Chapter 2

Implementing Non-Static Features

2.1 Introduction

2.1.1 What are Non-Utilities?

We learned in the previous chapter how to implement a utility, a special kind of class in which the attributes are associated with the class itself rather than with its instances. In this chapter we turn to the general case in which different instances of the class can have different states. Most classes belong to the *non-utility* category, and because of this, it is common to drop the "non-utility" qualifier and refer to these classes as just *classes*. In this chapter, we focus our attention on those classes all features of which are *not* static. In the next chapter, we will discuss classes with both static and non-static features.

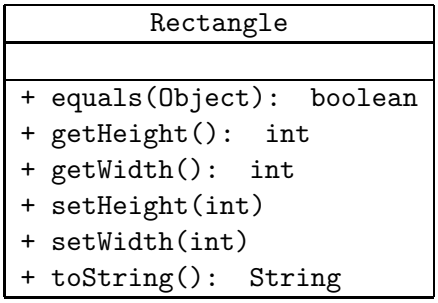
Since a class needs to be instantiated before it can be used, its client must begin by creating an instance of it (i.e. an object). As an example, consider the `Rectangle` class the API of which is shown here. The following fragment shows how a client can use this class. The client starts by creating an empty rectangle using the default constructor and then printing it. It then creates a non-empty rectangle and prints it. Finally, it changes the width of the non-empty rectangle using a mutator and then prints the mutated object.

```
1  Rectangle empty = new Rectangle();
2  output.println(empty);
3
4  int width = 0;
5  int height = 1;
6  Rectangle rectangle = new Rectangle(width, height);
7  output.println(rectangle);
8
9  width = 2;
10 rectangle.setWidth(width);
11 output.println(rectangle);
```

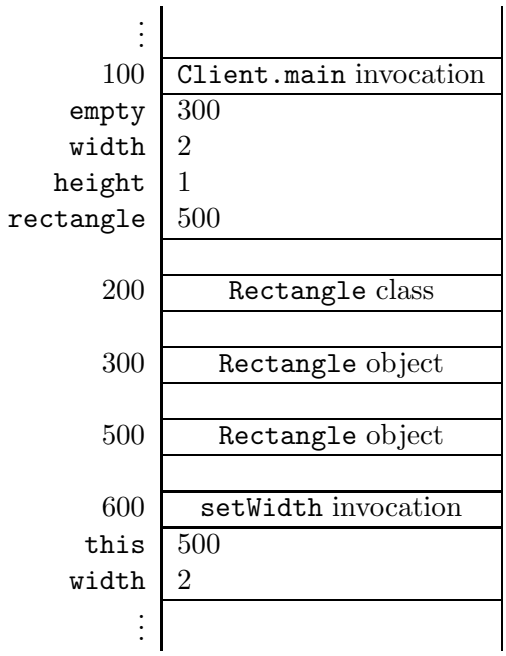
The full example can be found at this link.

2.1.2 UML and Memory Diagrams

We saw in Section 1.1.3 that UML class diagrams can be used to represent utilities. These same diagrams can also be used to represent other classes by simply dropping the `<<utility>>` stereotype. For example, a UML class diagram for the `Rectangle` class of the previous section is shown below. Note that the attribute box is empty and that the return of void methods is left blank.



The memory diagrams introduced in Section 1.1.3 can also be used for other classes with one notable addition: the *implicit parameter* in invocation blocks. As an example, consider the code fragment of the previous section which creates two instances of the `Rectangle` class and mutates one of them. When the invocation of the `setWidth` method on line 10 is executed, memory can be depicted as shown in the following diagram.



The diagram shows an invocation block for the `main` method of the `Client` class, a class block for the `Rectangle` class, two object blocks for the two created `Rectangle` instances, and an invocation block for the `setWidth` method. Recall that, in the client's concern, an invocation block contains one entry for each argument passed to the method. For the implementer, this is true only if the

invoked method is static. For non-static methods, however, the block contains an additional entry for the so-called implicit parameter. This parameter is automatically added by the compiler so even a method such as `getWidth`, which does not have any parameter, will end up having one parameter. The name of the added parameter is `this` and its value is the value of the object reference on which the method is invoked.

With the implicit parameter in the invocation block we can see how a method residing in the class block *knows* on which object it should operate. For example, when the `setWidth` method receives control, it will find 500 as the value of the parameter `this` and 2 as value of the parameter `width`. Hence, all it needs to do is set the `width` of the `Rectangle` object on address 500 to the value 2. We will revisit the implicit parameter at more depth throughout this chapter.

2.1.3 Check your Understanding

Consider the following fragment of client code.

```
1 String s1 = new String("computing");
2 String s2 = new String("^c[a-z]+");
3 boolean ok = s1.matches(s2);
4 output.println(ok);
```

Draw a memory diagram that depicts a snapshot of memory when the statement in line 3 is executing, i.e. just before the `matches` method returns.

2.2 How: The Class Definition

2.2.1 Class Structure

Implementing a class means writing its so-called *class definition*. Its general structure is identical to that of a utility class, namely

```
1  // any needed package statement
2  // any needed import statements
3
4  public class SomeName
5  {
6     // the attribute section
7
8     // the constructor section
9
10    // the method section
11 }
```

The class definition has three sections, one for attributes, one for constructors, and one for methods. These three sections may appear in any order but, as a matter of style, we will adhere to the above order.

If a class implements one or more interface then its header must declare this fact by using the keyword `implements` followed by a comma-separated list of implemented interfaces. For example, if class `C` implements the interfaces `I1` and `I2` then its header becomes

```
1 public class C implements I1, I2
```

This information also appears in the API. A client can deduce from this information that the class `C` contains all the methods specified in the interfaces `I1` and `I2`. As we will see later in this chapter, an implementer has to ensure that the class `C` implements all the methods specified in the interfaces `I1` and `I2`. The latter fact is checked by the compiler.

