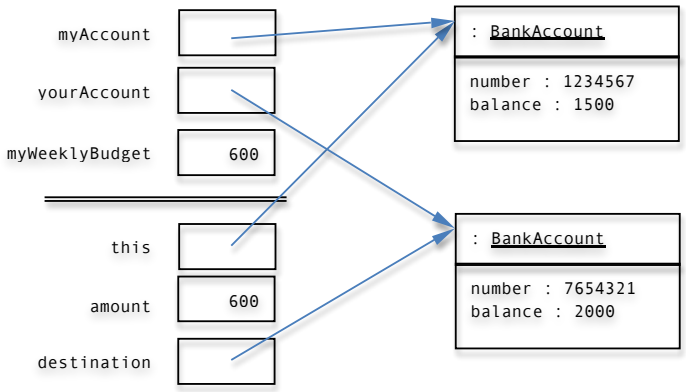


Figure 7: Call by value.

In section 1.1.2.3 on page 18, we have emphasized that instance methods involve an implicit argument that is bound to the object against which the method is invoked. In section 1.3, we explain that we can access that implicit argument in the body of an instance method via the predefined variable **this**. For the invocation of the instance method `transferTo` below, the Java Virtual Machine thus initializes the implicit argument **this** with a copy of the reference stored in the variable `myAccount`. Figure 7 illustrates the binding of both formal arguments with both bank accounts.

Figure 7 displays the contents of variables and formal arguments upon entry to the instance method `transferTo`. Upon exit from that method, the Java Virtual Machine releases the locations allocated to the formal arguments. It also releases locations allocated to local variables in the body of that method, if any. Figure 8 displays the contents of the variables `myAccount`, `yourAccount` and `myWeeklyBudget` upon returning from the method `transferTo`. Because the body of the method operates on a private copy of the amount to be transferred, changes to that formal argument do not influence the corresponding actual argument. The method `transferTo` also changes the balances of both bank accounts involved in it. These changes are visible, because formal arguments with reference semantics share the objects referenced by their corresponding actual arguments. Figure 8 shows that the balance of the account referenced by `myAccount` has been decreased to `900`. It further shows that the balance of the account referenced by `yourAccount` has been increased to `2600`.

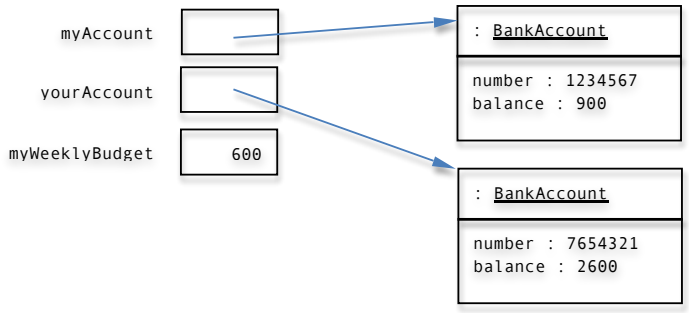


Figure 8: Call by value.

The semantics of "call by value" in binding actual arguments to formal arguments, equally apply to the result returned by instance methods and class methods. In other

words, methods in Java always *return* their result *by value*. Each non-void method thus returns a copy of the result computed in the body of that method. If a method returns a value of primitive type, it returns a copy of the primitive value resulting from the execution of its body. If a method has a class type as its result type, the method returns a copy of the reference to the object resulting from the execution of its body.

1.2.3 Primitive types

A programming language must somehow offer a number of *primitive types*. Such a type defines a set of primitive values complemented with operators to perform computations with these values. Primitive types serve as basic building blocks in defining more complex types. In this chapter, we have defined the type of bank accounts in terms of the primitive types **`int`**, **`long`** and **`boolean`**. Indeed, the state of a bank account consists of its number represented as an integer value, its balance represented as a long integer value, and its blocked state represented as a boolean. In section 1.3, we will use operators on primitive values in implementing the methods applicable to bank accounts. As an example, the body of the instance method `deposit` uses the operator + to compute the sum of the balance of its prime bank account and the given amount of money.

A primitive type is much like a class. They both describe a collection of elements – objects or values. Operations applicable to primitive values – methods or operators – complement that collection of elements. Java does not attempt to unify classes and primitive types in a single concept. The language features the definition of eight primitive types. We discuss each of them briefly in this section. Other object oriented programming languages introduce primitive types as predefined classes. Examples of such languages are Eiffel and C#. In these languages, primitive types correspond to predefined expanded classes (Eiffel), respectively to predefined structs (C#). These classes are special in the sense that not all their features are worked out in the usual way. An example are operators applicable to primitive values. They are offered as so-called native methods of their class, because it is impossible to work out an implementation for these methods. Instead, they are more or less directly mapped on the instruction set of the underlying machine. Languages such as Eiffel and C# further introduce dedicated constructs to support specific aspects of primitive types. Examples are constructs denoting literals, which become shorthand notations for invoking constructors.

1.2.3.1 Integral types

Java offers the primitive types **byte**, **short**, **char**, **int** and **long** to work with integer numbers. We briefly discuss all of them in this section, except for the type **char**. We postpone the discussion of that type in section 1.2.3.3. The primitive type **int** supports 32-bit signed integer numbers in the range [-2 147 483 648 .. 2 147 483 647]. The primitive type **long** supports 64-bit signed integer numbers in the range [-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807]. The symbolic constants `MIN_VALUE` and `MAX_VALUE` denote the boundary values for the integral types **int** and **long**. These constants are defined in the corresponding wrapper classes `Integer`, respectively `Long`. The symbolic constant `Integer.MIN_VALUE` thus denotes the most negative value in the type **int**. The symbolic constant `Long.MAX_VALUE` denotes the largest value in the type **long**.

The primitive type **byte** supports 8-bit signed integer numbers in the range [-128 .. 127]. The primitive type **short** supports 16-bit signed integer numbers in the range [-32 768 .. 32 767]. Both types are not really intended for numerical computations. Java programmers use values of primitive type **byte** or **short** in manipulating streams of bits. If we nevertheless use a byte value or a short value as an operand in some integer computation, the Java Virtual Machine first promotes that value to a value of type **int** or of type **long**. The actual type depends on the type of the other operands involved in the computation. If at least one of the operands involved in a computation is of type **long**, the JVM promotes operands of type **byte** and **short** to type **long**. Otherwise, it promotes values of type **byte** and **short** to type **int**.

Java not only establishes the range of integer values of different types. The language also imposes the 2-complement representation on all integer values. In this way, we can easily port Java programs from one platform to another. Indeed, Java guarantees that all computations with integer values have the same result on all possible platforms. Other programming languages, such as C++, use platform-dependent sizes and platform-dependent representations for primitive types. In such languages, a program running on one platform may use 32 bits for integer values, whereas that same program may use 64 bits when executed on another platform. Programs written in such languages thus adjust themselves to the specific characteristics of the executing platform. Such programs are harder to port from one platform to another. However, their execution is usually much more efficient.

Literals

An *integer literal* denotes a particular integer value. An integer literal is of type **long** if it has a suffix `l` or `L`. Otherwise the literal is of type **int**. The integer literal `123L` thus denotes a value of type **long**, whereas the literal `123` denotes a value of type **int**. Java does not offer a syntax for denoting literals of type **byte** or type **short**. Whenever we assign an integer literal to a variable of type **byte** or of type **short**, however, the Java Virtual Machine automatically narrows that literal to an 8-bit integer, respectively to a 16-bit integer.

We can write integer literals in decimal base, in hexadecimal base or in octal base. A decimal number is either `0` or starts with a non-zero digit. An octal number starts with the zero digit, and is followed by one or more octal digits – digits in the range `0..7`. The octal literal `026` is thus identical to the literal `22` in decimal base. An hexadecimal number starts with `0x` or `0X`, and is followed by one or more hexadecimal numbers – digits in the ranges `0..9`, `A..E` or `a..e`. The hexadecimal literal `0x16` is thus identical to the literal `22` in decimal base, and the literal `0XF2` is identical to the literal `242` in decimal base. The Java compiler rejects integer literals that are not in the range of their type. The integer literal `2147483650` is thus illegal, because it exceeds the largest possible value in the primitive type **int**.

Operators

Java supports the following well-known operators for integral types: `+` (unary plus), `-` (unary minus), `+` (binary addition), `-` (binary subtraction), `*` (multiplication), `/` (integer division), and `%` (integer remainder). An integer division yields the integer part of the division of its operands. `25/7` thus yields `3`. The result of an integer division is non-negative, if both operands have the same sign. It is non-positive if the operands have opposite sign. `9/4` thus yields `2`, whereas `-9/4` yields `-2`. The integer remainder always yields a result such that `(a/b)*b+(a%b)` is equal to `a`. `9%4` thus yields `1`, whereas `-9/4` yields `-1`.

We can compare integer values using the relational operators `==` (equality), `!=` (inequality), `<` (less than), `<=` (less than or equal), `>` (greater than), and `>=` (greater than or equal). All these operators take two operands, and yield a boolean as their result.

