Chapter 1

Implementing Static Features

1.1 Introduction

1.1.1 What are Utilities?

Recall that a class is an entity that has attributes and methods. Most classes *cannot* be used unless they are instantiated first. The instantiation process creates copies of the class attributes, and this allows different instances to have different attribute values, i.e. different states.

But what if a class has no attributes at all; i.e. its instances are stateless? Or, what if all instances of a class have the same attribute values, i.e. the same state? In both cases, there is no need to create instances because doing so would waste valuable resources. Such a class is called a *utility*, and in it, the attributes are associated with the class itself, not with instances. Indeed, we *cannot* instantiate a utility and we will *not* find a constructor section in its API.

In Java, we recognize a utility from its API by noting that all its features (both attributes and methods) are `static`.

Since a utility cannot be instantiated, its client must access its public attributes and invoke its public methods by using the class name. As an example, consider the `Math` utility whose API is shown here and which is a simplified version of the one in the package `java.lang`. The following fragment shows how a client can use this utility.

```
1  output.println("The value of PI is: " + Math.PI);
2
3  output.println("Enter two integers:");
4  int a = input.nextInt();
5  int b = input.nextInt();
6  output.println("The smaller of the two entries is: " + Math.min(a, b));
7
8  output.println("Enter a base and an exponent:");
9  int base = input.nextInt();
10 int exponent = input.nextInt();
11 output.println("The entered power is: " + Math.pow(base, exponent));
```

The full example can be found by following this link.

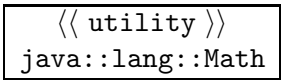
1.1.2 Why are Utilities Needed?

A utility is used in situations in which there is no need to have multiple copies of the class features; i.e. a single copy suffices. For example, a class that does not hold any state (i.e. has no attributes) is a good candidates for a utility. A class without attributes has no state so it cannot remember anything about previous invocations, e.g. which method was invoked last, when was a method invoked, or what arguments were passed before. A method in such a class has therefore nothing to work with other than the arguments passed to it, and its return (if any) must depend only on the values of these arguments. A second example of a class that is appropriate for a utility is one that has a fixed state (i.e. all its attributes are `static` and `final`).

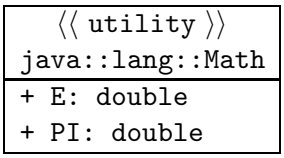
It should be noted that most, if not all, classes that make up a real-life application are *not* utilities. We start with utilities because they are simpler than non-utilities. Moreover, the skills we will acquire in implementing them apply as-is to non-utility classes.

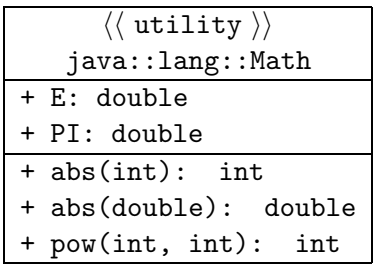
1.1.3 UML and Memory Diagrams

The *Unified Modeling Language* (UML) allows us to represent utilities through diagrams. A minimal UML class diagram consists of a rectangular box that contains the name of the class preceded by the word *utility* between two guillemets, as shown below. The class name can be fully qualified, with its package and sub-package names, but with a double-colon (rather than a dot) as the separator character.

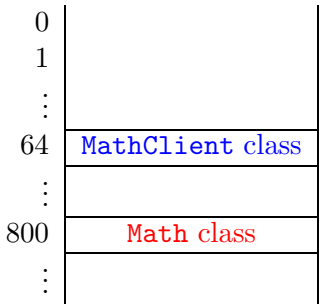


If additional features need to be exposed then a second (and a third) box can be added to hold the attributes (and methods) of the utility, as shown below. The + sign appearing before a feature indicates that the feature is public.

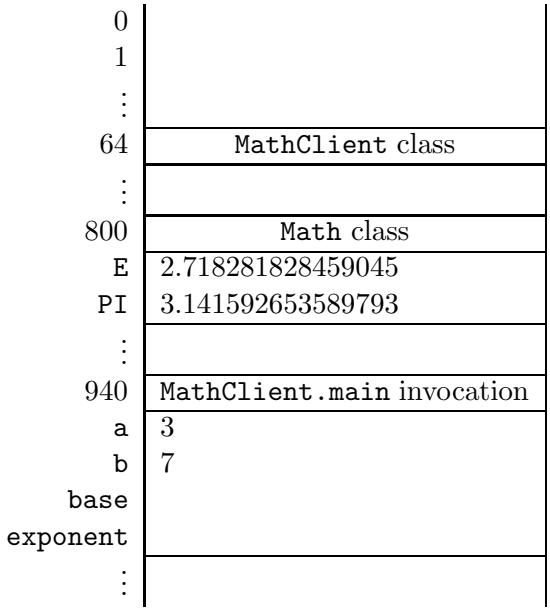




In addition to UML diagrams, we sometimes use memory diagrams to show how different classes reside in memory. For example, the memory diagram below contains two class blocks representating the client class named `MathClient` residing in memory beginning at address `64` and the `Math` utility at address `800`. The addresses used in the figure are of course arbitrary.



More details can be provided in these memory diagrams. For example, in the memory diagram below the class block of the `Math` utility also contains the constants `E` and `PI`. Furthermore, this more detailed memory diagram contains an invocation block representing the invocation of the `main` method of the `MathClient` class. This diagram models memory after the user has entered the values 3 and 7 but before the user has entered the values for the base and exponent.



1.1.4 Check your Understanding

Consider the following classes:

- `Arrays` in the `java.util` package,
- `DocumentBuilderFactory` in the `javax.xml.parsers` package,

- `Collections` in the `java.util` package and
- `InputStream` in the `java.io` package.

If you attempt to use the typical `new` keyword to instantiate any of these classes, you will find that the attempt will fail. Does this mean that these classes are utilities? Argue that only two of them are utilities.

1.2 How: The Class Definition of a Utility

1.2.1 Class Structure

Implementing a utility means writing its so-called *class definition*. Here is the general form:

```
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 definition starts with a `package` statement if this class belongs to a named package. This is then followed by any needed `import` statements as is the case of client apps. The body of the class has three sections, one for attributes, one for constructors, and one for methods, which is reminiscent of the API structure. These three sections may appear in any order but, as a matter of style, we will adhere to the above order.

1.2.2 Declaring and Initializing Attributes

The state of a utility is held in its attributes. If a utility does not have state, the attribute section is omitted. Otherwise, it contains one definition per attribute. The definition has one of the two following general forms:

access `static` *type name* `=` *value*`;`

or

access `static final` *type name* `=` *value*`;`

It should not be surprising that the keyword `static` is present; after all, we are studying utilities in this chapter and all features in a utility are `static`. Here is an explanation of each of the remaining elements:

- *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` (appearing in the second general form above) is used if this attribute is a constant.
- *type* specifies the type of this attribute. It can be primitive or non-primitive.
- *value* specifies the initial value of this attribute and should be compatible with its type; i.e. this should be a valid assignment statement.

Here is an example:

```
1 public class Math
2 {
3    public static final double PI = 3.141592653589793;
4
5    ...
6 }
```

When we define attributes in a utility, we should keep two things in mind:

- It is best to initialize attributes as we declare them. If we delay or forget their initialization, the compiler will *not* issue a compile-time error; it will simply assign default values to them. Relying on such defaults is not a good idea because they may change from one version of the language to the next. Furthermore, explicit declaration mitigates *assumption risks* (scenarios in which people make implicit assumptions and do not communicate them, thinking they are obvious), which can lead to logic errors.
- The scope of an attribute is the entire class.

Finally, even though attributes have types and hold values, they are different from local variables. [2]

1.2.3 The Constructor Section

The constructor section of a utility named `SomeName` consists of a single statement:

[1] The access modifier in Java can also be left blank. This makes the attribute visible only to classes in the same package as this class (more on this later).

[2] Local variables can appear only within the definition of a method or a constructor, and their scope is limited to the definition in which they appear. In addition, local variables must be initialized before being used; i.e. the compiler does not assign default values to them.

```
private SomeName(){}
```

It is not surprising that the constructor has the same name as the class but this code looks rather odd: why is the constructor private and why is it empty? And if it is meant to be empty then why is it included at all in the definition?

Recall that utilities are never instantiated, and hence, there should not be a constructor section in their APIs. This implies that there should not be any public constructor in the class definition of a utility. But when the Java compiler encounters a class without any constructor at all, it automatically adds a *public* constructor to it. To prevent this from happening, we add the above private constructor. Its body is empty because we really do not need a constructor; it is there only to thwart the compiler's attempt at adding a public one.

Based on this, the class definition of our example utility becomes:

```
public class Math
{
   public static final double PI = 3.141592653589793;

   private Math(){}

   ...
}
```

1.2.4 Defining Methods

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

access `static` *type name parameter-list*

or

access `static` *type name parameter-list* `throws` *type*

As in the case of attributes, it should not be surprising that the keyword `static` is present. And also as for attributes, the access modifier can be either `public` or `private`. [3] Recall that the return *type* is either `void` or the actual type of the return.

Let us look at some of the methods of the `Math` class. Consider `min(int, int)`, which returns the smaller of its two arguments. It is a `public` method with a return type of `int`. Hence, its definition must have the following header:

```
public static int min(int a, int b)
```

[3] The access modifier in Java can also be left blank. This makes the method visible only to classes in the same package as this class (more on this later).

When a client program invokes this method by writing something like `Math.min(3, 5)`, control is transferred to the method with `a` initialized to `3` and `b` initialized to `5`. The body of the method does the real work. It should figure out which parameter (`a` or `b`) is smaller and return it to the client. Returning the result to the client is expressed by means of the *return* statement. This statement consists of the keyword `return` followed by an expression. The type of the expression should be compatible with the return type of the method. The execution of the return statement results in the value of the expression being returned to the client. Here is a possible implementation of the `min` method:

```
1  int min;
2  if (a <= b)
3  {
4     min = a;
5  }
6  else
7  {
8     min = b;
9  }
10 return min;
```

In fact, we can implement this functionality in one statement:

```
1  public static int min(int a, int b)
2  {
3     return (a <= b) ? a : b;
4  }
```

Next, let us look at the `pow` method. It is a `public` method that takes two `int` arguments and return an `int` (being the first argument raised to the second). Hence, its definition must have this layout:

```
1  public static int pow(int a, int b)
2  {
3     ...
4  }
```

In the body of this method we must compute and return $\texttt{a}^{\texttt{b}}$, i.e.

$$\underbrace{\texttt{a} \times \texttt{a} \times \cdots \times \texttt{a}}_{\texttt{b} \text{ times}}$$

To do this, we need to set up a loop. And since we know how many times the loop should iterate, we use a `for` loop:

```
1  int pow = ?
2  for (int i = 0; i < ?; i++)
3  {
```

```
4       pow = pow * a;
5   }
6   return pow;
```

Next, we need to determine the initial value of the local variable `pow` and the number of iterations of the loop. A *loop invariant* can help us do this. Recall that a loop invariant is a boolean expression that holds at the beginning of every iteration. It usually tells us something about the variables that play a key role in the loop. In the above loop, the variables `pow` and `i` play a key role. They are related as follows: $\texttt{pow} = \texttt{a}^{\texttt{i}}$. We will take this as our loop invariant. Now we can determine the initial value of `pow`. Since the loop invariant has to hold at the beginning of the first iteration of the loop and since `i` is initialized to `0`, we can conclude that we have to initialize `pow` to `1` since $\texttt{a}^0 = 1$. The loop invariant also tells us when to exit the loop. We want to end the loop if $\texttt{pow} = \texttt{a}^{\texttt{b}}$. So when we reach an iteration at which `i` = `b`, the loop condition should evaluate to false. Therefore, as long as `i != b` (or `i < b`) we should continue the loop. The complete implementation of our `Math` utility can be found by following this link.

1.2.5 Testing a Utility

In order to test a utility we need to write a client class, i.e. an app, that invokes every method in it and verifies that it behaves according to its specification. In the lingo of the *software development process* such a test is known as a *unit test* as opposed to an *integration test*, which is performed after all classes in the application have been built.

Testing a method in a utility involves generating a test vector (a collection of test cases that meet the precondition of the method) and then invoking the method for each test case. The method's return is then checked against an oracle to ensure that the method's postcondition is met.[4]

As an example, let us write a unit test for our `Math` class and let us focus only on its `min(int, int)` method. Since any `int` is a valid argument for this method, we will use the `nextInt()` method of the `java.util.Random` class to generate random arguments that are uniformly distributed over the entire `int` range. The following fragment demonstrates this step:

```
1  Random random = new Random();
2  int a = random.nextInt();
3  int b = random.nextInt();
4  int min = Math.min(a, b);
```

Our test vector will thus consist of several, say 100, of such `(a,b)` pairs.

The next step involves determining if the method satisfies its postcondition; i.e. if `min` is indeed the smaller of `a` and `b`. We will choose direct verification as our oracle:

```
1  if (a < min || b < min || (a != min && b != min))
2  {
3     output.print("The method failed the test case: ");
4     output.println("a = " + a + ", b = " + b);
```

[4] In general, one must also ensure that the values of all fields remain consistent with the class invariant. But since most utilities do not have non-constant fields, we will ignore this part of the test.

```
5 }
```

The above process is repeated for each of the remaining test cases in our test vector. The full tester can be found by following this link.

1.2.6 Check your Understanding

Write a utility that contains the method:

```
1 public static long factorial(int n)
```

The method computes and returns the factorial of the given integer. The return is a `long` (rather than an `int`) because the factorial function grows rapidly and quickly exceeds the `int` range. You can safely assume that the passed integer is not negative (we will see later in this chapter how to better handle this condition). Test your implementation using the integers `[0,20]` as test vector and your calculator as oracle.

1.3 Arguments versus Parameters

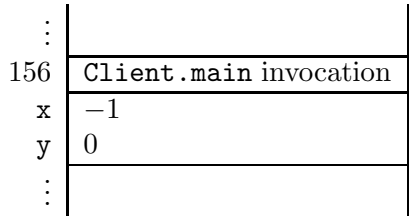
1.3.1 Passing Arguments By Value

Object-oriented programming aims at creating a layer of separation between the client's code and the implementer's code in order to reduce their complexities. The client's code and the implementer's code cannot operate in total isolation, however, because they need to cooperate through method invocation. Hence, a mechanism needs to be in place to facilitate such cooperation without allowing either party to modify the variables of the other party without their explicit knowledge. This mechanism is known as *pass-by-value*. We examine it from the client's view in this section and look at its impact on the implementer in the next section.

Consider the following fragment of client code that invokes the `abs` method of the `Math` class.

```
1 int x = -1;
2 int y = 0;
3 y = Math.abs(x);
```

Having reached the beginning of line 3, our memory model contains the following invocation block.



Assume that the implementer's code of the `abs` method is the following.

```
1 public static int abs(int a)
2 {
```

```
3     int abs;
4     if (a < 0)
5     {
6        abs = -a;
7     }
8     else
9     {
10       abs = a;
11    }
12    return abs;
13 }
```

Once the `abs` method is invoked, an invocation block is added to our memory diagram. When the invocation reaches the beginning of line 12 of the `abs` method, our memory diagram looks as follows.

⋮	
156	`Client.main` invocation
`x`	-1
`y`	0
⋮	
244	`Math.abs` invocation
`a`	-1
`abs`	1
⋮	

Note that `x` and `a` live at different memory locations. When the `abs` method is invoked, the value of `x` (*not* its memory location) is passed. Hence, the name pass-by-value. As a consequence, changes to `a` have no impact on `x`.

What were the values of the client variables `x` and `y` before the invocation and what are their values afterwards? The value of `y` was 0 and it has become 1. Hence, the invocation did result in a change in the client's data but this change did not take place implicitly or secretively; the client asked for it (through an assignment statement) and was fully aware of its consequence (through the method's postcondition). As for `x`, its value was -1 prior to the invocation. If this value were to change afterwards then this would be an implicit change that would take place without the client's knowledge. Fortunately, the pass-by-value mechanism prevents that; i.e. the value of `x` remains -1 after the invocation *regardless* of how `abs` is implemented.

In general, the comma-separated entities that appear between two parentheses after the method name in an invocation statement are known as *arguments*. Hence, `x` in the above invocation is an argument. The fact that the values of the arguments (and not their memory locations) are passed to the implementer guarantees that the arguments remain intact no matter what the implementer does.

It is important to know that the pass-by-value mechanism remains in force even if the argument is an object reference. In other words, an object reference argument is guaranteed not to change

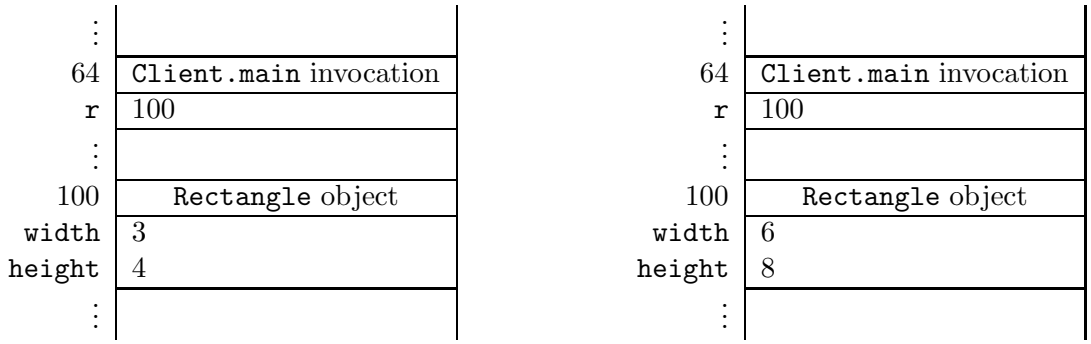
after an invocation (but the object to which it refers might). Let us look at an example that illustrates this point: assume that there is a utility called `Doubler` with the method:

```
1 public static void twice(Rectangle x)
```

The method takes a reference to a `Rectangle` object and doubles the sizes of the rectangle's width and height. The API of the `Rectangle` class can be found following this link. Here is a fragment of a client app that creates a `Rectangle` with width 3 and height 4 and then invokes the above method:

```
1 Rectangle r = new Rectangle(3, 4);
2 Doubler.twice(r);
```

In this example, the client passes the argument `r` to the method. Based on our understanding of the pass-by-value mechanism, the method cannot change `r` but it can change (mutate) the `Rectangle` to which `r` points.



The memory diagram above assumes that the `Rectangle` object resides in a memory block that starts at address 100. The diagram shows that the value of `r` was 100 before the invocation and remained 100 after it, as expected. The object itself was mutated as evidenced by the doubling of its sides. The fact that the object was changed does not contradict the pass-by-value mechanism because the object itself is not an argument; its reference is. [5]

1.3.2 Initializing Parameters

We saw in the previous section that the pass-by-value mechanism allows the client to establish a one-way communication channel with the implementer. The client uses this channel to pass the values of the arguments to the implementer knowing that these values cannot change upon return. We now look at this mechanism from the implementer's side.

The implementer uses the term *parameters* to refer to the variables that appear between parentheses in the method header. For example, the header of the `twice` method that was introduced in the previous section contains one parameter named `x`:

```
1 public static void twice(Rectangle x)
```

[5] Recall that an object is a cluster in memory that belongs to neither the client nor the implementer. Programs can own object references, and they can make the references point at an object, but they do not own objects.

We see that parameters are *declared* in the header but how are they *initialized*? The pass-by-value mechanism states that parameters are initialized to the passed values of the arguments. Hence, the parameter `x` in the above header will have 100 as its initial value (see memory diagram) because this is the value of the argument passed by the client. We can see how the one-way channel is implemented: since the client and the implementer are using two distinct sets of variables linked only through initialization, it is clear that the implementer cannot possibly modify any of the client's arguments.

Let us now implement the `twice` method. We need to determine the sides of the passed `Rectangle` and double them. Here is the implementation:

```
1 public static void twice(Rectangle x)
2 {
3    x.setWidth(2 * x.getWidth());
4    x.setHeight(2 * x.getHeight());
5 }
```

Note that the method receives the value of the object reference (i.e. the address of the object in memory) and then *mutates* the object (i.e. changes its state in place); it does *not* change the address of the object in memory.

As a second example of the impact of pass-by-value on the implementer, suppose a client writes a main method containing the following code snippet.