2.2.2 Declaring and *not* Initializing Attributes

As an implementer of a class, our first task is to determine the attributes that are needed to meet the specification stated in the API of the class. This task is non-trivial because a typical API does not specify its attributes directly. We therefore need to read the API carefully, especially its constructor section, in order to understand what the class encapsulates and then *infer* the needed attributes. It should be noted that the final outcome of this task is not unique: different implementers may come up with a different attributes (in number or in type) and all these implementations are "correct" because their differences are not visible to the client.

Once the attributes are determined, we define them in the attribute section of the class. The definition has the following general form:

access type name`;`

or

access `final` *type name*`;`

Next, we discuss the different elements in some detail.

- *access* (also known as the *access modifier*) can be either `public`, which means this is a *field*, or `private`, which means it is invisible to the client. [1] It is a key software engineering guideline to keep all non-final attributes private. First of all, it forces the client to use a mutator and, hence, prevents the client from assigning wrong values to them, e.g. assigning a negative value to a field intended to hold the age of a person. Secondly, private attributes do not show up in the API and, hence, declaring attributes private (rather than public) simplifies the API. Finally, since private attributes do not show up in the API, their type and name can be changed by the implementer without introducing any changes to the API.

- `final` is used if this attribute is a constant. Note that since these attributes are not static, the word "constant" means it is constant per object, *not* per class; i.e. the value of the attribute cannot be changed, but it does not have to be the same for all instances of the class.

- *type* specifies the type of this attribute. It can be primitive or non-primitive.

[1] The access modifier in Java can also be `protected`, which makes the attribute visible only to subclasses of this class, or can be left blank, which makes the attribute visible only to classes in the same package as this class.

As an example, we consider the `Rectangle` class introduced in the previous section. We see from its API that it encapsulates a rectangle. We also note that the API lists constructors, accessors, and mutators that refer to the width and height of the rectangle using the `int` type. Hence, it is reasonable to capture the state of such a rectangle using a pair of `int` attributes (other possibilities exist and we will examine this further later in this chapter). Based on this, we start the class definition of `Rectangle` as follows:

```
1 public class Rectangle
2 {
3    private int width;
4    private int height;
5    ...
6 }
```

When we define attributes or refer to them, we have to keep the following points in mind:

- We should *not* initialize attributes as we declare them. The initialization of attributes is done on an object-by-object basis by the constructor. Contrast this with the opposite rule that we saw in the previous chapter for static attributes. Spend some time to think through this difference: why are static attributes initialized with their declaration whereas non-static ones are not?

- It is unfortunate that the Java compiler takes a cavalier attitude toward attribute initialization: it does not object if we initialize a non-static attribute as we declare it, and it will assign a default value for each attribute we leave without initialization! This behaviour makes code harder to understand, increases the exposure to assumption risks (see Section 1.2.2), and encourages habbits that may not be supported in other languages. In this book, we will always initialize static attributes as we declare them and leave the initialization of non-static ones to the constructor section.

- The scope of an attribute is the entire class.

- Recall that we refer to static attributes using the `C.a` syntax, where `C` is the name of the class to which the attribute belongs and `a` is the name of the attribute. A similar syntax is used for *non-static* attributes. We use the object reference in place of the class name. We already saw in Section 2.1.2 that the keyword `this` is used to refer to the object on which the method is invoked. Hence, we write `this.a` whenever we want to refer to the attribute `a` of the object on which the method is invoked.

- It is unfortunate that the Java compiler takes a casual approach toward attribute reference: it does not object if we use `this` to refer to `static` attributes (although tools such as Eclipse do provide a warning) or if we refer to attributes by their name without the `this.` prefix (i.e. as if they were local variables). This may lead to confusion, especially when an attribute and a local variable or parameter have the same name, and makes the code harder to understand and maintain. In this book, whenever we refer to an attribute from within its class definition, we will always use the dot operator preceded by the class name for static attributes and the dot operator preceded by `this` for non-static ones.

The key difference between utility and non-utility classes derives from the fact that state (i.e. the values of all the attributes) plays a key role in the functionality and purpose of a non-utility class.

2.2.3 The Constructor Section

The constructor of a class is invoked whenever the class is instantiated (i.e. whenever the `new` operator is used) and its job is to initialize the state (i.e. the attributes) of the created object. In order to provide versatility and convenience to the client of the class, several constructors may appear in the API of the class and their definitions comprise the constructor section of the class definition.

The definition of a constructor starts with a header with the following general form:

access signature

The access modifier can be either `private` or `public`.[2] The signature consists of the name of the constructor, which has to be the same as the name of the class, followed by the parameter list. If the constructor is public, we can copy and paste the header as-is from the API of the class.

As an example, the default constructor of our `Rectangle` class is implemented as follows.

```
1 public Rectangle()
2 {
3    this.width = 0;
4    this.height = 0;
5 }
```

Note how `this` is used to refer to the object the attributes of which are being initialized.

Similarly, we implement the two-parameter constructor of the `Rectangle` class as follows.

```
1 public Rectangle(int width, int height)
2 {
3    this.width = width;
4    this.height = height;
5 }
```

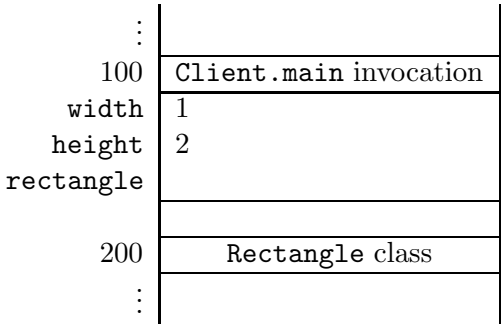
Note that within the body of the constructor, `width` refers to the parameter of the constructor, whereas `this.width` refers to the attribute of the object under construction.