In Java, we can also approach integer values as sequences of bits. The language offers the following shift operators: `<<` (left shift), `>>` (signed right shift), and `>>>`

(unsigned right shift). In a signed right shift, the sign bit is inserted at the left-hand side. In an unsigned right shift, a zero bit is inserted at the left-hand side. `0X80000000 >> 2` thus yields `0XE0000000`, whereas `0X80000000 >>> 2` yields `0X20000000`.

The language further offers the operators ~ (bitwise complement), & (bitwise and), | (bitwise inclusive or), and ^ (bitwise exclusive or) to manipulate streams of bits. The bitwise complement operator yields an integer in which each bits is the opposite of the corresponding bit in its operand. The binary bitwise operators yield an integer in which each bit is obtained by and-ing, respectively or-ing or xor-ing the corresponding bits in both operands.

The final operators for integral types are ++ (increment) and -- (decrement). Both operators only apply to entities of an integral type. We can specify them in prefix or in postfix to their operand. The increment operator increments the value stored in the entity against which we apply it. The decrement operator decrements the value of its entity. When applied in prefix, the operator first increments or decrements its entity. The operator then returns the new contents of that entity as its result. When applied in postfix, the operator still increments or decrements its entity. However, in this case the operator returns the old contents of its entity s its result. Assuming the contents of some variable x is 5, the expression `x-- + --x` thus yields `8 (5+3)`. At the same time, the contents of the variable x changes to `3`.

In Java, *assignment* itself is also an operator. The evaluation of an assignment results in the value assigned to the variable at its left-hand side. This means that we can use an assignment anywhere Java expects an expression. An example is the statement `x=y=z+3;` in which several assignments occur in a single statement. The assignment operator evaluates from right to left. In the example, the value resulting from the expression `z+3` is thus first assigned to the variable `y`. The value resulting from the evaluation of its right-hand side – `z+3` – is the resulting value of that assignment. In the example, that result is subsequently assigned to the variable `x`. That assignment in turn results in the assigned value. At this point, the execution of the statement terminates and the Java Virtual Machine throws away the value resulting from the last assignment.

We can combine assignments with all numerical binary operators leading to the notion of *composed assignments*. The left-hand side of a composed assignment at the same time acts as the left-hand operand of its binary operator. The r ight-hand side of a

composed assignment operator is actually the right-hand operand of its binary operator. The result of the binary operator is assigned to the variable at the left-hand side. An example of a composed assignment is `x+=8`. This expression first computes `x+8`. The result of that addition is subsequently assigned to the variable `x`. Each composed assignment is a shorthand for an assignment involving the binary operator at its right-hand side. The expression `x+=8` is thus equivalent with the expression `x=x+8`.

Overflow

Additions, subtractions and multiplications involving integer values can *overflow*. This means that their result may be outside the range of the integral type. The increment and decrement operators can also overflow. The Java Virtual Machine does not signal such overflows. Instead, it changes the result such that it fits in the range of the integral type involved. Technically, the result of such a computation is taken from the low-order bits of that result represented in some sufficiently large two's-complement format. As an example, the expression `Integer.MAX_VALUE+1` yields `Integer.MIN_VALUE`. In each integral type, `MIN_VALUE` has no corresponding positive value in the range of that type. Therefore, the expression `-Integer.MIN_VALUE` yields `Integer.MIN_VALUE`.

The operators integer division and integer remainder may result in a division by `0`. In that case, the Java Virtual Machine throws an exception of type `ArithmeticException`. In Chapter 3, we explain that exceptions interrupt the normal flow of control. As soon as an exception is thrown during the execution of a Java program, the Java Virtual Machine starts searching for a catcher to deal with the problem signaled by that exception. If the catcher is able to solve the problem – if the execution of that catcher does not throw another exception – the program resumes its execution with the instructions following that catcher.

1.2.3.2 Floating point types

Conceptually, floating point numbers are of the form $\pm 0.\texttt{mmmm}^{\texttt{ee}}$ and involve a mantissa `mmmm` and an exponent `ee`. Java offers the primitive type **`float`** defining floating point numbers in single precision, and the primitive type **`double`** defining floating point numbers in double precision. The type **`float`** uses 32 bits. Its largest positive value is 0.340282347e+39, and its smallest positive number is 0.140239846e-44. The type **`double`** uses 64 bits. Its largest positive number is 0.179769313486231570e+309, and its smallest positive number is

0.4940656458412465441e-325. The symbolic constants `MIN_VALUE` and `MAX_VALUE` of the corresponding wrapper classes define these boundary values. Both floating point types in Java use the IEEE 754 standard for representing floating point numbers. In this way, Java guarantees that all programs involving floating point calculations behave the same on all platforms.

Floating point numbers serve to approximate real numbers as defined in mathematics. The mathematical set of real numbers is infinite and contiguous. Computer programs approximate a real number with the closest floating point number. Because the size of the mantissa is fixed for all the values of a floating point type, floating point numbers do not have the same coverage all over their range. As an example, there are much more floating point numbers between 0.0 and 1.0 than there are floating point numbers between 2000.0 and 2001.0.

Computations with floating point numbers are tedious. They are in fact a discipline on their own. In this book, we do not intend to study floating point computations in detail. We only want to draw attention to the potential loss in precision in floating point computations. Let's assume for a moment that we use 32-bit floating point numbers to represent the balance of a bank account. Consider then a bank account with a balance of 0.12345679e+10, and a deposit of 1317.24 to that account. The resulting balance of that account should be 0.123456921724e+10. However, because of the limited number of digits in the mantissa of 32-bit floating point numbers, the actual balance is 0.12345692e+11. Instead of having deposited an amount of 1317.24, we have only effectively deposited an amount of 1300 to that account. Needless to say that such a behavior is unacceptable in a professional software system supporting banking transactions.

Literals

Java offers *floating point literals* for both floating point types. If a floating point literal has suffix `f` or `F`, it is a literal of type **`float`**. If a floating point literal has no suffix, or it has `d` or `D` as its suffix, the literal is of type **`double`**. The literal `0.1F` thus denotes a value of type **`float`**. The literals `0.1` and `0.2D` both denote a value of type **`double`**. We can represent floating point numbers in decimal base and in hexadecimal base. In general, a floating point literal involves a whole-number part, a decimal point, a fractional part, an exponent and a type suffix. For floating point numbers in decimal base, the exponent starts with the letter `e` or `E` and the mantissa only uses digits. For floating point numbers in

hexadecimal base, the exponent starts with `p` or `P` and the mantissa uses the spelling rules for integer numbers in hexadecimal format. Examples of well-formed floating point literals are `23.456E-3`, `0.5678` and `0xABCp5`. As for integer literals, the Java compiler rejects floating point literals that are not in the range of their type.

Operators

Most of the operators applicable to integer values also apply to floating point values. First of all, the following numerical operators also apply to values of both floating point types: `+` (unary plus), `-` (unary minus), `+` (binary addition), `-` (binary subtraction), `*` (multiplication), `/` (floating point division), `%` (floating point remainder), `++` (increment), and `--` (decrement). Computations involving floating point numbers adhere to the IEEE 754 standard, except for the floating point remainder. Java has chosen not to follow the IEEE standard for the latter operator, such that the operator yields comparable results when applied to integer values and floating point values. For binary operators involving an integral operand and a floating point operand, the integral operand is first promoted to the floating point type of the other operand. If one operand of an operator is of type **`float`** and the other operand is of type **`double`**, the former operand is first promoted to type **`double`**.

The relational operators `==` (equality), `!=` (inequality), `<` (less than), `<=` (less than or equal), `>` (greater than), and `>=` (greater than or equal) equally apply to floating point values. Contrary to values of integral types, Java does not let us approach floating point values as streams of bits. Therefore, none of the bitwise operators apply to values of type **`float`** or **`double`**.

Overflow and underflow

Computations involving floating point numbers may result in overflow. A computation involving floating point numbers *overflows* if its result is not in the range of its type. The IEEE standard for floating point numbers introduces special values to represent positive infinity and negative infinity. The wrapper classes `Float` and `Double` introduce the symbolic constants `NEGATIVE_INFINITY` and `POSITIVE_INFINITY` denoting these special values. Whenever a numerical computation involving floating point numbers overflows, its result is set to positive infinity if the overflow occurs at the positive side of the spectrum of floating point numbers. Its result is set to negative infinity if the overflow occurs at the negative side. As an example, the expression `x/0.0` yields positive infinity if

x is positive, and negative infinity if x is negative. Floating point operations thus never throw exceptions, not even in the case of a division by zero.

A computation involving floating point numbers *underflows* if its result is too close to zero to fit in the range of its type. The IEEE standard for floating point numbers distinguishes between positive zero, denoted `+0.0`, and negative zero, denoted `-0.0`. Positive zero and negative zero represent the same floating point value, meaning that the expression `(+0.0 == -0.0)` yields true. However, for computations such as `1.0/+0.0` and `1.0/-0.0`, the sign is significant. The former yields positive infinity, whereas the latter yields negative infinity.