```
1 int x = 1;
2 int y = 2;
3 Util.swap(x, y); // swaps the values of x and y
```

The implementer is asked to write the method `swap` that allows its client to swap two `int` values:

```
1 public static void swap(int a, int b)
2 {
3    ?
4 }
```

The pass-by-value mechanism makes implementing this method *impossible*! No matter how the values of the two parameters `a` and `b` are interchanged, the arguments will not be affected.

1.3.3 Attribute Shadowing

We saw in the previous section that parameters are declared in the header of the method and are initialized using the passed values of the arguments. Hence, they behave like local variables and have a scope that extends from the very beginning of the method (its opening brace) to its end (its closing brace). Recall that attributes are declared outside all methods and that their scope encompasses the entire class. This raises a question: what if one of the parameters of a method in a class happens to have the same name as an attribute of that class? For example, a utility `Util` may have an attribute named `count` and the method:

```
1 public static void increment(int count)
```

```
{
   count = count + count;
   ...
}
```

Which `count` is being referenced here, the attribute or the parameter? Java in this case gives the priority to the parameter; i.e. the parameter *shadows* the attribute (hides it). The above method will therefore double the value of its parameter, not the attribute. This situation is clearly confusing and can lead to logic errors. Because of this, we will, as a matter of style, always use the utility name when we refer to its attributes. Hence, if the above method were meant to increment the class attribute by the value of the parameter, its code should have been written as follows:

```
public static void increment(int count)
{
   Util.count = Util.count + count;
   ...
}
```

1.3.4 Method Overloading and Binding

The argument-parameter mapping dictates how the compiler should bind an invocation to a method. Given an invocation of the form `C.m(arguments)`, the compiler starts by ensuring that class `C` is available and does have a method named `m`. Next, the compiler examines the method's signature to ensure that the values of the arguments can indeed be assigned to the method's parameters. This requires that the number of arguments is equal to the number of parameters and that the arguments are compatible with their corresponding parameter types.

When a method is overloaded and binding can be made with more than one variant of the method, the compiler tries to limit the possibilities by using the *most specific* policy: select the method that requires the least amount of promotion. If that fails, the compiler issues an *ambiguous invocation* error.

1.3.5 Check your Understanding

Consider the utility `PassByValue` whose source code is shown here. This utility is used by the following client:

```
import java.util.*;
import java.io.PrintStream;

public class PassByValueClient
{
   public static void main(String[] args)
   {
      PrintStream output = System.out;
      int a = 1;
```

```
10        PassByValue.add(a);
11        output.println(a);
12        Set<String> b = new HashSet<String>();
13        PassByValue.add(b);
14        output.println(b.size());
15        List<Integer> c = new ArrayList<Integer>();
16        PassByValue.add(c);
17        output.println(c.size());
18     }
19  }
```

Predict the output of this client. You can check the correctness of your prediction by saving the above client to a file and then compiling it and running it.

1.4 Preconditions versus Validation

1.4.1 Preconditions and the Client

Before invoking a method, a client must ensure that its precondition is met. Failing to do so can lead to unpredictable results. Consider, for example, the following method which computes and returns the factorial of its argument:

```
1  public static long factorial(int x)
```

If the contract of this method specifies the precondition "`x >= 0`", then it is the client's responsibility to ensure that `x` is not negative.

What happens if we invoke this method and pass -1? The result is unpredictable: the implementer may have used an algorithm that crashes if `x` is negative, and in that case an exception will occur. It is also possible that the implementer may have started by computing the absolute value of `x`, and in that case the wrong result, `1`, is returned, and this will likely lead to a logic error in the client's program. What is worse, the result is not guaranteed to be the same every time because the implementer can change the algorithm (in a future release or dynamically for web services) without informing the client (recall that it would break the encapsulation if the implementer informed the client). In conclusion, the client cannot rely on any particular outcome (an exception or a special return) if a method is invoked when its precondition does not hold.

It is clear from the above that the precondition is 100% in the client's concern. The implementer can assume that the precondition is met and should not be concerned with the possibility that it is not.

1.4.2 Validation and the Implementer

If the contract of a method does *not* specify a precondition, the client can invoke it without any validation. Consider, for example, the following method in the `java.lang.Math` class which computes and returns the positive (real) square root of its argument:

```
1 public static double sqrt(double x)
```

The contract (i.e. the API) does not specify a precondition; i.e. the precondition is `true`. Hence, a client can invoke it and pass any `double` value. In particular, the client can pass a negative value for `x` and still expect the method to behave in a predictable manner. Indeed, the API of this method specifies (as part of the postcondition) that if `x` is less than zero then the return is `Double.NaN`, the special `double` that indicates an out-of-range value. Hence, detecting a negative value (i.e. validating the parameters) in the absence of a precondition is 100% in the implementer's concern. The implementation of the above method must therefore have the following structure:

```
1  public static double sqrt(double x)
2  {
3     double sqrt;
4     if (x < 0)
5     {
6        sqrt = Double.NaN;
7     }
8     else
9     {
10       // compute the square root and assign it to sqrt
11    }
12    return sqrt;
13 }
```

Instead of returning a special value, the implementer can also throw an exception.

```
1  public static double sqrt(double x) throws IllegalArgumentException
2  {
3     double sqrt;
4     if (x < 0)
5     {
6        throw new IllegalArgumentException("Argument of sqrt method cannot be
            negative");
7     }
8     else
9     {
10       // compute the square root and assign it to sqrt
11    }
12    return sqrt;
13 }
```

Note that `throws IllegalArgumentException` has been added to the header of the method.

As a matter of good style, one should leave exceptions to exceptional situations. Hence, if the problem is plausible and likely to occur under normal conditions then a special return is preferable (e.g. adding a duplicate element to a `Set`). On the other hand if the problem is unpredictable (i.e.

exceptional) then it is preferable to throw an exception (e.g. connecting to a remote URL while the network is down). No matter what action we choose, we have to make sure it is documented in the postcondition either under *Returns* or *Throws*.

In summary, the onus is on the implementer if no precondition is specified. The implementer must plan for special conditions, determine what to do when they occur, and document the taken action as part of the postcondition.

1.4.3 Check your Understanding

The utility `PreVal` has the API shown here. As you can see from the API, it has three methods: `loadFactor`, `markup`, and `deviation`. `loadFactor` works by invoking `markup`, and `markup` works by invoking `deviation`. In other words, `loadFactor` is a client of `markup`, and `markup` is a client of `deviation`.

Assume that `deviation` is already implemented (for the issue at hand, it does not really matter what it computes and returns). How would you implement `loadFactor` and `markup`? Make sure you properly assume the responsibilities of client and/or implementer when you handle pre- and postconditions.

Compare your implementation with the one shown here.

1.5 Documenting a Utility Class

1.5.1 API Generation through `javadoc`

Utilities (and, in fact, all non-app classes) have two types of documentation: internal and external. Internal documentation appears in the form of one-line comments (prefixed by `//`) or multi-line comments (surrounded by `/*` and `*/`). Its objective is to explain to implementers how the class works. In contrast, external documentation explains to clients what the class does; i.e. its usage. The external documentation of a class is also known as its *contract* or *API*.

It may seem natural to separate the class definition from its external documentation; i.e. store the former in one file (e.g. `Math.java`) and the latter in another (e.g. `Math.html`). After all, these two files have different formats: general text versus HTML. Such separation, however, does not work well because, in real life, one finds that the API changes as the class is being implemented. The person maintaining the API may make changes to it and forget to inform the person maintaining the source file, and vice versa. Such loss of synchronization leads to an API that describes a certain behaviour while the actual class has a different behaviour.

The ideal solution is to keep the external documentation embedded in the source file and to have a utility that extracts it and formats it as HTML. This way, the HTML file would be for output-only and no one would attempt to edit it (because it is overwritten every time the utility is executed). This is the approach adopted by Java.

The API text appears within the class definition as multi-line comments surrounded by `/**` (*two* asterisks instead of one) and `*/`. These comments are placed immediately before every public attribute, constructor, and method in the class [6]. In addition, such comments are needed before the

[6] It is a good habit to also document private features in addition to public ones. Even though such documentation

class header to document the class as a whole; what it encapsulates, its invariants, who implemented it, etc.

As an example, let us document one of the `min` methods of the `Math` class:

```
/**
  Returns the smaller of two int values.

  @param a An argument.
  @param b Another argument.
  @return The smaller of a and b.
*/
public static int min(int a, int b)
{
   ...
}
```

The first sentence gives an overall description of what the method does. When we write external documentation, we have to keep in mind that all text will be converted to HTML. Hence, line breaks, tabs, and other whitespace characters are all treated as a single space. If we feel the need to format the output, use HTML tags, e.g. `<code>`, `<br/>`, `<ul>`, etc. The API extraction utility will place the first sentence of our documentation (up to the first period) in the *Method Summary* section. The *Method Detail* section will contain this first sentence plus any following sentences.

Embedded in the documentation are special tags identified by the `@` prefix. In the above example, the `@param` is used to document every parameter of the method. The `@return` tag describes what the method returns. The generated API is given below.

min

```
public static int min(int a,
                      int b)
```

Returns the smaller of two int values.

Parameters:
`a` - An argument.
`b` - Another argument.
Returns:
The smaller of a and b.

If a method does not take any parameters then no `@param` tag should be used. Similarly, there should not be a `@return` tag if the method is `void`.

Here is an example for an attribute:

```
/**
```

is not visible to the client, it will prove invaluable to the implementer who is maintaining the class.

```
  The double value that is closer than any other to pi, the ratio of the
  circumference of a circle to its diameter.
*/
public static final double PI = 3.141592653589793;
```

No special tags are normally needed for attributes.

Finally, here is an example for the documentation of the class as a whole (to appear at the top of the API):

```
/**
  The class Math contains some methods for performing basic numeric operations
  and some basic constants. The implementations of the methods have been
  simplified and may not handle special cases correctly.

  @author cbc
  @see java.lang.Math
*/
public class Math
```

The `@see` tag allows us to cross reference a second class (or feature) that is related to the current one.

Once a class is documented, we can extract the API by using the following command:

`javadoc -d` *dir* `Math.java`

The `d` switch allows us to specify a directory *dir* in which the generated API should be stored. The `javadoc` utility has many other switches and options. More information about the generation of APIs can be found at `java.sun.com/javadoc`.

1.5.2 Documenting Preconditions and Exceptions

Consider the following method in our `Math` class:

```
/**
  Returns the value of the first argument raised to the second argument.