Consider the following fragment of client code.

```
1 int width = 1;
2 int height = 2;
3 Rectangle rectangle;
4 rectangle = new Rectangle(width, height);
```

After the execution has reached the end of line 3, memory can be depicted as follows.

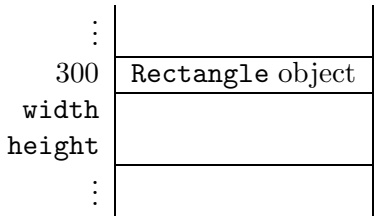
[2] The access modifier in Java can also be `protected`, which makes the constructor visible only to subclasses of this class, or can be left blank, which makes the constructor visible only to classes in the same package as this class.



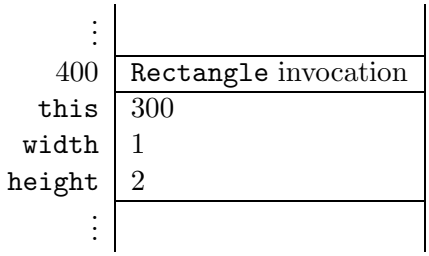
The execution of line 4 can be split into three parts:

- the allocation of a block of memory for a `Rectangle` object,
- the execution of the two-parameter constructor of the `Rectangle` class to initialize the `Rectangle` object, and
- the assignment of the value of the created `Rectangle` object, i.e. its memory location, to the variable `rectangle`.

In the first part, a block of memory is allocated for the new `Rectangle` object. This can be reflected in our memory diagram by adding the following.



In the second part, the two-parameter constructor is invoked. This invocation gives rise to an invocation block. This invocation block not only contains the parameters `width` and `height` and their values, but also the implicit parameter `this` and its value. Recall that the value of `this` is the reference to the object on which the constructor is invoked. In this case, it is 300.



As a result of the execution of the constructor, the state of the `Rectangle` object is initialized. For example, execution of the assignment

```
1 this.width = width;
```

amounts to assigning the value of the parameter `width`, which is 1 according to the invocation block, to the attribute `width` of the object referred to by `this`, which is the object at address 300 according to the invocation block. After the constructor returns, the above invocation block is deleted from memory and the block of the `Rectangle` object now has the following content.

⋮	
300	`Rectangle` object
`width`	1
`height`	2
⋮	

In the third and final part, the local variable `rectangle` is assigned its value. As a consequence, the invocation block of the `main` method can now be depicted as follows.

⋮	
100	`Client.main` invocation
`width`	1
`height`	2
`rectangle`	300
⋮	

Finally, we implement the one-parameter constructor shown in the API. This so-called *copy constructor*[3] enables the client to create a copy of a `Rectangle` object by simply passing that object as an argument, rather than passing the width and height individually as arguments.

```
1 public Rectangle(Rectangle rectangle)
2 {
3    this.width = rectangle.width;
4    this.height = rectangle.height;
5 }
```

Note how this constructor initializes the state of `this` rectangle instance using the state of the passed `rectangle`. Note also that we can access the attributes of the passed instance even though they are private. This is because privacy is a class issue, not an object issue, i.e. there is no privacy across objects belonging to the same class. Note also that we could have used the public accessors of the passed instance instead of using the attributes directly, i.e.

```
1 public Rectangle(Rectangle rectangle)
2 {
3    this.width = rectangle.getWidth();
4    this.height = rectangle.getHeight();
5 }
```

[3]In Java, objects can also be copied using the `clone` method. Since most other object oriented programming languages also support copy constructors, but mostly do not provide a `clone` method, we will restrict our attention to copy constructors in this book.

The two approaches are similar but, as we will see later in this chapter, the second one leads to code that is scalable and easier to understand and maintain.

If we define a class and we do not include any constructor (neither public nor private), then a default constructor, i.e. one that takes no arguments, is added by the compiler. This constuctor is only added if the class does not contain any constructor at all. In this constructor, the attributes are initialized to the default values.

2.2.4 Defining Methods

The method section of the class definition contains one definition per method. The method header has the following general form:

access type signature

As in the case of attributes, the access modifier can be either `public` or `private`.[4] Recall that the *type* is either `void` or the actual type of the return. Recall also that the method's *signature* consists of its name followed by its parameter list.

Which methods appear in the method section of a class definition depends of course on the class and its functionality. Nevertheless, there are common themes such as state-related methods (accessors and mutators), obligatory methods that override those of the `Object` class (such as `toString`, `equals`, and `hashcode`), interface-related methods (such as `compareTo`), and class-specific methods (such as `getArea` and `scale`). Let us take a look at some of these general themes and examine the implementation of each in the `Rectangle` class.

Accessors

An accessor enables the client of the class to gain access to the value of one of the (otherwise invisible) attributes of an instance of the class. For example, the `getWidth` accessor in `Rectangle` returns the value of the width of the rectangle on which the method is invoked. Hence, it is implemented as follows.

```
1 public int getWidth()
2 {
3    return this.width;
4 }
```

It is common to name the accessor of attribute `x` as `getX()` unless the type of `x` is `boolean`. If the type of `x` is `boolean`, then the name `isX()` is generally used for the accessor of `x`.