The result of some computations involving floating point numbers is undefined. The IEEE standard for floating point numbers introduces the special value *Not-a-Number* abbreviated *NaN*. The wrapper classes `Float` and `Double` introduce the symbolic constant `NaN` denoting this special value. Examples of computations that yield `NaN` are `0.0/0.0` and `Float.POSITIVE_INFINITY/Float.NEGATIVE_INFINITY`. Moreover, all numerical operators for which at least one of the operands is `NaN` yield `NaN`. The special value `NaN` is unordered, meaning that the relational operators <, <=, >, and >= yield false if at least one of their operands is `NaN`. The relational operator == yields false if at least one of its operands is `NaN`. The relational operator != yields true if at least one of its operands is `NaN`. This means that for some floating point variable x storing `NaN` as its contents, the comparison `x == x` yields **`false`**, whereas the comparison `x != x` yields **`true`**!

1.2.3.3 Characters

The type **`char`** in Java defines 16-bit characters that use *Unicode* for their internal representation. Java programs themselves are also written using the Unicode character set. Originally, the Unicode standard was designed as a fixed-width 16-bit character encoding. Later versions have extended the standard to allow for characters whose representation requires more than 16 bits. Such characters are called *supplementary characters*. To represent the complete range of characters using only 16-bit units, Unicode introduces an encoding called UTF-16. In this encoding, representations for supplementary characters use 16-bit pairs. In Java, the type **`char`** itself does not include the supplementary characters.

Literals

The simplest way to denote a character in Java source code is to enclose it in single quotes. Examples of such *character literals* are `'a'`, `'0'`, `'#'`, `'é'`, and `'Ω'`. Java offers escape sequences to denote some non-graphic characters such as backspace (`'\b'`), horizontal tab (`'\t'`), carriage return (`'\r'`), single quote (`'\''`), and double quote (`'\"'`). Character literals can also be denoted by means of their hexadecimal value as established by the Unicode standard. Examples of such character literals are `'\u03a9'` and `'\uFFFF'`. Finally, for reasons of compatibility with the programming language C, it is also possible to denote character literals using their octal value. Examples of such literals are `'\036'` and`'\177'`.

Operators

In Java, the type **`char`** is an integral type. This means that all the operators applicable to the primitive types **`byte`**, **`short`**, **`int`** and **`long`** equally apply to values of primitive type **`char`**. As an example, the expression `('a' < '#')` checks whether the internal code for the character `'a'` is less than the internal code for the character `'#'`. As another example, the expression `('a' + '\u0004')` yields the character whose internal code is 4 higher than the internal code of the character `'a'`. In Unicode, that character is the character `'e'`.

1.2.3.4 Booleans

The type **`boolean`** involves just two elements, denoted **`true`** and **`false`**. The elements of type boolean are unordered. From the relational operators, only the equality operator (==) and the inequality operator (!=) therefore apply to boolean values. Java offers a set of operators specifically intended to manipulate boolean values.

- The unary operator ! (logical complement or negation) returns **`true`** when applied to **`false`**, and **`false`** when applied to **`true`**.
- The binary operator & (unconditional and) returns **`true`** if both its operands are **`true`**, and **`false`** otherwise. The binary operator && (conditional and) returns the value of its right-hand operand if its left-hand operand is **`true`**. It returns **`false`** if its left-hand operand is **`false`**.
- The binary operator | (unconditional inclusive or) returns **`false`** if both its operands are **`false`**, and **`true`** otherwise. The binary operator || (conditional

inclusive or) returns the value of its right-hand operand if its left-hand operand is **false**. It returns **true** if its left-hand operand is **true**.

- The binary operator ^ (exclusive or) returns **true** if its operands have different values, and **false** if they have the same value.
- The ternary operator ? : (conditional operator) returns the value of its second operand if its first operand is **true**. It returns the value of its third operand if the value of its first operand is **false**. In the former case, the third operand is not evaluated. In the latter case, the value of the second operand is not computed. Only the first operand must be of type **boolean**. The second and the third operand must have a common super type, which acts as the result type of the expression. As an example, the expression `(amount>0 ? myAccount : yourAccount)` returns `myAccount` if `amount` is positive. It returns `yourAccount` if amount is negative or zero.

The conditional operators && and || do not evaluate their right-hand operand, if they can derive their result from the evaluation of their left-hand operand. This simplifies the coding of boolean expressions, in which the evaluation of the right-hand operand may fail under certain conditions. As an example, consider the expression `((myInt != 0) && (yourInt/myInt == 0))`. In case the integer variable `myInt` is zero, that expression yields **false** without evaluating its right-hand operand. If we would use the unconditional operator & instead, the Java Virtual Machine always evaluates the right-hand operand. In the example, that evaluation would throw `ArithmeticException` in case `myInt` is zero. Because the conditional operators && and || are also slightly more efficient, there is no reason to use the operators & and | in boolean expressions. Java offers the unconditional operators & and | for manipulating sequences of bits, as we discussed in section 1.2.3.1.

Coding Rule 24: Use the conditional operators && and || to combine boolean subexpressions instead of the unconditional operators & and |.

1.3 Class Implementation

The specification of a class is the contract between the clients of that class and its designers. In the next step in the development of a class, the designers fulfill their part of the contract. They work out implementations for all the methods – instance methods, class methods and constructors – specified in the interface of the class. The

implementation of a method instructs the machine how to achieve the stated effects. In Java, the implementation of a method is worked out in its body. It consists of a sequence of statements that the Java Virtual Machine executes each time the method is invoked. These statements consult and change information stored at the level of the class itself and at the level of individual objects.