  @param base The base.
  @param exponent The exponent.
  @return base<sup>exponent</sup>.
*/
public static int pow(int base, int exponent)
```

It is clear from the return type that the class designer is not interested in cases in which `exponent` is negative. Ensuring this can be either left to the client, by making it a precondition, or to the implementer, by addressing it in the postcondition. Either way, the API must reflect and document the choice, and that is what we focus on in this section.

Documenting a Precondition

If the designer opts for a precondition, we add a tag to specify it. Since Javadoc does not currently have such a tag, we add a custom one and name it `pre.` (as a matter of style, we put a period at the end of user-defined tags to distiguish them from built-in ones). The documentation becomes:

```
1 /**
2   Returns the value of the first argument raised to the second argument.
3
4   @param base The base.
5   @param exponent The exponent.
6   @pre. exponent >= 0.
7   @return base<sup>exponent</sup>.
8 */
9 public static int pow(int base, int exponent)
```

In order to recognize the new tag and place it in between the parameters and the return in the generated API, a few switches need to be used in the `javadoc` command:

```
javadoc -tag param -tag pre.:a:"Precondition:  " -tag return -d dir Math.java
```

The resulting API is given below.

pow

```
public static int pow(int base,
                      int exponent)
```

Returns the value of the first argument raised to the power of the second argument.

Parameters:
`base` - The base.
`exponent` - The exponent.
Precondition:
exponent >= 0.
Returns:
$base^{exponent}$.

In order to create links to other classes, we can exploit the `-link` option. For example, to create links to classes in the Java standard library, the option

```
-link http://java.sun.com/javase/6/docs/api/
```

can be used.

Documenting an Exception

If the designer opts for throwing an exception, we used the built-in tag `throws` to document this action. Assuming that `IllegalArgumentException` is to be thrown, the documentation becomes:

```
/**
  Returns the value of the first argument raised to the second argument.

  @param base The base.
  @param exponent The exponent.
  @return base<sup>exponent</sup>.
  @throws IllegalArgumentException if exponent < 0.
*/
public static int pow(int base, int exponent) throws IllegalArgumentException
```

The resulting API is shown below.

pow

```
public static int pow(int base,
                      int exponent)
               throws java.lang.IllegalArgumentException
```

Returns the value of the first argument raised to the second argument.

Parameters:
`base` - The base.
`exponent` - The exponent.
Returns:
$\text{base}^{\text{exponent}}$.
Throws:
`java.lang.IllegalArgumentException` - if exponent < 0.

1.5.3 Check your Understanding

You implemented a utility in Section 1.2 that included the following `factorial` method:

```
public static long factorial(int n)
```

Recall that the return is a `long` (rather than an `int`) because the factorial function grows rapidly and quickly exceeds the `int` range. Enhance your implementation by including the following features:

- Proper treatment of `n` ≥ 0 as a precondition.
- Full `javadoc` documentation of the class.
- Generation of the API.

1.6 Beyond the Basics

1.6.1 Using an Interface to Declare Parameters

Suppose that we are asked to implement a utility that includes the following method:

```
1 public static int frequency(ArrayList<Long> list, long x)
```

The method counts and returns the number of times the long `x` appears in the passed `ArrayList`. The return is 0 if `x` is not present in `list`. How would we implement this method? Think about it and then write the code and test it.

Suppose that we were later asked to add a second method to our utility, one with the following header:

```
1 public static int frequency(LinkedList<Long> list, long x)
```

This method behaves exactly the same as the earlier one except it takes a `LinkedList` rather than an `ArrayList`. We *could* add this method to our utility (as an overloaded version of the earlier one) but that would be a clear violation of a key software engineering principle that calls for avoiding *code duplication*. If we think about the algorithm needed for either method, we will quickly realize that they both use the same logic. Indeed, the needed methods (e.g. `list.size()`, `list.get(int)`, or `list.iterator()`) are all present in the `List` interface. Hence, we could combine the above two methods into just one by declaring the `list` parameter through its interface rather than its class:

```
1 public static int frequency(List<Long> list, long x)
```

Note that even though we focused on interfaces, the observation of this subsection applies as well to abstract and concrete superclasses. For example, before implementing a method, say `setTime`, that takes a parameter of type `Time` that extends `Date`, ask we can ask ourselves "Does the method's logic depends on the features of `Time` or on those of its superclass `Date`?" If it depends on those of `Date` then declare its parameter as a `Date` type. This way, the method can be invoked with an argument of either type and still function correctly.

Generally speaking, using the "highest" possible interface, class, or abstract class to declare parameters, makes our methods more versatile because their parameters would be able to accommodate a larger set of arguments.

1.6.2 Using Generics for Parameters and Returns

We saw that using an interface rather than a class to declare parameters allows us to consolidate

```
1 public static int frequency(ArrayList<Long> list, long x)
```

and

```
1 public static int frequency(LinkedList<Long> list, long x)
```

into one method

```
1 public static int frequency(List<Long> list, long x)
```

In this section we pursue a similar consolidation but in an orthogonal dimension: the type of the elements in the list.

Let us start with a simple case in which we seek to support `int` in addition to `long`. Can a client of the above method invoke it by passing a `List<Integer>` and an `int`? The answer is

no because `List<Integer>` is not a subclass of `List<Long>`. In general, there is no inheritance relation between `List<A>` and `List<B>` even if `A` is a subclass of `B`.[7] Hence, even if we changed the header of our method to

```
1 public static int frequency(List<Object> list, Object x)
```

the client would still not be able to invoke it by passing a `List<Integer>` and an `int` for the same reason.

The solution is to use generics in our implementation. Rather than use `Integer` or `Long` to refer to the type, let us use the type parameter `T`:

```
1 public static <T> int frequency(List<T> list, T x)
```

The token `<T>` before the return type declares that this is a generic method. We implement the body of the method as if `T` were a concrete type. Here is a possible implementation:

```
1  public static <T> int frequency(List<T> list, T x)
2  {
3     int result = 0;
4     for (T e : list)
5     {
6        if (e.equals(x))
7        {
8           result++;
9        }
10    }
11    return result;
12 }
```

Generally speaking, using generics makes the methods more versatile without sacrificing type safety.[8]

1.6.3 Check your Understanding

We like to write the utility `IntegerArrayList` that contains the following method:

```
1 public static double sum(ArrayList<Integer> list)
```

The method returns the sum of the passed collection of integers. Implement the utility, test it, and make sure it behaves as specified.

We now seek to go beyond the basics by generalizing this method in two orthogonal ways. We will do this in stages:

[7] The technical name for this behaviour is that generics are not *covariant*.

[8] We may think that using `<Object>` for parameter declaration can achieve the same versatility as using generics. Doing so, however, sacrifices type safety because it thwarts the type checking done by the compiler and invariably leads to crashes due to illegal type casts.

- Copy `IntegerArrayList.java` to `IntegerCollection.java` and refactor the method in `IntegerCollection` so that its argument can be of type `ArrayList<Integer>`, `LinkedList<Integer>`, `TreeSet<Integer>`, or `HashSet<Integer>`. Compile and test the new utility and make sure it behaves as specified.

- Copy `IntegerArrayList.java` to `NumberArrayList.java` and refactor the method in `NumberArrayList` so that its argument is an `ArrayList` of any numeric type, e.g. `ArrayList<Integer>`, `ArrayList<Long>`, `ArrayList<Double>`, etc. Compile and test the new utility and make sure it behaves as specified. Recall that all numeric types extend `Number`. Note, however, that generics are not covariant, and hence, you cannot simply use `ArrayList<Number>` because even though `Integer` *is a* `Number`, `ArrayList<Integer>` is *not* an `ArrayList<Number>`.

- Create the `NumberCollection` utility so that its `sum` method combines the above two generalizations; i.e. it accepts any collection of any numeric type.

1.6.4 Generics with Wildcards

Given a list `list` of `Date` objects, we seek to write a utility class with a method named `smaller` that determines if all elements of a given list are smaller than a given `Date` object `x`. We are not concerned here with the details of the implementation of the method, just with its signature. Only the fact that the method `smaller` will use the method `compareTo` to compare the elements of `list` with `x` will play a role in our discussion below. One suggestion for the signature of `smaller` would be to use:

```
1 public static boolean smaller(List<Date> list, Date x)
```

This signature, however, restricts the usability of the method to the `Date` type. We like to come up with a signature that makes the method usable for *any* list of comparable objects and *any* object `x` that can be compared with the list elements. For example, the list could be of type `List<Time>` and `x` could be of type `Date` which is the superclass of `Time` and can be compared with it.

Next, we discuss five different proposals for a more versatile signature and briefly critique each proposal by commenting on its type safety and versatility.

```
1 public static boolean smaller(List list, Object x)
```

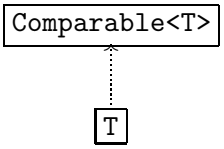
This proposal is clearly not typesafe because it does guarantee that the list elements are comparable with `x`. The implementer will not be able to invoke `compareTo` on `x` without casting it first, and such a cast may lead to a runtime error.

```
1 public static <T> boolean smaller(List<T> list, T x)
```

Also this proposal is not typesafe because it does guarantee that `T` implements the `Comparable` interface. This signature can thus lead to the same problems as the first one.

```
1 public static <T extends Comparable<T>> boolean smaller(List<T> list, T x)
```

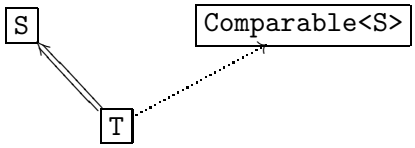
In this case, the type parameter `T` is restricted: only a class like `Date` that implements the interface `Comparable<Date>` can be used. This restriction can roughly be captured by the following UML-like diagram.



Note that the class `Time` that implements `Comparable<Date>` cannot be used. Hence, its applicability is limited to lists of objects that can be compared with an object of the same type.