Mutators

A mutator enables the client to modify, or mutate, the value of one of the attributes of an instance of the class. For example, the `setWidth` mutator in `Rectangle` changes the width of the rectangle on which it is invoked to the passed value. Hence, it is implemented as follows.

[4] The access modifier in Java can also be `protected`, which makes the method visible only to subclasses of this class, or can be left blank, which makes the method visible only to classes in the same package as this class.

```
public void setWidth(int width)
{
   this.width = width;
}
```

It is common to use the name `setX` for the mutator of the attribute `x`. The mutator receives the new value for `x` as an argument.

Based on the requirements, the API designer determines if an attribute should have only an accessor (read-only), only a mutator (write-only), both an accessor and a mutator (read-write), or neither.

The `toString` Method

It is highly recommended that each class overrides the `toString` method of the `Object` class, because its return (a class name and an address) is usually not useful. In a typical API design, the method should return a textual representation of the object on which it is invoked, e.g. its name or elements of its state. For example, the `toString` method in `Rectangle` class returns a string that identifies the object as an instance of the `Rectangle` class and provides its width and height:

```
Rectangle of width 1 and height 2
```

The method can be implemented as follows.

```
public String toString()
{
   return "Rectangle of width " + this.width + " and height " + this.height;
}
```

The `equals` Method

It is also highly recommended that each class overrides the `equals` method of the `Object` class, because its return is based on the equality of memory addresses (i.e. two objects are equal if and only if they have the same memory address). Typically, the `equals` method should compare the states of the objects, rather than their memory addresses. For example, two rectangles are considered equal if they have the same height and the same width. Whatever the definition of equality is, the `equals` method must satisfy the following properties:

- reflexive: `x.equals(x)` returns true for any `x` different from null,
- symmetric: `x.equals(y)` returns true if and only if `y.equals(x)` returns true for all `x` and `y` different from null,
- transitive: if `x.equals(y)` returns true and `y.equals(z)` returns true then `x.equals(z)` returns true for all `x`, `y` and `z` different from null,
- `x.equals(null)` returns `false` for all `x` different from null.

One of the peculiarities of the `equals` method is that its parameter is of type `Object`. Consider, for example, our class `Rectangle`. If the parameter of the `equals` method were of type `Rectangle`, then this `equals` method would not override the `equals` method of the `Object` class (in that case, the `Rectangle` class would have two `equals` methods).

Since the parameter of the `equals` method is of type `Object`, we must first ensure that the passed argument is of the same type as the class being implemented. This can be done in two ways: either using the `getClass` method or exploiting `instanceof`. We will always take the former approach. Later in this book, we will discuss why using the `getClass` method is better than exploiting `instanceof`.

An instance of the class `Class`, which is part of the package `java.lang`, represents a class. For each class, there is a unique instance of the class `Class` that represents it. For example, there exists a unique `Class` object that represents the `Rectangle` class. Given a `Rectangle` object named `rectangle`, we can get the `Class` object that represents the `Rectangle` class by `rectangle.getClass()`. Since each class is represented by a unique `Class` object, we can use `x.getClass() == y.getClass()` to test if `x` and `y` are instances of the same class.

If the passed object is of the same type as the object on which the `equals` method is invoked, then we can compare the states of the object. For example, for the `Rectangle` class the `equals` method can be implemented as follows.

```
1  public boolean equals(Object object)
2  {
3     boolean equal;
4     if (object != null && this.getClass() == object.getClass())
5     {
6        Rectangle other = (Rectangle) object;
7        equal = (this.width == other.width) && (this.height == other.height);
8     }
9     else
10    {
11       equal = false;
12    }
13    return equal;
14 }
```

Note that we first had to ensure that the parameter is not null. Otherwise, the invocation `other.getClass()` may give rise to a `NullPointerException`.

One may wonder why we cast `object` to a `Rectangle` in line 6, since we already checked in line 4 that `object` is a `Rectangle`. Since `object` is declared to be of type `Object` and the class `Object` does not contain the attributes `width` and `height`, the compiler would report errors such as

```
Rectangle.java: cannot find symbol
symbol  : variable width
location: class java.lang.Object
                equal = (this.width == object.width) && (this.height == object.height);
```

if `object` were not cast to a `Rectangle` (more precisely, if we were to remove line 6 and replace `other` with `object` in line 7).

Note also the role played by the `equal` local variable: one could implement `equals` without it but this would lead to multiple return statements in the method, which may make the code harder to maintain. In this book, all non-void methods will have a *single* `return` statement at the very end of their body.

We leave it to the reader to check that our `equals` method of the `Rectangle` class satisfies the four properties mentioned above.

The `compareTo` Method

The API designer may want to impose an order relation on the instances of a class so that, for example, a list of them can be sorted. To indicate that instances of the class `C` have such a *natural order*, the class `C` is made to implement the `Comparable<C>` interface. This interface has a single method which takes an instance of class `C` as its argument and returns in integer. It returns a negative, zero, or positive integer depending on whether the instance on which it is invoked is smaller, equal, or greater than the passed instance. The exact definition of "smaller" and "greater" depends on the class but it must obey the following conditions:

- `x.compareTo(y)` returns 0 if and only if `y.compareTo(x)` returns 0,
- `x.compareTo(y)` returns a value smaller than 0 if and only if `y.compareTo(x)` returns a value greater than 0,
- `x.compareTo(y)` throws an exception if and only if `y.compareTo(x)` throws an exception,
- if `x.compareTo(y)` returns a value smaller than 0 and `y.compareTo(z)` returns a value smaller than 0 then `x.compareTo(z)` returns a value smaller than 0, and
- if `x.compareTo(y)` returns 0 and `y.compareTo(z)` returns 0 then `x.compareTo(z)` returns 0.