Java integrates aspects of specification, representation and implementation of a class in a single file. We refer to that all-embracing description as the *definition of a class*. The file in which we store the definition of a class must have the same name as its class complemented with a suffix `.java`. Example 6 below lists the full definition of the class of bank accounts. The file in which we store that definition must have `BankAccount.java` as its name. As mentioned before, javadoc extracts all documentation comments from that file. It produces an HTML file displaying the specification of the defined class to all potential clients. The Java compiler also processes the file storing the definition of a class. The compiler produces a translation in terms of bytecode instructions. It stores that translation in a file named after the translated class and complemented with a suffix `.class`. The Java Virtual Machine processes such class files.

```
/**
 * A class of bank accounts involving a bank code, a number,
 * a credit limit, a balance limit, a balance and a blocking
 * facility.
 *
 * @version 2.0
 * @author  Eric Steegmans
 */
class BankAccount {

  /**
   * Initialize this new bank account with given number, given
   * balance and given blocked state.
   *
   * @param   number
   *          The number for this new bank account.
   * @param   balance
   *          The balance for this new bank account.
   * @param   isBlocked
   *          The blocked state for this new bank account.
   * @post    If the given number is not negative, the initial
   *          number of this new bank account is equal the
   *          given number. Otherwise, its initial number is
   *          equal to 0.
   * @post    If the given balance is not below the credit
   *          limit and not above the balance limit, the
   *          initial balance of this new bank account is equal
   *          to the given balance. Otherwise, its initial
   *          balance is equal to 0.
```

```
 * @post    The initial blocked state of this new bank
 *          account is equal to the given flag.
 */
public BankAccount
    (int number, long balance, boolean isBlocked)
{
  if (number < 0)
    number = 0;
  this.number = number;
  setBalance(balance);
  setBlocked(isBlocked);
}

/**
 * Initialize this new bank account as an unblocked account
 * with given number and given balance.
 *
 * @param   number
 *          The number for this new bank account.
 * @param   balance
 *          The balance for this new bank account.
 * @effect  This new bank account is initialized with the
 *          given number as its number, the given balance as
 *          its balance, and false as its blocked state.
 */
public BankAccount(int number, long balance) {
  this(number,balance,false);
}

/**
 * Initialize this new bank account as an unblocked account
 * with given number and zero balance.
 *
 * @param   number
 *          The number for this new bank account.
 * @effect  This new bank account is initialized with the
 *          given number as its number and zero as its
 *          initial balance.
 */
public BankAccount(int number) {
  this(number,0);
}

/**
 * Return the number of this bank account.
 *   The number of a bank account serves to distinguish that
 *   account from all other bank accounts.
 */
@Basic @Immutable
public int getNumber() {
  return this.number;
}

/**
 * Variable registering the number of this bank account.
 */
private final int number;
```

```
/**
 * Return the bank code that applies to all bank accounts.
 *   The bank code identifies the bank to which all bank
 *   accounts belong.
 */
@Basic @Immutable
public static int getBankCode() {
  return bankCode;
}

/**
 * Variable registering the bank code that applies to all
 * bank accounts.
 */
private static final int bankCode = 123;

/**
 * Return the balance of this bank account.
 *   The balance of a bank account expresses the amount of
 *   money available on that account.
 */
@Basic
public long getBalance() {
  return this.balance;
}

/**
 * Check whether this bank account has a higher balance than
 * the given amount of money.
 *
 * @param   amount
 *          The amount of money to compare with.
 * @return  True if and only if the balance of this bank
 *          account is greater than the given amount.
 */
public boolean hasHigherBalanceThan(long amount) {
  return getBalance() > amount;
}

/**
 * Check whether this bank account has a higher balance than
 * the other bank account.
 *
 * @param   other
 *          The bank account to compare with.
 * @return  True if and only if the other bank account is
 *          effective, and if this bank account has a higher
 *          balance than the balance of the other bank
 *          account.
 */
public boolean hasHigherBalanceThan(BankAccount other) {
  return
     ( (other != null)
    && (this.hasHigherBalanceThan(other.getBalance())) );
}

/**
 * Deposit the given amount of money to this bank account.
```

```
     *
     * @param   amount
     *          The amount of money to be deposited.
     * @post    If the given amount of money is positive, and if
     *          the old balance of this bank account is not above
     *          the balance limit decremented with the given
     *          amount of money, the new balance of this bank
     *          account is equal to the old balance of this bank
     *          account incremented with the given amount of
     *          money.
     */
    public void deposit(long amount) {
      if ( (amount > 0)
         && (getBalance() <= getBalanceLimit() - amount) )
        setBalance(getBalance()+amount);
    }

    /**
     * Withdraw the given amount of money from this bank account.
     *
     * @param   amount
     *          The amount of money to be withdrawn.
     * @post    If the given amount of money is positive, and if
     *          this bank account is not blocked, and if the old
     *          balance of this bank account is not below the
     *          credit limit incremented with the given amount of
     *          money, the new balance of this bank account is
     *          equal to the old balance of this bank account
     *          decremented with the given amount of money.
     */
    public void withdraw(long amount) {
      if ( (amount > 0) && (! isBlocked())
        && (getBalance() >= getCreditLimit() + amount) )
          setBalance(getBalance()-amount);
    }

    /**
     * Transfer the given amount of money from this bank account
     * to the given destination account.
     *
     * @param   amount
     *          The amount of money to be transferred.
     * @param   destination
     *          The bank account to transfer the money to.
     * @effect  If the given destination account is effective and
     *          not the same as this bank account, and if this
     *          bank account is not blocked, and if the old
     *          balance of this bank account is not below the
     *          credit limit incremented with the given amount of
     *          money, and if the old balance of the given
     *          destination account is not above the balance
     *          limit incremented with the given amount of money,
     *          the given amount of money is withdrawn from this
     *          bank account, and deposited to the given
     *          destination account.
     */
    public void transferTo
```

```
      (long amount, BankAccount destination)
{
  if ( (amount > 0)
    && (destination != null)
    && (! this.isBlocked())
    && (this.getBalance() >= getCreditLimit() + amount)
    && (destination.getBalance() <=
            getBalanceLimit() - amount) )
  {
    this.withdraw(amount);
    destination.deposit(amount);
  }
}

/**
 * Set the balance of this bank account to the given balance.
 *
 * @param   balance
 *          The new balance for this bank account.
 * @post    If the given balance is not below the credit
 *          limit and not above the balance limit, the new
 *          balance of this bank account is equal to the
 *          given balance.
 */
private void setBalance(long balance) {
  if ( (balance >= getCreditLimit())
    && (balance <= getBalanceLimit()) )
    this.balance = balance;
}

/**
 * Variable registering the balance of this bank account.
 */
private long balance = 0;

/**
 * Return the credit limit that applies to all bank accounts.
 *   The credit limit expresses the lowest possible value for
 *   the balance of a bank account.
 */
@Basic
public static long getCreditLimit() {
  return creditLimit;
}

/**
 * Set the credit limit that applies to all bank accounts to
 * the given credit limit.
 *
 * @param   creditLimit
 *          The new credit limit for all bank accounts.
 * @post    If the given credit limit is not above the credit
 *          limit that currently applies to all bank
 *          accounts, the new credit limit that applies to
 *          all bank accounts is equal to the given credit
 *          limit.
 */
public static void setCreditLimit(long creditLimit) {
```

```
    if (creditLimit < getCreditLimit())
      BankAccount.creditLimit = creditLimit;
  }

  /**
   * Variable registering the credit limit that applies to all
   * bank accounts.
   */
  private static long creditLimit = 0;

  /**
   * Return the balance limit that applies to all bank
   * accounts.
   *   The balance limit expresses the highest possible value
   *   for the balance of a bank account.
   */
  @Basic @Immutable
  public static long getBalanceLimit() {
    return balanceLimit;
  }

  /**
   * Variable registering the balance limit that applies to all
   * bank accounts.
   */
  private static final long balanceLimit = Long.MAX_VALUE;

  /**
   * Check whether this bank account is blocked.
   *   Some methods have no effect when invoked against blocked
   *   accounts.
   */
  @Basic
  public boolean isBlocked() {
    return this.isBlocked;
  }

  /**
   * Set the blocked state of this bank account according to
   * the given flag.
   *
   * @param   flag
   *          The new blocked state for this bank account.
   * @post    The new blocked state of this bank account is
   *          equal to the given flag.
   */
  public void setBlocked(boolean flag) {
    this.isBlocked = flag;
  }

  /**
   * Block this bank account.
   *
   * @effect  The blocked state of this bank account is set to
   *          true.
   */
```

```
  public void block() {
    setBlocked(true);
  }

  /**
   * Unblock this bank account.
   *
   * @effect  The blocked state of this bank account is set to
   *          false.
   */
  public void unblock() {
    setBlocked(false);
  }

  /**
   * Variable registering the blocked state of this bank
   * account.
   */
  private boolean isBlocked = false;

}
```

Example 6: Full definition of the class of bank accounts.

1.3.1 Instance methods

In section 1.1.2.3, we have explained that the definition of an instance method involves an implicit argument. Each time we invoke an instance method, its implicit argument is bound to the prime object against which the method is invoked. Instance methods also involve a list of formal arguments. The signature of an instance method introduces names for each of its formal arguments. These names serve to access formal arguments in the body of their method. In Java, we can access the implicit argument in the body of an instance method under the name **this**. We can think of **this** as a predefined formal argument that references the prime object against which the method is invoked. We must further think of **this** as an argument that is qualified **final** in its declaration. Java thus guarantees that **this** references the prime object during the entire execution of an instance method. In Figure 7 on page 80, we illustrated how **this** binds to the prime object against which we invoke an instance method.

Getters and setters

Because the predefined variable **this** references the prime object against which clients invoke an instance method, we can use it to access instance variables of that object. On page 68, we explained the general notation to select instance variables ascribed to objects. It consists of an object expression followed by the name of the instance variable to be selected. The expression **this**`.instanceVariableName` thus

selects the named instance variable ascribed to the prime object of an instance method. In Example 6 above, the bodies of the instance methods for getting and setting properties ascribed to bank accounts illustrate such selections.

As a first example, consider the definition of the instance method `getBalance` in the class of bank accounts. In the body of that method, we select the instance variable `balance` of the prime object **`this`**. The expression **`this`**`.balance` is called a *qualified selection* of the instance variable `balance`. In the expression we explicitly state from which object – the prime object in this case – we want to select an instance variable. We can select instance variables of the prime object in a more implicit way. Whenever possible, Java interprets *unqualified selections* of an instance variable as a selection of that instance variable from the prime object. An unqualified selection of an instance variable just consists of the name of that variable. The statement **`return`** `balance;` would thus be an alternative for the statement **`return this`**`.balance;` in the body of the instance method `getBalance`.

The getters `getNumber` and `isBlocked`, and the setters `setBalance` and `setBlocked` access instance variables of the prime object in a similar way. In the body of the method `setBalance`, we must access the instance variable `balance` in a qualified way. Indeed, the formal argument for the method is also called `balance`, and all non-qualified occurrences of `balance` in the body of that method refer to that argument. This is due to the static nesting of blocks, which we discuss in more detail in the next paragraph. Anyhow, because the formal argument of a setter for a property often has the same name as the instance variable storing the current value of that property, we prefer to select instance variables consistently in a qualified way in the body of setters.

Coding Advice 10: Prefer qualified selections of instance variables of objects over unqualified selections.

Java is a statically scoped language with block structure. A *block* is a sequence of statements and declarations enclosed in braces (`{…}`). Method bodies and class bodies are thus blocks. Blocks can be statically nested. As an example, the body of the instance method `setBalance` is statically nested in the body of the class of bank accounts. The code snippet below shows a more schematic example. The *scope* of a local variable is restricted to the block in which it is declared. The Java Virtual Machine can thus release memory allocated to a local variable as soon as the flow of control leaves the block in

which that variable is declared. In the code snippet below, the scope of the local variable `innerVariable` is thus restricted to the nested block in which it is declared. The Java compiler rejects all attempts to access that local variable in the enclosing block. In the same way, the body of a method delimits the scope of its formal arguments. As an example, the body of the instance method `setBalance` delimits the scope of its formal argument `balance`.