```
1 public static <T extends Comparable<? super T>> boolean smaller(List<T> list,
    T x)
```

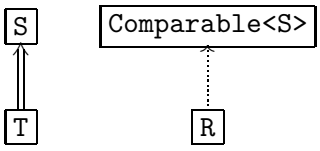
Also in this case, the type parameter `T` is restricted. This case is less restrictive than the previous one. The `?` is called a *wildcard*. The pattern `? super T` is matched by any super class of `T` or `T` itself. In this case, the restriction can roughly be captured by the following UML-like diagram. In the diagram, we use to denote that class `A` extends class `B` or `A` equals `B`. The wildcard is represented by `S` in the diagram.



The class `Date` matches the pattern `<T extends Comparable<? super T>>` since the class `Date` implements the interface `Comparable<Date>` (In this case, both `S` and `T` are matched with `Date`.) Also the class `Time` matches this pattern since the class `Time` implements the interface `Comparable<Date>`. (In this case, `S` and `T` are matched with `Date` and `Time`, respectively.) However, the method cannot be used for a `List<Time>` object and a `Date` object, since `T` would have to match `Date` but `List<Time>` is not compatible with `List<Date>`.

```
1 public static <T> boolean smaller(List<? extends Comparable<? super T>> list,
    T x)
```

To also capture the combination of a `List<Time>` object and a `Date` object we add another wildcard. In this case, the restriction can roughly be captured by the following UML-like diagram. In the diagram, the first wildcard is represented by `R` and the second one by `S`.



If `T` is matched by `Date`, then `List<Time>` matches `List<? extends Comparable<? super T>>` since the `Time` class implements the `Comparable<Date>` interface. (In this case, `S` and `T` are both matched with `Date` and `R` is matched with `Time`.)

1.6.5 Loop Invariants

As we already mentioned earlier, a loop invariant is a boolean expression that holds at the beginning of every iteration of the loop. Every loop has many loop invariants. For example, the boolean expression `true` is a loop invariant for every loop. Recall that the boolean expression `true` always holds and, hence, holds at beginning of every iteration of any loop.

Let us consider the following code snippet, which is very similar to the body of the `pow` method.

```
1 int pow = 1;
2 int i = 0;
3 while (i < b)
4 {
5    pow = pow * a;
6    i++;
7 }
8 return pow;
```

As we already mentioned above, `true` holds at the beginning of every iteration of a loop and, hence, is a loop invariant for the above loop. However, it does not say anything interesting about the loop.

The boolean expression `i >= 0` is also a loop invariant for the above loop. At the beginning of the first iteration, the variable `i` has the value zero and, hence, the boolean expression `i >= 0` holds. Since the variable `i` is incremented in the body of the loop, the loop invariant is maintained by the loop and, therefore, holds the beginning of subsequent iterations as well. This loop invariant is more interesting.

The boolean expression `i <= b` is a loop invariant for the above loop as well. From the precondition `b >= 0` and the fact that the variable `i` is initialized to zero, we can conclude that the boolean expression `i <= b` holds at the beginning of the first iteration of the loop. The body of the loop is only executed if the boolean expression `i < b` holds. In the body, the variable `i` is incremented by one and, hence, the boolean expression `i <= b` holds at the end of each iteration (which is the beginning of the next iteration). This loop invariant can be used to conclude that the boolean expression `i == b` holds at the end of the loop.

As we have already seen earlier, the boolean expression $\texttt{pow} == \texttt{a}^{\texttt{i}}$ is a loop invariant for the above loop as well. We can also combine loop invariants. Since the booleans expressions `i <= b` and $\texttt{pow} == \texttt{a}^{\texttt{i}}$ hold at the beginning of every iteration of the loop, the boolean expression $\texttt{i <= b \&\& pow} == \texttt{a}^{\texttt{i}}$ does as well and, hence, is a loop invariant as well. This loop invariant can be used to prove that $\texttt{pow} == \texttt{a}^{\texttt{b}}$ holds at the end of the above code snippet.

Coming up with a loop invariant that tells us something essential about the loop can be difficult. There are tools that can help us with finding loop invariants. However, these tools only partially solve the problem of finding an appropriate loop invariant. It can be shown that it is impossible to develop a tool that can find an appropriate loop invariant for any loop.

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.

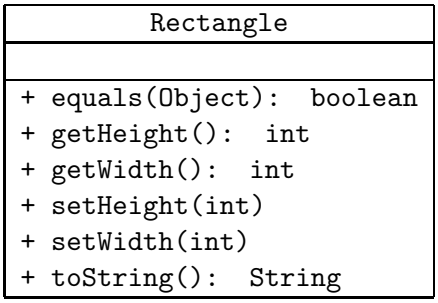
Since a class needs to be instantiated before it can be used, its client must begin by creating an instance of it (i.e. an object). As an example, consider the `Rectangle` class the API of which is shown here. The following fragment shows how a client can use this class. The client starts by creating 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);
```

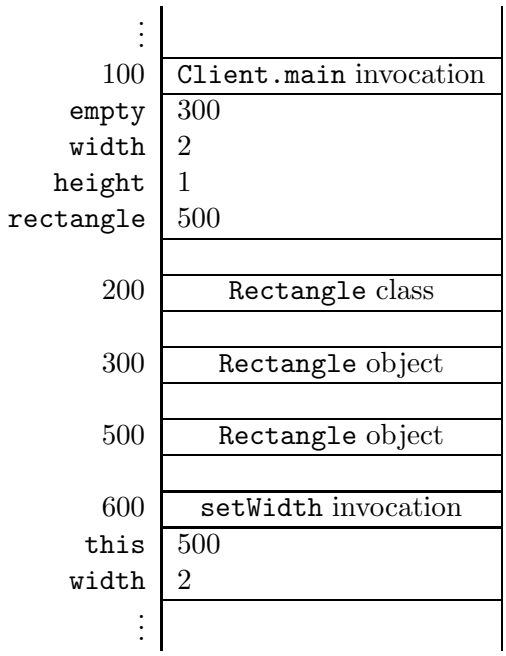
The full example can be found at this link.

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 }
```

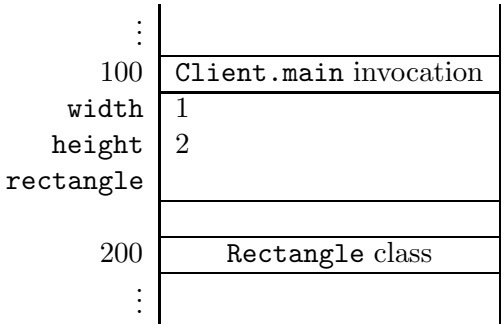
Note that within the body of the constructor, `width` refers to the parameter of the constructor, whereas `this.width` refers to the attribute of the object under construction.

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.

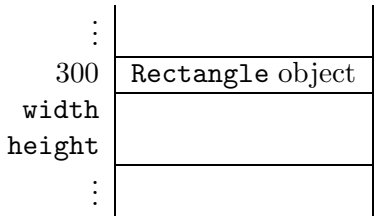
[2] The access modifier in Java can also be `protected`, which makes the constructor visible only to subclasses of this class, or can be left blank, which makes the constructor visible only to classes in the same package as this class.



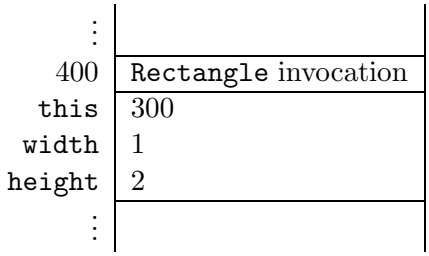
The execution of line 4 can be split into three parts:

- 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;
}
```

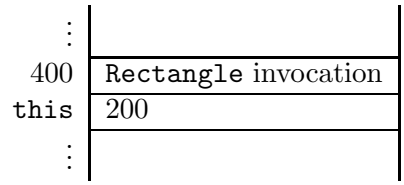
The default constructor can be seen as a special case of the two-parameter constructor. We can avoid some code duplication by rewriting the default constructor so that it merely delegates to the two-parameter constructor. As we have seen in Section 2.2.3, we store three pieces of information in the invocation 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 }
```

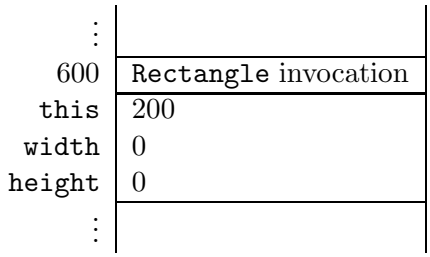
Consider the following fragment of client code.

```
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 }
```

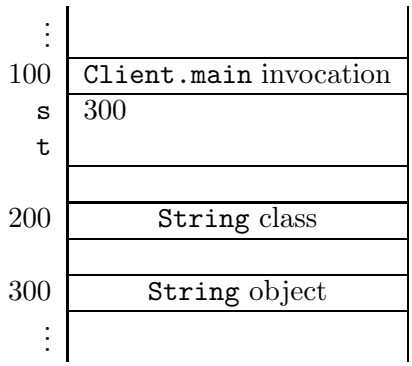
The refactored class can be found by following this link.

2.3.2 Immutable Objects

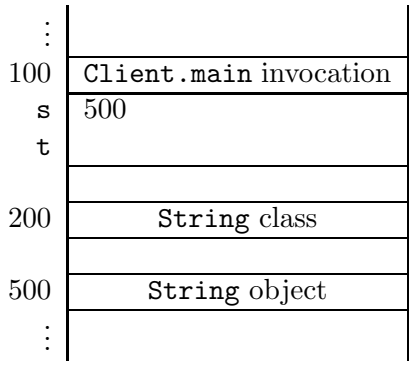
Consider the following snippet of client code.

```
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.



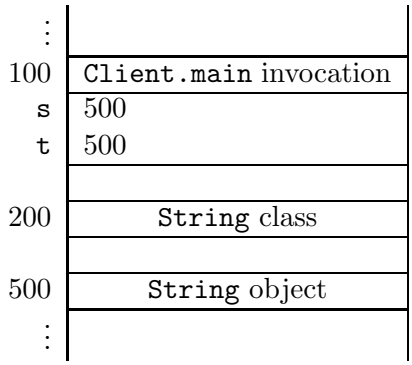
The state of the `String` object at address 300 reflects that this object represents the string `"yo"`. Somewhere in memory the characters `'y'` and `'o'` are stored. In line 2, the length of the string is doubled. However, when we create a `String` object, we cannot always predict its maximal length (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 $1 \times 2^{32} + 2$.