Furthermore, the definition in almost all APIs ensures that `x.compareTo(y)` returns 0 if and only if `x.equal(y)` returns true.

The API of the `compareTo` method in `Rectangle` defines a rectangle as being smaller than another rectangle if its width is smaller than the width of the other rectangle, or they share the same width but the height of the first rectangle is smaller than the height of the other rectangle. Based on this, the method can be implemented as follows.

```
1 public int compareTo(Rectangle rectangle)
2 {
3    int difference;
4    if (this.width != rectangle.width)
5    {
6        difference = this.width - rectangle.width;
7    }
```

```
8     else
9     {
10        difference = this.height - rectangle.height;
11    }
12    return difference;
13 }
```

Since the interface `Comparable` is generic, the `compareTo` method does not share the peculiarity of the `equals` method in that its parameter is not of type `Object`, but of the same type as its class. This is why its implementation is simpler as it neither has to use the `getClass` method nor has to cast. Note also that the implementation does not have to validate the incoming parameter for `null` even though the body will throw a `NullPointerException` in that case. This is because the API of the `Comparable` class states that a `NullPointerException` must be thrown if the argument is null (cf. third property above).

Besides implementing the `compareTo` method, we also add `implements Comparable<Rectangle>` to the header of the class.

Class-Specific Methods

The methods `getArea` and `scale` in the `Rectangle` class are examples of class-specific methods.

The `getArea` method returns the area of the rectangle on which it is invoked and can be implemented as follows.

```
1 public int getArea()
2 {
3    return this.width * this.height;
4 }
```

The `scale` method takes a scale factor, which is a non-negative integer according to the precondition, and scales the width and height of the rectangle on which it is invoked by that factor. This method can be implemented as follows.

```
1 public void scale(int factor)
2 {
3    this.width = this.width * factor;
4    this.height = this.height * factor;
5 }
```

The complete class can be found by following this link.

2.2.5 Documenting a Class

Just like utility classes, non-utility classes have two types of documentation: internal (intended for implementers) to explain *how* the class works, and external (intended for clients) to explain *what* the class does. Both types are embedded in the class definition (rather than kept in separate files) to ensure that code and documentation remain in synch.

The `javadoc` utility is used to extract external documentation and format it as HTML. The tags that we studied for utilities, including `@param`, `@pre.`, `@return`, and `@throws`, apply as-is to non-utility classes.

As an example, consider the width mutator of our `Rectangle` class. Its documenation can be done as shown below.

```
/**
   Sets the width of this rectangle to the given width.

   @param width The new width of this rectangle.
   @pre. width >= 0
*/
public void setWidth(int width)
```

As we can see, generating an API for a non-utility class follows the exact same rules as those for a utility class. This should not be surprising because the main difference between these two types of classes is state, and state, being private, does not show up in any external documentation.

The completely documented class can be found by following this link.

2.2.6 Testing a Class

There are two main differences between testing a (non-utility) class and testing a utility class. First of all, we also need to check that the constructors behave according to their specifications. Secondly, for each non-static method, each test case not only consists of values for the parameters but also a value for the implicit parameter `this`.

Since the class may have several constructors, our test app needs to be conducted using each of them separately in order to test every possible path in the code, i.e. full *white-box* testing.

As an example, let us write a tester for the two-parameter constructor and the accessors of the `Rectangle` class. According to its precondition, the two-parameter constructor only accepts widths and heights that are greater than or equal to 0. Hence, we randomly generate two integers between 0 and, say, 100.

```
Random random = new Random();
final int MAX = 100;
int width = random.nextInt(MAX + 1);
int height = random.nextInt(MAX + 1);
```

Our test vector for the two-parameter constructor will consist of several, say 100, of such `(width, height)` pairs. Next, we invoke the two-parameter constructor for each test case.

```
Rectangle rectangle = new Rectangle(width, height);
```

Finally, we combine the test case `(width, height)` with the created `Rectangle` object `rectangle` into a triple `(width, height, rectangle)`. This will be a test case for the accessors. Note that it also includes a value for the implicit parameter `this`. We choose direct verification as our oracle.

```
if (rectangle.getWidth() != width)
{
   output.print("Either the two-parameter constructor or the method getWidth
      failed the test case: ");
   output.println("width = " + width + ", height = " + height);
}
if (rectangle.getHeight() != height)
{
   output.print("Either the two-parameter constructor or the method getHeight
      failed the test case: ");
   output.println("width = " + width + ", height = " + height);
}
```

The full tester can be found by following this link.

2.3 Beyond the Basics

2.3.1 Avoiding Code Duplication

Generally, we try to avoid code duplication. Code containing duplicated fragments is usually more difficult to maintain. Here, we focus on those fragments of code that involve the attributes. These fragments may need to be changed when we decide to represent the state of an object in a different way. For example, rather than using a pair of integers we may decide to represent the state of a rectangle by means of a string consisting of the width and the height separated by a space. Having no duplicated code will usually minimize the number of changes to the code. For those fragments involving attributes we present three different techniques to avoid code duplication.