A nested block has access to a local variable declared in an enclosing block, unless that nested block in turn declares a local variable with the same name. In the code snippet below, the nested block thus has full access to the variable `outerVariable` declared in the enclosing block. On the other hand, the nested block has no access to the variable `otherVariable` declared in the enclosing block. Indeed, the nested block also declares a variable named `otherVariable`. All occurrences of `otherVariable` in the nested block refer to that variable. For the same reason, the instance variable `balance` is not accessible in an unqualified way in the body of the instance method `setBalance`. All occurrences of `balance` in the body of that method refer to its formal argument.

```
{ // Enclosing block
  int outerVariable;
  float otherVariable;
  …
  { // Nested block
    int innerVariable;
    float otherVariable;
    …
    // innerVariable and otherVariable accessible here!
    innerVariable = 5;
    outerVariable = 8;
    // otherVariable denotes variable declared in nested block!
    otherVariable = 11.0;
    …
  }
  …
  // innerVariable not accessible here!
  innerVariable = 3;
  // otherVariable denotes variable declared in encl. block!
  otherVariable = 7.0;
  …
}
```

In Java, methods use *return statements* to return a result to their caller. For a non-void method, each return statement must involve an expression whose static type is assignable to the result type of its method. Recall that Java uses "return by value" in returning a result to the caller of the method, meaning that the semantics of assignment apply to it. The method `getBalance` thus returns a copy of the contents of the instance

variable `balance` to the caller. For void methods, return statements do not involve an expression. Java implicitly adds a return statement at the end of the body of each void method. This is not the case for non-void methods. All possible paths leading through the body of such a method must end with an explicit return statement.

The execution of a return statement immediately terminates the execution of the method in which it occurs. Return statements may therefore occur at all points where statements are expected in the body of a method. The Java compiler checks whether all statements in the body of a method are in reach for execution. As an example, the Java compiler would reject a statement immediately following the return statement in the body of the method `getBalance`.

Complex methods

We only access instance variables for storing properties ascribed to individual objects in getters and setters associated with those properties. In the implementation of all other methods, we directly or indirectly invoke getters and setters to manipulate properties ascribed to objects and classes. In this way, we delimit aspects concerning the representation of objects and classes to a few methods of their class. Coding Rule 20 on page 67 imposes to restrict access to the instance variables to the body of their class. Because of this rule, changes in the representation of objects only affect the definition of their class. If we restrict access to instance variables to the bodies of getters and setters, changes in the representation of objects are even more simple to work out. The only exception to this rule are initializations of final instance variables, as we explain in section 1.3.3.

A side-effect of this rule is that we never directly access instance variables of other objects than the prime object of instance methods. Java itself does not prohibit us to access instance variables of other objects, even if we have qualified them **`private`**. As an example, we could use the expression `(other.balance)` in the body of the method `hasHigherBalanceThan` instead of `(other.getBalance())`. Because that body is in the scope of the class of bank accounts, it has access to the private instance variable `balance` of all bank accounts that are in its reach.

Coding Rule 25: Access non-final instance variables and class variables only in the body of getters and setters. Initialize final instance variables in the body of constructors.

As a first example, consider the implementation of the inspector `hasHigherBalanceThan(long)`. In the body of that method as worked out in Example 6 on page 98, we use the basic inspector `getBalance` to query the balance of the prime bank account. Instead, we could have chosen to access the instance variable `balance` directly. The return statement would then become `return this.balance > amount;`. That implementation is even slightly more efficient. It avoids the potential overhead caused by the invocation of the inspector `getBalance`. However, some day we may want to change the way we store information concerning bank accounts. We may for instance decide to store the state of all bank accounts in a relational database. Such changes do not influence the implementation of the inspector `hasHigherBalanceThan(long)` as worked out in Example 6. They only have an impact on the implementation of getters and setters. If, on the other hand, we directly access instance variables in the body of other methods, we must also change the implementation of these methods each time we change something to the representation of bank accounts.

As for accessing instance variables, we can invoke instance methods against the prime object in two different ways. In a qualified invocation, we invoke the method explicitly against the predefined variable `this`. In a non-qualified invocation, we invoke the method without explicitly stating its target object. In the body of the method `hasHigherBalanceThan(long)`, we invoke the inspector `getBalance` in a non-qualified way. Unqualified invocations of instance methods against the prime object are common practice in Java. We only use qualified invocations, if we want to emphasize that we invoke an instance method against the prime object. In the body of the method `transferTo` from Example 6, we use such a qualified invocation to emphasize that the money is withdrawn from `this`, and subsequently deposited to `destination`.

Coding Advice 11: Invoke instance methods against the prime object in an unqualified way, unless you want to emphasize that they are invoked against `this`.

As another example, consider the implementation of the mutator `withdraw`. In the body of that method, we use the methods `getBalance` and `setBalance` to query, respectively to change the balance of the prime account. That method uses a *conditional statement* to distinguish the normal case from the exceptional case. In its simplest form, a conditional statement in Java has the form `if (expression) statement`. Controlling expressions of conditional statements must be of type `boolean`. We must enclose them in parentheses. If the controlling expression of a conditional statement evaluates to `true`,

the statement following the controlling expression is executed. Otherwise, execution proceeds with the statement following the simple conditional statement. If we have more than one statement to execute in case the controlling expression evaluates to **true**, we must enclose all of them in a block. In a more expanded form, a conditional statement in Java has the form **if** `(expression) statement` **else** `statement`. If the controlling boolean expression of such a conditional statement evaluates to **true**, the statement following the controlling expression is executed. Otherwise, the statement following the keyword **else** is executed.

In depositing money to an account – and also in withdrawing money – we must see to it that the balance of that account stays in the range delimited by the credit limit at the low side and by the balance limit at the high side. A rather naïve way to formulate a test for the method `deposit` would be `(getBalance() + amount <= getBalanceLimit())`. However, the semantics of computations with integer numbers in Java tells us that an *overflow* may occur in computing the new balance as the sum of the old balance and the amount to be deposited. Indeed, assume both the balance limit and the current balance of the account are close to `Long.MAX_VALUE`. Adding a sufficiently high amount to the current balance then yields a value that exceeds the largest possible value in the type **long**. The semantics of integer additions in Java then states that the resulting value is negative. The naïve test would then conclude that the amount is acceptable for deposit.

In the bodies of the methods `deposit` and `withdraw`, we implement the tests in such a way that no overflows can occur in integer computations they use. For the method `deposit`, we formulate the test as `(getBalance() <= getBalanceLimit()-amount)`. Instead of adding the given amount to the current balance, we now subtract it from the balance limit. We can easily prove that this subtraction cannot result in an overflow. Indeed, the policy of our bank states that the balance limit is a strict positive value. Prior to the test, we have already verified that the deposit amount is also positive. By definition, the subtraction of a positive amount from a positive amount can never result in an overflow. In other words, this implementation of the test always yields the proper result. Notice that we also reformulate the postcondition in a similar way. Indeed, if we use operators such as addition and multiplication in specifications, we assume they have the semantics of the Java operators.

Besides getters and setters, complex methods may also use other methods in their implementation. As a simple example, consider the implementation of the inspector

`hasHigherBalanceThan(BankAccount)` in the definition of the class of bank accounts on page 98. That method uses the inspector `hasHigherBalanceThan(long)` to determine its result. In the implementation of the latter inspector, we use the getter associated with the balance of a bank account. Similar remarks apply to the implementation of the mutators `block` and `unblock` in the class of bank accounts.

The implementation of the method `transferTo` is a bit more complex. According to the specification of the method, the method must either realize a complete transaction or nothing at all. In particular, if the method successfully withdraws the given amount of money from the prime account, it must also successfully deposit that amount to the destination account, and vice versa. In the implementation as worked out in Example 6, we choose to check all conditions imposed on both transactions, before we invoke the methods `withdraw` and `deposit` against the proper account. The disadvantage of this implementation is that we now check all conditions twice: once in the body of the method `transferTo` itself, and once in the bodies of the methods `deposit` and `withdraw`. Moreover, if the conditions for withdrawing or depositing change, we must change the body of the method `transferTo` as well.

The code snippet below shows an alternative implementation. In this version, we invoke the methods `withdraw` and `deposit` unconditionally. Upon return, we check whether the withdrawal, respectively the deposit was effective by comparing the balance upon entry with the balance upon return. The disadvantage of this implementation is that we must re-deposit the money to the prime bank account, in case the withdrawal is successful and the deposit to the destination account fails. In a single-threaded environment such a restore causes no problems. In a multi-threaded environment – an environment in which several threads operate on the same set of bank accounts – the re-deposit may fail. Other threads may then change the balance of the prime bank account in between the withdraw and the re-deposit.

```
public void transferTo(long amount, BankAccount destination)
{
  if (destination != null) {
    long oldBalance = this.getBalance();
    this.withdraw(amount);
    if (this.getBalance() != oldBalance) {
      // Withdrawal was successful!
      oldBalance = destination.getBalance();
      destination.deposit(amount);
      if (destination.getBalance() == oldBalance) {
        // Deposit was not successful! Re-deposit!
        this.deposit(amount);
```

```
        }
      }
    }
}
```

In the broad area of designing algorithms, the technique of implementing more complex methods in terms of less complex ones is known as "*divide and conquer*". This principle states that we must solve a problem either directly because solving it is easy, or by *dividing* it into two or more smaller problems if the problem is too complex to be solved at once. In the latter case, we say that we *conquer* the problem by dividing it into several smaller problems. We solve the smaller problems in turn by recursively applying "divide and conquer". The implementation of the method `transferTo` illustrates this strategy. The problem of transferring money from one account to another is divided into the problem of withdrawing that amount from one account and of depositing it to another account. We solve the problems of withdrawing and depositing amounts of money, in turn, using methods for querying and setting the balance of an account. The latter problems are simple enough to be solved directly.