Chapter 3

Mixing Static and Non-Static Features

3.1 Introduction

In Chapter 1, we focused on static features. Non-static features were the topic of Chapter 2. In this chapter, we mix the two.

In order to focus on the general ideas without being distracted by the details of a particular class, we will illustrate most using a very simple class, the `Rectangle` class that we implemented in the previous chapter.

3.2 Incorporating an Invocation Counter

There are a variety of situations in which we may want to keep track of the number of times a particular method in a class is invoked. Such an invocation count provides a measure of how "popular" a method is, and may for example be used to determine if the efficiency of the method needs to be improved. The count can also be used in settings in which a cost (in dollars, time, or system resources) is associated with using the method and we need to profile the utilization of this cost.

As an example, suppose we were told to keep track of the number of times the `getArea` method of the `Rectangle` class is used and to report this count via a new method named `getCount`. How can we refactor the `Rectangle` class to incorporate this change? As an implementer, we first have to decide how to represent this additional information. Introducing an attribute, named `count`, of type `int` is a natural choice. If we expect an enormous number of invocations, we may choose an attribute of type `long` instead. As always, it should be a `private` attribute.

Should we make the attribute static or non-static? If we make it non-static, then it will be associated with instances of the class. Hence, we will have a separate counter for each instance of the class. This counter will record the number of invocations made on that particular instance. If, however, we make it static, then it will be associated with the class. Hence, there will be a single counter. This counter will keep track of the number of invocations made on all instances of the

class. In other words, the static count would be the sum of all the non-static counts. What we are after is an overall count so we make the attribute static.

We add the following declaration to the attribute section.

```
1 private static int count = 0;
```

Since the attribute is static, in accordance with our attribute initialization guidelines, we initialize the attribute together with its declaration, *not* in the constructor. This implies that no changes need to be made to the constructor section.

We need to increment this counter whenever `getArea` is invoked so we add the following line to its body.

```
1 Rectangle.count++;
```

Note how the attribute was referenced: since it is static we prefix it with the class name.

Finally, we add a new method that acts as an accessor for the new attribute.

```
1 public static int getCount()
2 {
3    return Rectangle.count;
4 }
```

Note that this method is static as it only involves static attributes.

3.3 Stamping a Serial Number on Objects

Assume that we want to associate unique serial number with every instance of a class. This mimics the situation in many factories where each product has a serial number. This pattern can be thought of as made up of two parts, one of which is similar to the invocation count that we discussed in the previous section, and the other is new.

The similarity with the invocation count becomes clear if we consider counting the number of times a constructor is called. In a multi-constructor class, it is common to have all constructors chained so they all ultimately call one of them. If we apply the invocation count idea to that constructor, we will be able to generate a serial number for each instance.

The second part of this pattern requires that we stamp the generated serial number on the instance, i.e. store it in the instance, and this can be done by adding a non-static attribute to hold it.

We therefore conclude that implementing this pattern calls for *two* attributes, one static to generate numbers serially and one non-static to hold the serial number per instance.

Let us apply this pattern by requiring that every rectangle we create from our `Rectangle` class carries a unique serial number that starts at 1 and increments serially. Based on the above discussion, this means we need to add the following declarations to the attribute section.

```
1 private static int count = 0;
2 private int number;
```

For the attribute `number`, we introduce an accessor and mutator.

Next, we need to increment `count`. Recall that after re-factoring the `Rectangle` class to avoid code duplication, we chained its three constructors so that two of them delegate to the two-parameter constructor which, in turn, delegates to the mutators. It is in this two-parameter constructor where we should increment the `count`.

Furthermore, since the attribute `number` was added to the state, it needs to be initialized and we do that also in the two-parameter constructor. Hence, we add two lines to the body of this constructor, as follows.

```
1 public Rectangle(int width, int height)
2 {
3    this.setWidth(width);
4    this.setHeight(height);
5    Rectangle.count++;
6    this.setNumber(Rectangle.count);
7 }
```

The serial number is available to the client via the accessor. Note, however, that we probably do not want to allow the client to change this number. This can be accomplished by making the mutator private.

3.4 Maintaining a Singleton

Imagine having a large, multi-class application in which several classes require the services of one particular class that represents a resource, e.g. a database connection. If every class were allowed to instantiate this resource class then we would end up with several instances of it and this could be wasteful (e.g. too many connections to the same database). What we need is the ability to ensure that only a single instance of a class can be created (per virtual machine) and to provide global access to this instance. The lone instance in this scenario is known as a *singleton* and in this section we discuss how to implement the so-called *singleton design pattern*.

First of all, how do we prevent clients of our class from instantiating it more than once? Anytime a client uses the `new` operator in conjunction with the class name, a new instance is created. Hence, it is clear from the outset that the client should not be able to use `new` and this, in turn, implies that our class must not have any public constructor. We therefore make all the class constructors private.

Now that we have prevented the client from instantiating our class, we need to find a way to deliver the lone instance to the client, and we do that through the return of a method. The method has got to be `static` or else the client cannot invoke it! Hence, this line of reasoning leads us to create a method with a header similar to the following.

```
1 public static ClassName getInstance()
```

How do we store the single instance so that the method can return it? We need to have an attribute that stores this instance thereby persisting it at the class level. This implies the attribute must be static.

```
private static ClassName instance = new ClassName(...);
```

Note that, as always for static attributes, we combine the declaration and the initialization. To initialize the instance, we invoke one of the private constructors. In the body of the `getInstance` method, we simply return the attribute as follows.

```
public static ClassName getInstance()
{
   return ClassName.instance;
}
```

Let us apply this by requiring that our `Rectangle` class be a singleton. We start with making all constructors of the `Rectangle` class private. Based on the above discussion, we need to introduce a static attribute to hold the singleton. We therefore add the following to the attribute section.

```
private static Rectangle instance = new Rectangle();
```

Furthermore, we add the following method to our `Rectangle` class.

```
public static Rectangle getInstance()
{
   return Rectangle.instance;
}
```

Note that the singleton returned by the `getInstance` method represents an empty rectangle, i.e. a rectangle with width zero and height zero. This is not a limitation because the client can use the public mutators to mutate this instance to any desired width or height. The key point is that, regardless of width and height, there is at most one `Rectangle` object in memory at all times.

The `Rectangle` object is created when the `Rectangle` class is loaded into memory. If the invocation of the constructor is expensive (in terms of time or memory), we may want to delay the creation of the `Rectangle` object until the first invocation of the `getInstance` method. In that case, we initialize the attribute `instance` to `null`. We will create the singleton upon receiving the very first request from the client. The `null` value acts as a flag to enable us to determine if the singleton has or has not been created yet.

```
private static Rectangle instance = null;
```

In this case, the `getInstance` method must determine whether an instance has been created yet. If it has, then a reference to it should be returned; otherwise, it should be created by means of the private constructor, stored, and then returned.

```
public static Rectangle getInstance()
{
   if (Rectangle.instance == null)
   {
      Rectangle.instance = new Rectangle();
   }
```

```
7     return Rectangle.instance;
8 }
```

3.5 Enforcing One Instance Per State

Rather than allowing just a single instance of the class, as we did in the previous section, in this section we want to ensure that there is only one instance of our class for every possible combination attribute values. In other words, we need to create a "singleton for every state."

Consider the following code fragment in a client app and try to predict its output.

```
1 String s1 = "York";
2 String s2 = "York";
3 output.println(s1.equals(s2) + " - " + (s1 == s2));
```

The output is `true - true`. While it is not surprising that `s1.equals(s2)` returns `true` (after all, the two strings are made up of the same character sequence), it may be unexpected to see that `s1 == s2` returns `true` as well. After all, the `String` objects `s1` and `s2` may be thought to be different objects residing at different addresses in memory. In an effort to save memory, the Java compiler creates only one instance for each string literal such as `"York"` and makes all references point to it. The compiler is therefore enforcing a single instance per state, where state in this case refers to the characters that make up the string.[1]

How can we enforce a single instance per state? In order to prevent the client from controlling instantiation, we must make all constructors private, as was done for the singleton. Moreover, we better make all mutators private (and, hence, the class becomes immutable as we have seen in Section 2.3.2). Otherwise, the client can change the state and thus create two instances having the same state. And for the singleton pattern, we should provide a static method named, for example, `getInstance`. This time, the client should be able to pass arguments to it to specify the desired state for the requested instance. The body of this method must determine whether an instance of the desired state has been already created. If it has, then return it; otherwise, create the instance using a private constructor, store it, and then return it. The attribute to store the instances has to be static and has to be able to hold many instances, corresponding to many states. It must therefore be a collection. One can, for example, use a `Map` whose key represents the state and whose value represents the lone instance that has that state. The facts that mutators must be private and instance storage must be in a collection make this scenario slightly different from the singleton design pattern.

Let us apply this by requiring that our `Rectangle` class allows only one instance for a given width and height. Based on the above discussion, we need to add a static `Map` attribute to hold the created instances. We add the following declaration to the attribute section:

```
1 private static Map<String, Rectangle> instances = new TreeMap<String,
      Rectangle>();
```

[1] It should be noted that the compiler does this for string literals only. If the character sequence is constructed using concatenation or any other mechanism, different instances may be created even if they share the same character sequence.

Note that we initialized the attribute to an empty map. Next, we make the three constructors and the two mutators of the class private. Finally, we add the following method.

```
1  public static Rectangle getInstance(int width, int height)
2  {
3     String key = width + "-" + height;
4     Rectangle instance = Rectangle.instances.get(key);
5     if (instance == null)
6     {
7        instance = new Rectangle(width, height);
8        Rectangle.instances.put(key, instance);
9     }
10    return instance;
11 }
```

Note that in order to facilitate storage in the map, we need to come up with a unique key that identifies the state. In the above implementation we simply concatenated the width and the height after delimiting them by a hyphen. Alternatives will be discussed in the next section.

3.6 Check you Understanding

Refactor the `Rectangle` class developed in previous section in the following ways.

- Instead of using a `String` as key, use the class `java.awt.Point` for the keys of the `Map`.
- Instead of using a `String` as key, develop a class `Pair` with two attributes of type `int` and use this class for the keys of the `Map`.
- Make the class `Pair` generic and use this class for the keys of the `Map`.
- Instead of using a `Map`, use a `List` to store the instances.
- Instead of using a `Map`, use a `Set` to store the instances.