Constructor Chaining

Consider, for example, the work done by the default constructor of the `Rectangle` class.

```
public Rectangle()
{
   this.width = 0;
   this.height = 0;
}
```

We see that it initializes the attributes `width` and `height` to 0. This code is very similar to the two-parameter constructor which also initializes the attributes `width` and `height`.

```
public Rectangle(int width, int height)
{
   this.width = width;
   this.height = height;
}
```

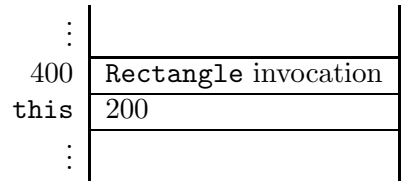
The default constructor can be seen as a special case of the two-parameter constructor. We can avoid some code duplication by rewriting the default constructor so that it merely delegates to the two-parameter constructor. As we have seen in Section 2.2.3, we store three pieces of information in the invocation block: the values of the parameters `width` (0 in this case) and `height`(0 in this case) and the value of the implicit parameter `this`. These three ingredients can be combined as follows.

```
1 public Rectangle()
2 {
3    this(0, 0);
4 }
```

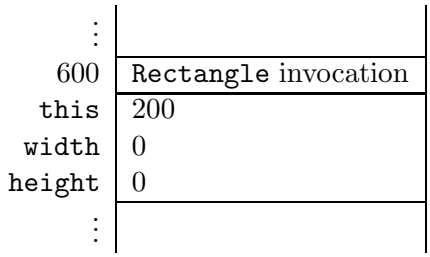
Consider the following fragment of client code.

```
1 Rectangle empty = new Rectangle();
```

Assume that the block for the new `Rectangle` object is allocated at address 200. Then the invocation of the constructor gives rise to the following block in memory.



The execution of the body of the default constructor gives rise to the invocation of the two-parameter constructor, which is reflected by the following block in memory.



Subsquently, the execution of the body of the two-parameter constructor results in the attributes `width` and `height` of the `Rectangle` object at address 200 being set to zero.

Having one constructor delegating to another is known as *constructor chaining*. Similarly, we see that our implementation of the copy constructor duplicates some code. Again we can exploit constructor chaining to remove this redundancy.

```
1 public Rectangle(Rectangle rectangle)
2 {
3    this(rectangle.width, rectangle.height);
4 }
```

Hence, our first technique employs the `this(...)` construct to make all constructors delegate to one. The Java compiler requires that the `this(...)` statement be the very first in the body of a constructor.

Delegating to Mutators

Initially, we implemented the two-parameter constructor as follows.

```
1 public Rectangle(int width, int height)
2 {
3    this.width = width;
4    this.height = height;
5 }
```

Note that the above constructor duplicates the code of the mutators. Hence, we can rewrite this constructor as follows.

```
1 public Rectangle(int width, int height)
2 {
3    this.setWidth(width);
4    this.setHeight(height);
5 }
```

After having applied these two techniques, none of the constructors directly access the attributes and, hence, none is vulnerable to changes of the representation of those attributes. For example, if we decide to change the representation of the attributes from a pair of integers to a string of two integers separated by a space, then none of the constructors need to be refactored.

Delegating to Accessors

Our third technique ensures that attributes are only directly read in the accessors. For example, we implemented the `getArea` method of the `Rectangle` class as follows.

```
1 public int getArea()
2 {
3    return this.width * this.height;
4 }
```

Direct references to the attributes make this method vulnerable to changes of the representation of the attributes. We can therefore employ our third technique and rewrite the method as follows.

```
1 public int getArea()
2 {
3    return this.getWidth() * this.getHeight();
4 }
```

We can also apply this technique to the copy constructor. This results in the following constructor.

```
1 public Rectangle(Rectangle rectangle)
2 {
3    this(rectangle.getWidth(), rectangle.getHeight());
4 }
```

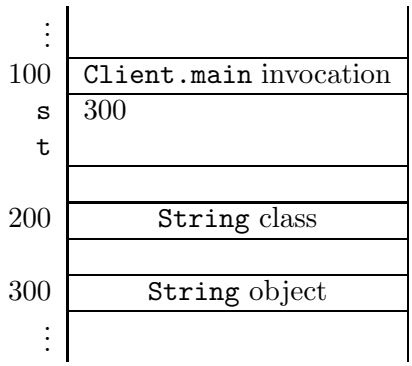
The refactored class can be found by following this link.

2.3.2 Immutable Objects

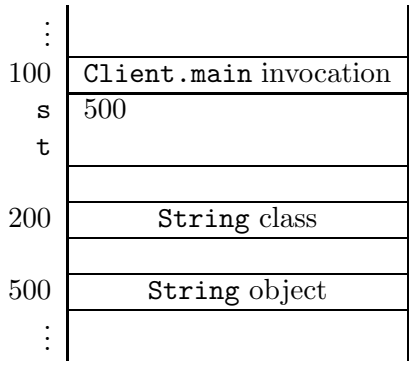
Consider the following snippet of client code.

```
1 String s = "yo";
2 s = s + s;
3 String t = "yoyo";
```

When the execution reaches the end of line 1, memory can be depicted as follows.



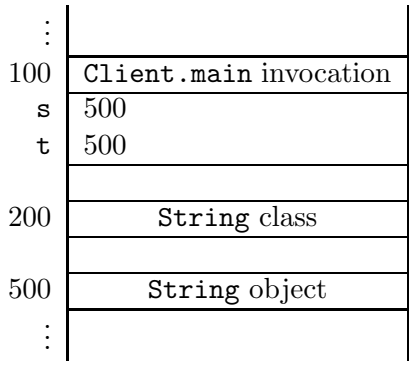
The state of the `String` object at address 300 reflects that this object represents the string `"yo"`. Somewhere in memory the characters `'y'` and `'o'` are stored. In line 2, the length of the string is doubled. However, when we create a `String` object, we cannot always predict its maximal length (it may, for example, depend on input provided by the user). No matter how big a memory block we initially allocate for the `String` object, it may not be big enough and, hence, we may have to allocate a new block. Assigning huge blocks of memory to `String` objects may be a waste of memory. Therefore, whenever we want to change a string, we create a new `String` object. For example, once the execution reaches the end of line 2, memory can depicted as follows.