1.3.2 Class methods

The implementation of class methods is very similar to the implementation of instance methods. In section 1.1.2.3 we already explained that class methods do not have an implicit argument. In other words, class methods do not have a predefined formal argument **`this`** referencing a prime object. All the methods of a class – both instance methods and class methods – have access to the static variables defined in the body of their class. Outside the body of their class, we must access static variables in a qualified way. In the body of their class, we can select class variables also in a non-qualified way. Consider the implementation of the static method `setCreditLimit` in Example 6 on page 98. In the body of that method, we use the class variable `creditLimit` in a qualified way. If we would change the name of the formal argument to `limit`, it would no longer conflict with the name of the class variable. Then, we could write the assignment in an unqualified way as in `creditLimit = limit;`. It is common practice not to access static variables in a qualified way in the body of their class, unless their name conflicts with names of formal arguments or local variables declared in a nested scope.

Coding Advice 12: Do not access static variables or invoke static methods in a qualified way in the body of their class, unless their names conflict with names of other entities in nested scopes.

We have access to static variables and static methods in the entire body of their class. This means that we can also invoke static methods in the body of instance methods of their class. In the definition of the class of bank accounts from Example 6, we use this facility several times. As an example, in the body of the instance method `deposit` we invoke the static method `getBalanceLimit`. We use that method there to check whether the resulting balance does not exceed the largest possible value imposed on the balance of bank accounts. At this point, Java allows us to access the static variable `balanceLimit` directly. However, for reasons of adaptability, we choose not to access static variables outside the bodies of getters and setters as stated in Coding Rule 25.

1.3.3 Constructors

Constructors serve to initialize newly created objects of their class. Upon entry to a constructor, all the non-final instance variables of the newly created object are initialized to the default value of their type. Final instance variables, on the other hand, have no initial value upon entry to a constructor. The Java compiler inserts explicit initializations of instance variables in their declaration as well as initializations in instance initialization blocks at the start of the body of constructors. The *expanded body* of a constructor is its actual body extended with explicit initializations in instance variable declarations and with the body of instance initialization blocks. The actual *body of a constructor* is very similar to the body of an instance method. In particular, constructors also have an implicit argument **`this`** that refers to the newly created object that must be initialized.

As a first example, consider the implementation of the most extended constructor in the class of bank accounts on page 98. The body of the constructor starts with the initialization of the number of the newly created bank account. In the class of bank accounts, we have qualified the instance variable `number` **`final`**. For such variables, Java always demands exactly one explicit initialization either in their declaration, in an instance initialization block or in the body of the constructor. In the example, we do not work out an explicit initialization in the declaration, and we do not use instance initialization blocks. We must therefore initialize the instance variable `number` in the body of the constructor. Because final instance variables are not initialized to the default value of their type, Java demands an explicit initialization along each possible path through the body of the constructor. The conditional statement **`if`** `(number > 0)` **`this`**`.number = number;` is thus not a proper alternative for the statements worked out in the body of the

constructor. Indeed, that alternative has no explicit initialization for the instance variable `number` in case the given number is not positive.

The body of the constructor proceeds with the initialization of the balance of the newly created bank account. We have explicitly initialized the instance variable `balance` in its declaration to `0`. The Java compiler thus inserts that initialization in front of the actual body of the constructor. In the actual body, we change the contents of the instance variable to the given initial balance, if the latter is in the range imposed on the balance of a bank account. Otherwise, we leave the initial value `0` for the balance untouched. Without the explicit initialization to `0`, the implementation of the constructor would have exactly the same effect. Indeed, upon allocating memory for the new bank account, the Java Virtual Machine initializes the balance to the default value `0` of the primitive type **`int`**. However, in Coding Advice 9 we recommend to initialize each non-final instance variable explicitly in their declaration, whenever a proper initial value is available.

In discussing the implementation of instance methods, we have already emphasized the importance of implementing methods in terms of other methods. Obviously, we also want to invoke constructors in the body of other constructors. In Java, a constructor can *invoke another constructor* of its class using the construct **`this`**`(`a_1`, `a_2`, …, `a_n`)`. In this construct, the keyword **`this`** is used as the name of a constructor of its class, and not as a variable referencing the prime object of the constructor. As for all method invocations, the construct involves a list of actual arguments. In Example 6 on page 98, we implement the other constructors of the class of bank accounts in this way. The constructor involving only a number and an initial balance invokes the most extended constructor. The constructor involving only a number invokes the constructor involving a number and an initial balance. If a constructor invokes another constructor of its class, the Java compiler does not extend its body with initializations of instance variables originating from their declaration or from instance initialization blocks. Those initializations are then executed as part of the body of the invoked constructor.

Java imposes rather severe restrictions on the invocation of constructors in the body of other constructors. First of all, we cannot invoke several constructors in the body of another constructor. Moreover, an invocation of a constructor in the body of another constructor must be the first statement in the body of the latter constructor. Finally, a constructor cannot invoke itself because that would lead to a non-terminating initialization process. Because of all these restrictions, we have no other option than to invoke more

extended constructors in the body of less extended constructors. Indeed, we only want to work out the further initialization of properties ascribed to new objects in the body of one and only one constructor. In our strategy, we work out all initializations in the most extended constructor. The most extended constructor is a constructor that uses a formal argument to initialize each property of the newly created object, with exception of properties that no constructor initializes by means of user-supplied values. By definition, only one such constructor exists in a class. Less extended constructors always invoke that most extended constructor.

If we would work out initializations in less extended constructors, we may have to work out the initialization of some properties in the body of several constructors. In the class of bank accounts, the least extended constructor would then initialize the number of the new bank account. The middle constructor would invoke the least extended constructor, and would work out the initialization of the balance. The most extended constructor would invoke the middle constructor, and would work out the initialization of the blocked state. Assume the class of bank accounts would introduce yet another constructor with signature `public BankAccount(int number, boolean isBlocked)`. That constructor would also invoke the least extended constructor to initialize the number of the new bank account. However, it would have to work out the initialization of the blocked state. That means that we would work out the initialization of the blocked state in this constructor and in the most extended constructor.

Having the same code at different points in a software system is one of the most awful things we can do in working out a software system. Coding Rule 26 below therefore recommends to invoke more extended constructors in the body of less extended constructors. In Chapter 3 on page 244, we discuss that checked exceptions may prohibit such invocations. We therefore suggest that constructors best use unchecked exceptions to signal exceptional cases. The compiler will then not check whether all exceptions that can be signaled by an invoked constructor are handled properly by the invoking constructor. In fact, we will suggest not to use any checked exceptions in Java programs.

Coding Rule 26: Invoke a more extended constructor of the class in the body of a less extended constructor.

1.4 Class Verification

As soon as we have completed the definition of a class, our job is not done yet. We must verify whether the definition of our class is correct. In the scope of object oriented programming, verifying the correctness of a class means checking whether the implementation of that class is consistent with its specification. In general, we can only verify the correct behavior of components, if we have a document describing how that component must behave. Therefore, we cannot verify the correctness of the specification of a class, because we have nothing to verify that specification against.

Different strategies exist to verify the correctness of a piece of software. First of all, we distinguish between static verification strategies and dynamic verification strategies. Static verification strategies only use the definition of a class. They do not carry out any experiments with the executable code. *Code inspection* is a simple example of a static verification strategy. Code inspection is just another name for a critical review of the definition of a class, preferably by a colleague software engineer. Formally *proving the correctness* of a piece of software is a much more complex example of a static verification strategy. That strategy first of all needs formal specifications of classes and methods. We introduce such a formal specification language in Chapter 2. Proving the correctness of a piece of software then becomes similar to proving the correctness of propositions in mathematics.

In dynamic verification strategies, we use concrete experiments to establish the correctness of a piece of software. In such strategies, we use the executable version of the class as it is produced by a compiler. If we use a dynamic strategy, the verification of a piece of software is often called *testing*. Testing a piece of software can only reveal errors. After having tested a component, we cannot guarantee that the component is free of errors. If we test the correctness of a single class, we talk about *unit testing*. If we are testing whether several classes – units – cooperate well, we talk about *integration testing*. In testing a piece of software, we need a strategy to develop a coherent collection of tests. In this section, we discuss the principles of black box testing to develop such a test suite for a single class. In Chapter 6 we discuss another strategy known as white box testing.

1.4.1 Principles of black box testing

Black box testing is a rather simple strategy to check whether a component behaves in the expected way. In *black box testing*, we treat the component that will be tested as a black box. This means that we derive an adequate set of tests from the functional description of the component that will be tested. Black box testing is therefore also called functional testing. Contrary to *white box testing*, we do not use any knowledge about the internal workings of the component in a black box testing. The strategy can therefore not guarantee that it covers all possible paths through the body of a method. Black box testing is used in several engineering disciplines. As an example, electronic engineers often use black box testing to verify whether electronic chips operate correctly. They produce signals at the input ports of the chip, and test whether the chip produces the expected signals at its output ports.