Chapter 4

Implementing Aggregation and Composition

4.1 Implementing Aggregation

4.1.1 What is Aggregation?

Aggregation is a relation. It is also known as the *has-a* relation. It is a relation on classes. The relation captures that one class is part of another class. For example, assume that the class `Student` represents a student. For each student we want to keep track of the student's homepage. We decide to represent the student's homepage as an instance of the class `URL`, which is part of the package `java.net`. In this case, the classes `Student` and `URL` are related by the aggregation relation. We say that `Student` has a `URL`, and `URL` is called a component of `Student`.

The API of the `Student` class can be found at this link. The header of the three-parameter constructor

```
public Student(String id, String name, URL homepage)
```

reflects that the client has to create two instances of the `String` class and one instance of the `URL` class in order to create an instance of the `Student` class. In the next code fragment, the client creates an instance of the `URL` class, creates an instance of the `Student` class from two `String` literals and the instance of the `URL` class, and finally prints the `Student` object.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Student student = new Student("123456789", "Jane Smith", homepage);
3 output.println(student);
```

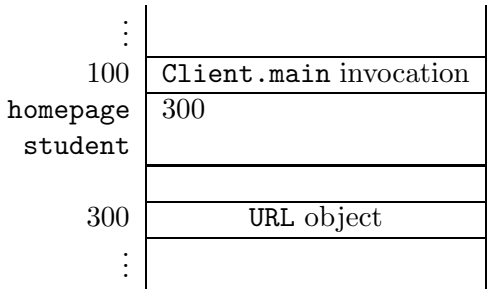
The complete client code can be found at this link.

4.1.2 UML and Memory Diagrams

The aggregation relation can be reflected in UML class diagrams. For example, the fact that `Student` has a `URL` can be depicted as follows.

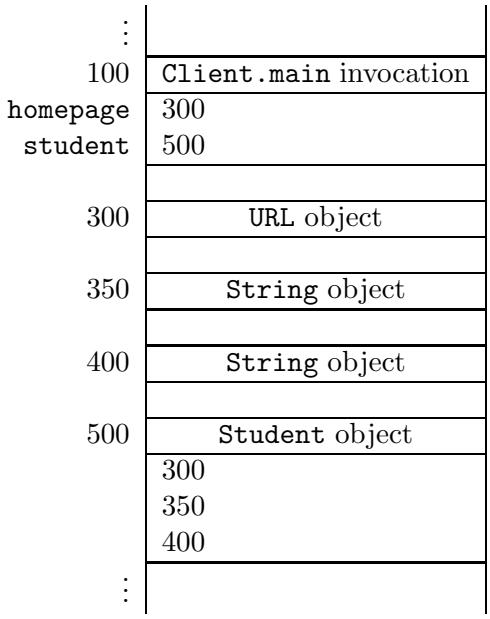
The number 1 in the above class diagram is called the multiplicity of the aggregation. It represents the number of `URL` objects that each `Student` object has: each `Student` has one `URL`.

Memory diagrams can also be utilized to reason about aggregation. For example, consider the code fragment introduced in Section 4.1.1. Once the execution reaches the end of line 1, memory can be depicted as follows.



The above diagram reflects that a `URL` object has been created at address 300 and that the local variable `homepage` refers to that object.

Once the execution reaches the end of line 2, the memory content has changed and can now be depicted as follows.



The diagram now also contains a `Student` object at address 500 and the local variable `student` refers to the `Student` object. Note also that the `Student` object refers to the two `String` objects and the `URL` object as it contains the addresses of these objects. We have not included yet the names of the attributes which refer to these objects. These attributes will be discussed in the next section.

4.1.3 The Attributes Section

As an implementer, we have to decide on the number of attributes and for each attribute we have to choose its type and its name. Usually, we look at the constructors and also the accessors and mutators in the API for inspiration. In the API of the `Student` class we find a three-parameter constructor with the following header.

```
public Student(String id, String name, URL homepage)
```

In the API, we also encounter the accessors `getId`, `getName` and `getHomepage` and the mutator `setHomepage`. These suggest the following three attributes.

```
1 private String id;
2 private String name;
3 private URL homepage;
```

Note, however, that choosing the attributes is in general not as simple as picking the parameters of the constructor with the most parameters. Although the above attributes are a natural choice, other choices are possible. We will come back to this in Section 4.1.6.

We will implement the constructors and methods in such a way that the following class invariant is maintained.

```
1 /* class invariant: this.id is a 9-digit string and this.name != null */
```

As we will see, maintaining this class invariant will allow us to simplify our code.

4.1.4 The Constructors Section

In the constructors, we initialize the attributes `id`, `name` and `homepage`. According to the API, the `Student` class has three constructors. One of them takes two arguments (`id` and `name`), another has three parameters (`id`, `name` and `homepage`), and the third one is a copy constructor (and, hence, takes a single argument of type `Student`). The two-parameter constructor can be used if the student does not have a homepage. To avoid code duplication, we will utilize constructor chaining as discussed in Section 2.3.1. In this case, the two-parameter constructor and the copy constructor both delegate to the three-parameter constructor. To reflect the fact that a student does not have a homepage, we initialize the attribute `homepage` to `null`. Hence, the body of the two-parameter constructor may be implemented as follows.

```
1 this(id, name, null);
```

The body of the copy constructor can be implemented as follows.

```
1 this(student.getId(), student.getName(), student.getHomepage());
```

To avoid code duplication, we use the accessors. These accessors will be discussed in the next section.

In the three-parameter constructor, we utilize the mutators to initialize the attributes. As we have seen in Section 2.3.1, this avoids code duplication. Hence, the body of the three-parameter constructor may be implemented as follows.

```
this.setId(id);
this.setName(name);
this.setHomepage(homepage);
```

According to the API, the three-parameter and the two-parameter constructor throw an `IllegalArgumentExcep` if the ID provided by the client is not valid. The parameter `id`, which is of type `String`, is valid if

- it is not `null`,
- it consists of nine digits.

Rather than checking it in the constructor, we have decided to delegate the validation to the mutator `setId`.

The fact that those constructors throw an `IllegalArgumentException` if the ID provided by the client is not valid is reflected in the header of the constructor (and, of course, also in the documentation comments). For example, the header of the two-parameter constructor is as follows.

```
public Student(String id, String name) throws IllegalArgumentException
```

One may wonder why the copy constructor does not throw an `IllegalArgumentException`.[1] Intuitively, since the argument, a `Student` object, was created successfully, its ID is valid. Formally, we can infer it from the class invariant.

We leave it to the reader to check that the class invariant holds at the end of each constructor invocation.

4.1.5 The Methods Section

Next, we discuss the (public and private) methods of the `Student` class.

Accessors

For each attribute, we introduce a corresponding accessor. Since all follow the pattern that we introduced in Section 2.2.4, we only present the accessor for the attribute `homepage`.

```
public URL getHomepage()
{
   return this.homepage;
}
```

Note that the return type of the above accessor is non-primitive, i.e. it returns an object of type `URL`.

[1] Since `IllegalArgumentException` is a runtime exception, we do not need to catch or specify it.

Mutators

Since we do not want to allow clients to change the name or the ID of a student, we simply make the mutators `setName` and `setId` private. As a consequence, these mutators do not appear in the API and the attributes `name` and `id` are immutable (by the client).

The mutator of the `name` attribute follows the pattern presented in Section 2.2.4.

```
1 private void setName(String name)
2 {
3    this.name = name;
4 }
```

Also the mutator for the `homepage` attribute follows the same pattern, but this mutator is public.

```
1 public void setHomepage(URL homepage)
2 {
3    this.homepage = homepage;
4 }
```

Recall that the mutator of the attribute `id` not only sets the value of the attribute, but also validates the passed argument. If the passed ID is invalid, the mutator throws an `IllegalArgumentException`. For an ID to be valid, it has to be different from `null` and consist of nine digits. To check the latter property, we can use the regular expression $\backslash d\{9\}$, which is captured by the string literal `"\\{d9}"`.[2] A string matches this pattern if and only if it consists of nine digits. Hence, we can implement the mutator `setId` as follows.

```
1  private void setId(String id) throws IllegalArgumentException
2  {
3     final String PATTERN = "\\d{9}";
4     if (id != null && id.matches(PATTERN))
5     {
6        this.id = id;
7     }
8     else
9     {
10       throw new IllegalArgumentException("Invalid ID");
11    }
12 }
```

The `toString` Method

As we already mentioned in Section 2.2.4, most classes override the `equals` method of the `Object` class. Also our `Student` class overrides the `equals` method. The method returns a string representa-

[2] In string literals, the backslash character ('\') serves to introduce so-called escaped constructs. For example, the expression \t represents a tab. The expression \\ represents a single backslash.

tion of the `Student` object on which it is invoked. According to the API, this string representation, when printed on the screen, will result in one or two lines. The first line contains the student's ID and name, separated by a colon. The second line is only present if the student has a homepage. In that case, the second line consists of (a string representation of) the URL of the student's homepage. For example, for the `Student` object created in the code snippet in Section 4.1.1, the invocation of the `toString` method on this object results in the following.

```
123456789:Jane Smith
www.cse.yorku.ca/~cse81234/
```

The `toString` method can be implemented as follows.