Instances of the `String` class are *immutable*; i.e. once a `String` object is created, it can no longer be changed. Methods such as `substring` and `toUpperCase` do not mutate the `String` object on which they are invoked; they simply return a brand new `String` object.

The class `String` is called immutable since its instances are. Wrapper classes such as `Integer` and `Double` are other examples of immutable classes.

Since `String` objects are immutable, there is no need to create a new `String` object for the variable `t` in line 3 of the above code snippet. Once the execution reaches the end of line 3, our memory diagram looks as follows.



API designers may demand immutability either because of the nature of the class or, as is the case for the `String` class, for efficiency reasons. Regardless, we need to be able to implement such immutable classes.

As an example, assume that our `Rectangle` class is supposed to be immutable. At first glance, we may be tempted to simply delete all its mutators. However, this may break other parts of the implementation, in particular those parts that rely on (i.e. delegate to) these mutators. In fact, recall that in Section 2.3.1 we avoided code duplication by delegating to mutators. Hence, rather than deleting all mutators we can simply make them private. For example, the width mutator remains as before except for its access modifier:

```
1 private void setWidth(int width)
2 {
3     this.width = width;
4 }
```

2.3.3 Attribute Caching

When we make design choices while we are identifying the attributes of our class, we often opt for a *minimal* set of attributes that capture the state of an instance. For example, the state of a `Rectangle` object involves a width and a height and this suggests using two integers as attributes. The API of `Rectangle` does have a method named `getArea`. However, adding an attribute named `area`, which holds the area of the rectangle, would lead to redundant information. This information can easily be computed from the width and the height, as we do in the `getArea` method. There is no need to treat it as part of the state; i.e. it needs not be computed and stored for every instance.

Generally, we try to keep the attributes independent (and thus minimal) because this is consistent with the general principle of avoiding redundancy. There are situations, however, in which a redundant attribute is kept because recomputing it requires extensive resources. This process is

known as *caching* the attribute value, and the attribute itself is known as a *cache*. As an example, let us add the following (admittedly artificial) method to the API of our `Rectangle` class:

```
public int getGreatestCommonDivisor()
```

This method returns the greatest common divisor of the width and the height of a rectangle. Rather than computing the greatest common divisor of the width and height everytime the method `getGreatestCommonDivisor` is invoked, we cache it. That is, we introduce an attribute, say `greatestCommonDivisor`.

```
1 private int greatestCommonDivisor;
```

The method `getGreatestCommonDivisor` simply returns the value of this attribute.

```
1 public int getGreatestCommonDivisor()
2 {
3    return this.greatestCommonDivisor;
4 }
```

Whenever the state of a rectangle changes, the value of the attribute `getGreatestCommonDivisor` needs to be recomputed. Hence, this should be done in both mutators. To avoid code duplication, we introduce a private method.

```
1 private void setGreatestCommonDivisor()
2 {
3    // assign to this.greatestCommonDivisor the greatest common divisor
4    // of this.width and this.height
5 }
```

This method is invoked in both mutators. For example, the mutator for the attribute `width` is modified resulting in the following.

```
1 public void setWidth(int width)
2 {
3    this.width = width;
4    this.setGreatestCommonDivisor();
5 }
```

Note that this is an example of an attribute the accessor of which is public and the mutator of which is private.

2.3.4 Class Invariants

We may want to document that a property about the state of each instance of a particular class "always" holds. For example, we may want to express that the width and the height of a `Rectangle` object are greater than or equal to 0. Such a property can be documented as a *class invariant*. A class invariant is a predicate involving the attributes of the class. For example, the class invariant that expresses the width and the height of a `Rectangle` object to be greater than or equal to 0 can be documented as follows in the class `Rectangle`.

```
1 /* Class invariant: this.width >= 0 && this.height >= 0 */
```

Note that the above is not a documentation comment and, hence, will not show up in the API.

A class invariant has to satisfy the following two constraints.

- The class invariant has to be true after each public constructor invocation, provided that the client ensures that the precondition of the invoked constructor is met. That is, if the class invariant holds when the method is invoked and the precondition of the method holds as well, then the class invariant also holds when the method returns.[5] Let us check this constraint for the above class invariant. Obviously, the width and the height of a `Rectangle` object are greater than or equal to 0 when it is created by the default constructor. If we create a `Rectangle` object using the two-parameter constructor, then the precondition of this constructor implies that the width and the height are greater than or equal to 0. Finally, assume that we create a `Rectangle` object by means of the copy constructor. From the class invariant we can infer that the width and the height of the `Rectangle` object that is provided as an argument to the copy constructor are greater than or equal to 0. Hence, also the width and the height of the created `Rectangle` object are greater than or equal to 0.

- The class invariant has to be maintained by each public method invocation, provided that the client ensures that the precondition of the invoked method is met. Again, let us check this constraint for the above class invariant. Obviously, we only have to consider those public methods that change the attributes `width` or `height`. Hence, we only have to consider the methods `setWidth`, `setHeight` and `scale`. For the mutators, the preconditions imply that the new width and the new height are greater than or equal to 0. Finally, let us consider the `scale` method. To check that this method maintains the class invariant, assume that the class invariant holds, i.e. the width and height are greater than or equal to 0, and its precondition is met, i.e. `factor` ≥ 0. Then, the new width and the new height are of course also greater than or equal to 0.

While testing a (non-utility) class, we may want to check that the class invariant holds after each invocation of a public constructor and is maintained by each invocation of a public method.