In the context of object oriented programming, we use black box testing to check whether the non-private methods offered by a class behave as expected. We derive a test suite from the specification – the documentation – of the methods. Obviously, we cannot test private methods of a class. They are not accessible outside the body of their class. We do not use any details concerning the implementation of methods that will be tested. This means that we can set up a test suite as soon as we have completed the class interface. Verification and implementation of a class can thus proceed in parallel. In fact, we don't even need software engineers for sketching a coherent set of tests, if the documentation is written in a natural language.

Boundary testing

In the context of black box testing, different strategies exist to set up a test suite. *Boundary value testing* is a simple strategy for setting up a series of tests for software components. The strategy starts from the observation that lots of errors tend to occur near extreme values for input variables. A study ordered by the U.S. Army reveals that a large portion of errors in software systems are boundary value faults. A typical example are loops iterating through a sequence of elements. Often such loops fail to handle the last element in the sequence, or attempt to iterate one more element beyond the last element in the sequence.

Figure 9 illustrates the basic principles underlying boundary value testing. For each input variable, we set up tests involving (1) the minimum value of that input variable, (2) a value just above that minimum value, (3) a medium value, (4) a value just below the maximum value, and (5) the maximum value for that input variable. As an example, consider the instance method for withdrawing some amount of money from a bank account. For this method, the balance of the bank account and the amount to be withdrawn act as the variables x_1 and x_2 in Figure 9. The test suite thus includes five tests against a bank account with random medium value for its balance. The amounts to be withdrawn from that account range between the largest and the smallest possible amount that can be withdrawn. Other tests run against bank accounts with a balance equal to or close to the credit limit for bank accounts, respectively close to the balance limit for bank accounts.

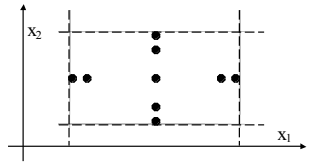


Figure 9: Boundary value testing.

Several variants on the basic strategy for boundary value testing exist. *Robustness testing* extends the tests generated by a basic boundary value testing strategy with values outside the regular domain of input values. In the example of the instance method `withdraw`, this would lead to extra tests for which the method has announced not to change the balance of the account against which it is invoked. Another extension to boundary value testing is *worst-case testing*. In this strategy, all the selected values in the different ranges for input arguments are combined one by one. *Special value testing* is yet another variation on the general theme of boundary value testing. This strategy adds tests covering special values for input variables to the test suite.

Equivalence testing

Equivalence testing is a more refined strategy for building black box tests. This strategy partitions the domain of possible input values into disjoint subsets. The basic criterion to partition the input domain is that the software component that will be tested behaves the same for each set of values in a particular subset. The test suite then includes an actual test for one representative value from each subset. For some methods, it is more appropriate to partition the domain of possible output values instead of the domain of input

values. In that case, the individual tests in the test suite are such that they each generate an output in a different subset of the partition.

Figure 10 illustrates the basic principles underlying equivalence class testing. In the example, the range of input values for the first argument is partitioned into three subsets. The range of input values for the second argument is partitioned into two subsets. In total, this results in a partitioning of the input domain into 6 subsets. The generated test suite uses a representative value from each of these six subsets. For the method to withdraw an amount of money from a bank account, the domain of withdrawal amounts is partitioned into three ranges: (1) negative amount of money to be withdrawn, (2) an amount of money that keeps the balance above the credit limit, and (3) an amount of money that would result in a balance below the credit limit. The domain of values for the balance of an account is not partitioned. As a result, equivalence testing for this method yields three different tests.

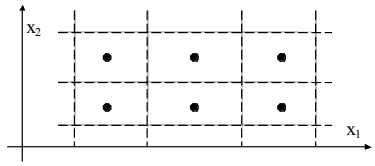


Figure 10: Equivalence testing.

It must be obvious that equivalence class testing has the potential to result in test suites of superior quality. First of all, there are good reasons to believe that the test suite is more complete. By definition, different tests cover different cases that we can distinguish in the specification of a method. This is not at all guaranteed by boundary value testing. On top of more complete test suites, strategies for equivalence class testing also avoid redundant tests. Boundary value testing strategies typically produce lots of tests that cover more or less the same case in the specification of the software component under test. In this book, we prefer to use equivalence testing in building test suites.

Testing Advice 1: Work out a separate test for each case you can distinguish in the specification of a method, if you opt for black box testing.

1.4.2 Black box testing with JUnit

In testing a piece of software, we must be aware that software changes over time. We may have to add new methods to the definition of a class or to remove existing

methods from it. We may have to change the specification of a method, or we may have to change its implementation. Some of these changes imply changes to the test suite as well. Regardless of what we actually change, we always execute all the tests in the test suite to see whether the class still behaves as expected. In developing a test suite, we must thus be aware that it will run several times against its class. A proper test suite must therefore not spit out lines and lines of test results, which software engineers must verify manually. A proper test suite checks itself the results obtained from the individual tests, and only reports on tests that fail.

JUnit is a framework for developing unit tests. The framework supports the development of test suites and test cases. A *test suite* in JUnit is a collection of tests. We discuss test suites in Chapter 6. A *test case* in JUnit is a collection of tests using the same set up. We do not intend to discuss all the facilities offered by JUnit in full detail in this book. Moreover, JUnit uses Java constructs such as packages, inheritance and reflection, which we only discuss in later chapters of this book.

We suggest to develop a separate test case for each class in a software system. A test suite then collects all the test cases for all the classes involved in the software system under test. Example 7 below lists part of a JUnit test case for the class of bank accounts. The example defines a class named `BankAccountTest` that collects all the tests for all the methods offered by the class of bank accounts. The listed fragment only shows the tests for the mutator `withdraw` and for the inspector `hasHigherBalanceThan`. Notice that the name for a test case for a class consists of the name of that class followed by the noun `Test`.

Testing Advice 2: Develop a JUnit test case for each class in a software system that will be tested.

JUnit test cases use ingredients from the JUnit framework. That framework is organized in a series of packages. In Chapter 5 we explain that packages lead to *fully qualified names* for classes and packages. All we need to know right now is that the fully qualified name of a class consists of its own name preceded by the fully qualified name of its package. If we use classes under their fully qualified names, we have a lot of typing to do. Java therefore offers facilities to import elements of packages. If we import an element of a package, we can use it from that point on under its own name. In Example 7 below, we import among others the features `Before` and `Test` from the JUnit

framework. These features are all part of the package `org.junit`. The fully qualified name of those features is thus `org.junit.Before`, respectively `org.junit.Test`. Because we import these features in our test case, we must not use their fully qualified name each time we need them.

Test fixture

JUnit offers facilities to set up and to tear down test configurations. The framework refers to such configurations as *test fixtures*. A test case may define both instance methods and class methods to set up such fixtures. Starting from version 4, JUnit uses annotations to identify features of test cases and test suites. JUnit imposes the annotation `@Before` on instance methods that set up test fixtures. The annotation `@BeforeClass` is imposed on class methods that set up test fixtures. A test case may introduce several instance methods and several class methods to set up text fixtures. However, we introduce at most one such instance method and one such class method.

Class methods annotated `@BeforeClass` are executed at the start of the test case. JUnit guarantees that they are all executed before the first actual test. We use class methods annotated `@BeforeClass` to set up *immutable test fixtures*. Such a fixture consists of a series of objects whose state does not change during the entire test case. Class variables reference objects part of an immutable test fixture. Class methods annotated `@BeforeClass` create the objects and initialize the class variables. In Example 7 below, the immutable test fixture involves two bank accounts with balances `300` and `500` respectively and a blocked account. The class variables `accountWithBalance300`, `accountWithBalance500` and `blockedAccount` reference these bank accounts. The class method `setUpImmutableFixture` – annotated `@BeforeClass` – creates these bank accounts and initializes the class variables. We use the bank accounts with balances `300` and `500` in the tests for the inspector `hasHigherBalanceThan`. By definition that inspector does not change the state of the objects involved in it. The blocked account is used in a test for the method `withdraw`. That test neither changes the state of that bank account.

Testing Advice 3: Complement a test case with an immutable test fixture to collect objects whose state does not change during the entire test case.

Instance methods annotated `@Before` are executed just before each individual test. They define a *mutable test fixture*. Such a fixture consists of a series of objects whose

state may change during individual tests. Instance methods annotated `@Before` typically create objects and assign references to them to instance variables. Because these methods are executed before each test, mutable test fixtures are re-created before each individual test. In Example 7 below, the mutable test fixture only involves a bank account with balance `1000`. The instance variable `accountWithBalance1000` references this bank account. The instance method `setUpMutableFixture` – annotated `@Before` – creates that bank account and registers a reference to it in the instance variable `accountWitBalance1000`. We use that bank account in several tests for the mutator `withdraw`. Some of these tests change the state of that bank account. Example 7 only shows part of the test fixtures for the class of bank accounts. We obviously need more bank accounts to test more complex methods. As an example, we need at least two bank accounts to test the method `transferTo`.