2.3.5 The `hashCode` Method

One of the obligatory methods that every class inherits (or overrides) from the `Object` class is the `hashCode` method. It returns an integer that acts as an identifier for the object on which the method is invoked. A `hashCode` method implements a hash function, which maps objects to integers, and the method is used in classes such as `HashSet` and `HashMap`. Full treatment of hash functions is beyond the scope of this book.

In whatever way we map objects to integers, the `hashCode` method has to satisfy the following property:

- if `x.equals(y)` returns true then `x.hashCode()` and `y.hashCode()` return the same integer for all `x` and `y` different from null.

[5] The class invariant does not need to hold continuously when the body of the method is executed.

The opposite is not required, i.e. if two objects are unequal according to the `equals` method, then the `hashCode` method need not return distinct integers. However, we should be aware that returning distinct integers for unequal objects may improve the performance of methods in classes such as `HashSet` and `HashMap`.

For the wrapper class `Integer`, the `hashCode` method can be implemented as follows.

```
1 public int hashCode()
2 {
3    return this.value;
4 }
```

where the attribute `value` hold the `int` value of the `Integer` object. In this case, the `hashCode` maps two objects to the same integer if and only if the objects are the same according to the `equals` method. However, such a one-to-one correspondence is not always possible. For example, the wrapper class `Double` has 2^{64} different instances, but there are only 2^{32} different integers.

Now let us return to our `Rectangle` class. Let us first choose *not* to implement it (i.e. use the one inherited from `Object`) and consider the following client fragment:

```
1 Set<Rectangle> set = new HashSet<Rectangle>();
2 set.add(new Rectangle());
3 set.add(new Rectangle());
4 output.println(set.size());
```

Since the two added rectangles are equal, one expects the set to have only one element because sets keep only distinct elements. This fragment, however, leads to both elements being added to the set; i.e. the set will consist of elements that are deemed equal by the `equals` method! This example shows that it is critical to override the `hashCode` method and ensure that its implementation satisfies the above mentioned property.[6] For our `Rectangle` class, we can implement the `hashCode` method as follows.

```
1 public int hashCode()
2 {
3    return this.getWidth() + this.getHeight();
4 }
```

This ensures that when invoked on two rectangles of equal width and height, the method returns the same integer. Such an implementation meets the equality requirement and will indeed cause the above client fragment to store one and only one element in the set. It does not, however, meet the one-to-one correspondence. For example, the rectangle with width 2 and height 3 and the rectangle with width 4 and height 1 both return the same hash code, namely 5, even though they are different rectangles. Also in this case, no one-to-one correspondence is feasible since there are 2^{62} different rectangles, but only 2^{32} different integers.

[6] The `hashCode` method of the `Object` class satisfies the property with respect to the `equals` method of the `Object` class. Since we did override the `equals` method in the `Rectangle` class, we also have to override the `hashCode` method.

2.3.6 Validation and the Implementer

As we have already seen in Section 1.4.2, the implementer can validate the input to a method in different ways. Here, we revisit this topic for the mutator of the attribute `width`. In the original API of the `Rectangle` class, the parameter `width` is restricted to a non-negative integer by means of a precondition. Next, we present two alternative designs.

As in Section 1.4.2, instead of a precondition, we can use an exception. In this case, we throw an `IllegalArgumentException` whenever the value of the parameter `width` is smaller than 0. This can be implemented as follows.

```
1  public void setWidth(int width) throws IllegalArgumentException
2  {
3     if (width < 0)
4     {
5        throw new IllegalArgumentException("Argument of setWidth method cannot
             be negative");
6     }
7     else
8     {
9        this.width = width;
10    }
11 }
```

This method can be documented as follows.

```
1  /**
2     Sets the width of this rectangle to the given width.
3
4     @param width The new width of this rectangle.
5     @throws IllegalArgumentException if width < 0.
6  */
```

The above documentation comment and the method header give rise to the following snippet of the API.

setWidth

```
public void setWidth(int width)
              throws IllegalArgumentException
```

Sets the width of this rectangle to the given width.

Parameters:
`width` - The new width of this rectangle.
Throws:
`IllegalArgumentException` - if width < 0.

As an alternative, we may decide to set the attribute `width` to its new value only if that value is non-negative. Whether the attribute is set to a new value is returned to the client in the form of a boolean. This alternative design can be implemented as follows.

```
1 public boolean setWidth(int width)
2 {
3    boolean isSet = width >= 0;
4    if (isSet)
5    {
6       this.width = width;
7    }
8    return isSet;
9 }
```

This method can be documented as follows.

```
1 /**
2    Sets the width of this rectangle to the given width if
3    the given width is greater than or equal to 0.  Returns
4    whether the width has been set.
5 
6    @param width The new width of this rectangle.
7    @return true if width <= 0, false otherwise.
8 */
```

The above documentation comment and the method header give rise to the following snippet of the API.

setWidth

`public boolean` **`setWidth`**`(int width)`

Sets the width of this rectangle to the given width if the given width is greater than or equal to 0. Returns whether the width has been set.

Parameters:
`width` - The new width of this rectangle.
Returns:
true if width <= 0, false otherwise.

2.3.7 Check your Understanding

Refactor the `Rectangle` class in the following ways.

- Instead of using a pair of integers to represent the state of a rectangle, use a string consisting of the width and the height separated by a space. For example, a rectangle with width 1 and height 2 is represented by a single attribute the value of which is `"1 2"`.
- Instead of using a pair of `int`s to represent the state of a rectangle, use a `long`. For example, a rectangle with width 1 and height 2 is represented by a single attribute the value of which is $1 \times 2^{32} + 2$.