Testing Advice 4: Complement a test case with a mutable test fixture to collect objects whose state changes during at least one individual test.

The JUnit framework offers complementary methods to tear down test fixtures. Instance methods to tear down mutable test fixtures must be annotated `@After`. JUnit guarantees that these methods are exe cuted right after each individual test. Class methods to tear down immutable test fixtures must be annotated `@AfterClass`. They are executed after the last test in the test case. Methods to tear down test fixtures serve to perform some final actions, before their objects disappear from the scene. Typical examples of such final actions are releasing resources allocated to these objects, such as files, network connections and database connections. Other examples are test fixtures that change data stored in persistent storage structures such as databases and files. Test fixtures in this book do not involve such objects. We therefore never have a need to tear down test fixtures.

Individual tests

Once we have set up proper test fixtures, we can start working out individual tests. For each case that we distinguish in the specification of the method under test, we work out a test method in the test case. In JUnit 4, *test methods* must be annotated `@Test`. Test methods must be public instance methods, they may not have any formal arguments, and must not return a result. In Example 7 below, we define four methods to test the correctness of the method `withdraw`, and two test methods to test the

correctness of the method `hasHigherBalanceThan(BankAccount)`. We suggest to start the name of a test method with the name of the method that will be tested. We complement the name of the test method with a brief description of the particular case the test method aims at. As an example, the test method `withdraw_BalanceOverflow` tests the correctness of the method `withdraw` in case the balance is overdrawn. For overloaded methods, we extend the name of the method with information concerning the types of the arguments. As an example, the test method `hasHigherBalanceThanAccount_EffectiveCase` tests the correctness of the method `hasHigherBalanceThan` expecting a bank account as its formal argument.

Testing Advice 5: Name test methods after the method that will be tested complemented with a brief description of the actual case they aim at.

As mentioned before, the test case for the class of bank accounts includes four test methods for the instance method `withdraw`. The specification of the method `withdraw` clearly distinguishes these four cases. In addition to the regular case in which the amount is actually withdrawn from the prime bank account, the specification recognizes three exceptional cases in which the state of the prime bank account is left untouched. In addition to a test method for regular withdrawals, the test case therefore includes test methods to test the correctness of withdrawals in case the withdrawal amount is negative, in case the withdrawal amount would cause a balance overflow, and in case the prime bank account is blocked.

Each of the test methods in Example 7 below, reflects the typical structure for a test. First, the method under test is invoked against an object from one of the test fixtures. As an example, the test method `withdraw_LegalCase` invokes the method `withdraw` against the bank account with balance `1000` from the mutable test fixture. We best use a bank account from the mutable test fixture in this case. Otherwise, the state of that bank account would be different from that point on in the test case. For test methods that do not change the state of any object involved in it, it does not matter whether we use a bank account from the mutable test fixture or from the immutable test fixture. In that case, we just pick any object with a proper state for the test. As an example, in the test method `withdraw_NegativeAmount`, we could just as well use the bank account with balance `500` from the immutable test fixture.

Once we have invoked the method that under test against a proper object, we must compare the results obtained from that invocation against the expected results as described in the specification of that method. In testing the legal case for the method `withdraw`, a withdrawal of `200` is invoked against a bank account with balance `1000`. Upon return from the method, the balance of that account must thus be `800`. In all the tests worked out in Example 7 below, we calculate ourselves the result the method must return. We then compare those manually computed results against the results obtained from invoking the method under test. An alternative strategy would be to confront results obtained from method invocations with results obtained by evaluating ingredients from their specification. For testing the legal case of the instance method `withdraw`, we would then check whether the resulting balance is equal to the balance upon entry – stored in some local variable prior to the method invocation – diminished with the withdrawal amount.

JUnit offers a series of methods for evaluating results obtained from testing methods. All these methods start with the noun `assert`. We therefore refer to them as assert methods. They all serve to check whether certain conditions are indeed true during the execution of successive tests. The JUnit framework reports on each assertion failure during the execution of a test suite. One of the assert methods offered by JUnit is the static method `assertEquals`. This method compares whether the first argument – the expected result – is equal to the second argument – the actual result obtained from the test. The JUnit framework defines several overloaded methods of `assertEquals`. The one used in the test methods for withdrawals expects two objects. JUnit also offers overloaded methods `assertEquals` for comparing floating point values in single and in double precision and for comparing arrays of objects. The framework also offers versions for all these methods involving an additional argument of type `String`. We can use that extra argument to give some details concerning the particular assertion. In addition to the methods `assertEquals`, the framework offers other methods for asserting conditions. Examples are the methods `assertFalse` and `assertTrue` to check whether a given boolean expression evaluates to **`false`**, respectively to **`true`**. We use these methods in testing the method comparing the balance of two bank accounts.

Example 7 below defines two test methods to test the correctness of the instance method `hasHigherBalanceThan(BankAccount)`. The first test method uses an effective bank account as the account to compare with. The second test method covers the case of a non-effective bank account to compare with. In testing this method, we

more or less assume that the instance method `hasHigherBalanceThan(long)` is correct. Indeed, the inspector involving a bank account states that comparing the balance of its prime bank account with some other bank account, has the same effect as comparing the balance of its prime bank account with the balance of the other bank account – an integer number – in case that bank account is effective. The result obtained from the inspector involving a bank account is thus specified in terms of the inspector involving an integer number to compare with. For the latter inspector, the test case for the class of bank accounts will include at least two test methods. A first test method covers the case in which balance of the prime bank account is indeed higher than the supplied integer number. The second test method covers the case in which the balance of the prime bank account is not higher than the supplied integer number. We must not distinguish these cases again in working out test methods for the inspector to compare the balance of a bank account with another bank account.

```
import static org.junit.Assert.*;
import org.junit.*;

/**
 * A class collecting tests for the class of bank accounts.
 *
 * @version 2.0
 * @author  Eric Steegmans
 */
public class BankAccountTest {

  /**
   * Instance variable referencing bank accounts that may
   * change during individual tests.
   */
  private BankAccount accountBalance1000;

  /**
   * Class variables referencing bank accounts that do not
   * change during the entire test case.
   */
  private static BankAccount accountBalance300,
      accountBalance500, blockedAccount;

  /**
   * Set up a mutable test fixture.
   *
   * @post   The variable accountBalance1000 references a new
   *         bank account with a balance of 1000.
   */
  @Before
  public void setUpMutableFixture() {
    accountBalance1000 = new BankAccount(1111111, 1000);
  }
```

```
/**
 * Set up an immutable test fixture.
 *
 * @post   The variable accountBalance300 references a new
 *         bank account with a balance of 300.
 * @post   The variable accountBalance500 references a new
 *         bank account with a balance of 500.
 * @post   The variable blockedAccount references a new
 *         blocked bank account.
 */
@BeforeClass
public static void setUpImmutableFixture() {
  accountBalance300 = new BankAccount(1234567, 300);
  accountBalance500 = new BankAccount(7654321, 500);
  blockedAccount = new BankAccount(2121212, 333, true);
}

@Test
public void withdraw_LegalCase() {
  accountBalance1000.withdraw(200);
  assertEquals(800L, accountBalance1000.getBalance());
}

@Test
public void withdraw_NegativeAmount() {
  accountBalance1000.withdraw(-1200);
  assertEquals(1000L, accountBalance1000.getBalance());
}

@Test
public void withdraw_BalanceOverflow() {
  accountBalance1000.withdraw(Long.MAX_VALUE);
  assertEquals(1000L, accountBalance1000.getBalance());
}

@Test
public void withdraw_BlockedAccount() {
  long oldBalance = blockedAccount.getBalance();
  blockedAccount.withdraw(200);
  assertEquals(oldBalance, blockedAccount.getBalance());
}

@Test
public void hasHigherBalanceThanAccount_EffectiveCase() {
  assertTrue(accountBalance500.
      hasHigherBalanceThan(accountBalance300));
}

@Test
public void hasHigherBalanceThanAccount_NonEffectiveCase() {
  assertFalse(accountBalance500.
      hasHigherBalanceThan(null));
}

// Tests for other methods of class BankAccount.
```

```
}
```

Example 7: JUnit test case including tests for the instance methods `withdraw` and `hasHigherBalanceThan`.

Developing test methods for the other methods in the class of bank accounts is similar. In total, we have worked out 31 test methods to check the correctness of all the methods in the class of bank accounts. The complete set is available on the website accompanying this book. Notice that we cannot develop black box tests for basic inspectors. Indeed, a basic inspector does not specify the result it returns. The documentation comment attached to a basic inspector only describes the value it returns. Consider for example the basic inspector `getBalance` in the class of bank accounts. The specification of that method as worked out in Example 6 on page 98 only reveals that it returns the amount of money available on the account against which it is invoked. From that description, we cannot conclude what value the method must return when we invoke it against a bank account. In fact, we use that inspector in tests for other methods to check whether these methods manipulate the balance of a bank account in the appropriate way. As an example, in Example 7 above, we use the inspector `getBalance` to observe how the instance method `withdraw` manipulates the balance of the bank account against which it is invoked.