```
1  public String toString()
2  {
3     StringBuffer result = new StringBuffer();
4     result.append(this.getId());
5     result.append(":");
6     result.append(this.getName());
7     if (this.getHomepage() != null)
8     {
9        result.append("\n");
10       result.append(this.getHomepage().toString());
11    }
12    return result.toString();
13 }
```

Rather than constructing several `String` objects,[3] we decided to use a `StringBuffer` object instead. Note that we use two invocations of the `toString` method in the body of the `toString` method. In line 10, we invoke the `toString` method on the `URL` object to obtain a string representation of the student's homepage. In line 12, we invoke the `toString` method on the `StringBuffer` object. Note that we use the accessors, rather than accessing the attributes directly. This avoids code duplication and makes it easier to change the representation of the attributes if necessary.

The `equals` Method

Most classes also override the `equals` method of the `Object` class. According to the API of the `Student` class, two `Student` objects are considered equal if their IDs are equal. The `equals` method has the same general structure as the one in Section 2.2.4. Instead of comparing the width and the height of a rectangle, both being represented by attributes the type of which is primitive, we compare the IDs of the students, represented by an attribute of the non-primitive type `String`. As a consequence, we use `equals`, rather than `==`, to compare the IDs of the students.[4]

[3] Recall that the class `String` is immutable and, hence, concatenation does not change the object but returns a new object.

[4] Although the class `String` is immutable, it does not implement the "one instance per state" pattern and, hence, we should not use `==` to compare `String` objects. For example, the snippet

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

From the class invariant we can conclude that the invocation `this.getId()` in line 7 does not return `null`. Hence, the invocation of `equals` in line 7 never throws a `NullPointerException`.

The `hashCode` Method

Since we did override the `equals` method, we had better override the `hashCode` method as well to satisfy the property

- if `x.equals(y)` returns true then `x.hashCode()` and `y.hashCode()` return the same integer.

Since students are considered equal if their IDs are the same, we can let the `hashCode` method return the ID (as an integer). Note that, according to the class invariant, the ID is a 9-digit string and, therefore, can be represented as an `int`. Hence, the `hashCode` method can be implemented as follows.

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

The `compareTo` method

According to the API of the `Student` class, the class implements the `Comparable<Student>` interface. As a consequence, we have to implement the single method of that interface: the `compareTo`

```
String first = new String("123456789");
String second = new String("123456789");
```

creates two different `String` objects (and, hence, `first == second` returns false) which represent the same ID (`first.equals(second)` return true). Note that the class `Class` does implement the "one instance per state" pattern and, hence, we can use `==` to compare `Class` objects.

method. As we have already seen in Section 2.2.4, this method provides an ordering on objects which can be used, for example, for sorting. The ordering of students is based on an ordering of their IDs. The IDs are ordered lexicographically.[5] Hence, to compare two students we can simply get their IDs and delegate the comparison to the `String` class.

```
1 public int compareTo(Student student)
2 {
3    return this.getId().compareTo(student.getId());
4 }
```

Note that also in this case we use the accessors to get hold of the attributes.

Again we can infer from the class invariant that no `NullPointerException` is thrown in the body of the method.

The code of the entire `Student` class can be found by following this link.

4.1.6 Beyond the Basics

The Attributes Revisited

One may wonder whether we need an attribute of type `URL` in the class `Student`. Although it is a natural choice to represent the student's homepage by means of the an instance of the `URL` class, this information can be represented in several other ways as well. For example, we can represent it as a `String` object. To refactor our `Student` class such that the student's homepage is represented by an instance of the `String` class, we make the following three changes. First of all, we replace the declaration of the `homepage` attribute with

```
1 private String homepage;
```

We add to the class invariant[6]

```
1 this.homepage is a well-formed URL
```

Secondly, we modify the accessor of the attribute as follows.

```
1 public URL getHomepage()
2 {
```

[5] According the API of the `String` class, if two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value (recall that characters can be compared using the `<` relational operator) lexicographically precedes the other string. In this case, the `compareTo` method of the `String` class returns the difference of the two character values at position k in the two string, i.e.

```
this.charAt(k) - other.charAt(k)
```

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, the `compareTo` method of the `String` class returns the difference of the lengths of the strings, i.e.

```
this.length() - other.length()
```

[6] The string is well-formed if it specifies a legal protocol. Examples of legal protocols include http and ftp.

```
3     URL homepage;
4     if (this.homepage == null)
5     {
6        homepage = null;
7     }
8     else
9     {
10       try
11       {
12          homepage = new URL(this.homepage);
13       }
14       catch (MalformedURLException e)
15       {
16          homepage = null;
17       }
18    }
19    return homepage;
```

The API of the `URL` class tells us that the constructor `URL(String)` throws a `MalformedURLException` if the given string specifies an unknown protocol. Since a `MalformedURLException` is a checked exception, the exception has to be either specified or caught. Because the accessor `getHomepage` does not specify the exception in the API, we have to catch the exception. Note, however, that we can infer from the class invariant that `new URL(this.homepage)` never throws a `MalformedURLException`.[7]

Finally, we change the method `setHomepage` to the following mutator.

```
1  public void setHomepage(URL homepage)
2  {
3     if (homepage == null)
4     {
5        this.homepage = null;
6     }
7     else
8     {
9        this.homepage = homepage.toString();
10    }
11 }
```

Note that the new accessor and the mutator have to handle the case that the student has no homepage as a special case. This complicates the code. Since we generally strive to keep our code as simple as possible, we prefer to represent the homepage as a `URL` object, rather than a `String` object.

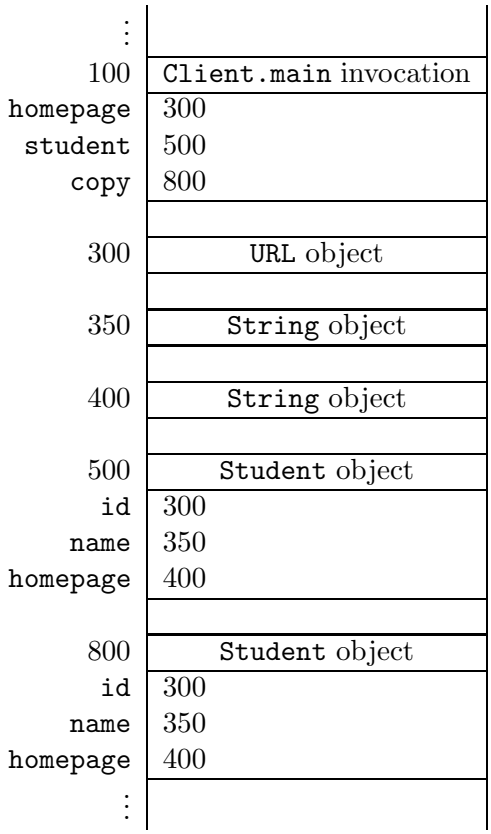
[7] The compiler, however, does not reach this conclusion since it is not aware of the class invariant.

The Copy Constructor Revisited

Consider the following fragment of client code that uses the copy constructor.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Student student = new Student("123456789", "Jane Smith", url);
3 Student copy = new Student(student);
```

Once we reach the end of line 3, memory can be depicted as follows.



Note that both `Student` objects refer to the same `String` and `URL` objects. If we were to change the state of either one of the `String` objects or the `URL` object, both `Student` object would change. However, the `String` class is immutable and, hence, we cannot change the state of `String` objects. Since the API of the `URL` class contains no public mutators, we cannot change the state of `URL` objects either.

4.2 Implementing Composition

4.2.1 What is Composition?

Composition is a special type of aggregation. Hence, it is also a relation on classes. The composition relation not only captures that one class is part of another class. An instance of the latter class

can be said to "own" an instance of the former class. As an example, we will modify our `Student` class. For each student, we also keep track of the student's date of joining the university. Let us represent this date by an instance of the `Date` class, which is part of the `java.util` package.

The API of this modification of the `Student` class can be found at this link. To create an instance of the `Student` class, the client also has to provide an instance of the `Date` class. In the next code snippet, the client creates an instance of the `URL` class, an instance of the `Date` class, and an instance of the `Student` class.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Date now = new Date();
3 Student student = new Student("123456789", "Jane Smith", homepage, now);
```

A key difference between the class `Date` and the classes `URL` and `String` is that the class `Date` is mutable, whereas the other classes are not. For example, the client can get the student's date of joining the university and add one hour to it as follows.

```
4 Date date = student.getJoinDate();
5 final long HOUR_IN_MILLISECONDS = 60 * 60 * 1000;
6 date.setTime(date.getTime() + HOUR_IN_MILLISECONDS);
```

The method `getTime` returns the number of milliseconds from January 1, 1970, 00:00:00 GMT until the date represented by the `Date` object on which it is invoked.

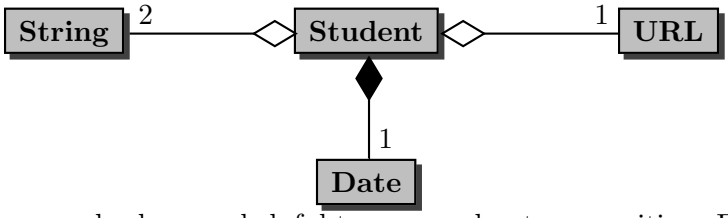
Recall that the classes `Student` and `Date` form a composition, i.e. a `Student` object "owns" its `Date` object. This means that the `Student` object has exclusive access to its `Date` object. Others, including the client, cannot modify its `Date` object.

The accessor `getJoinDate` cannot return a reference to the `Date` object of the `Student` object. Towards a contradiction, assume that the accessor `getJoinDate` does return a reference to the `Date` object of the `Student` object. In that case, the above code snippet mutates the `Date` object "owned" by the `Student`. But this contradicts that `Student` and `Date` form a composition.

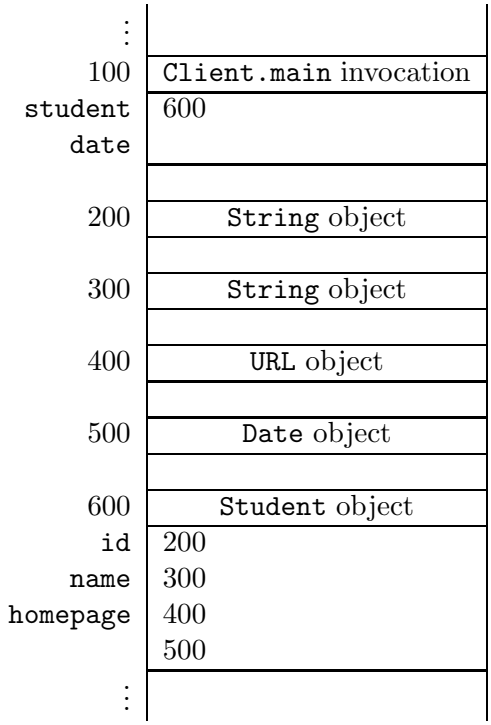
Rather than returning a reference to the `Date` object of the `Student` object, the mutator `getJoinDate` returns the reference to a copy of that `Date` object. Hence, the above code snippet mutates the copy of the `Date` object, not the `Date` object "owned" by the `Student` object.

4.2.2 UML and Memory Diagrams

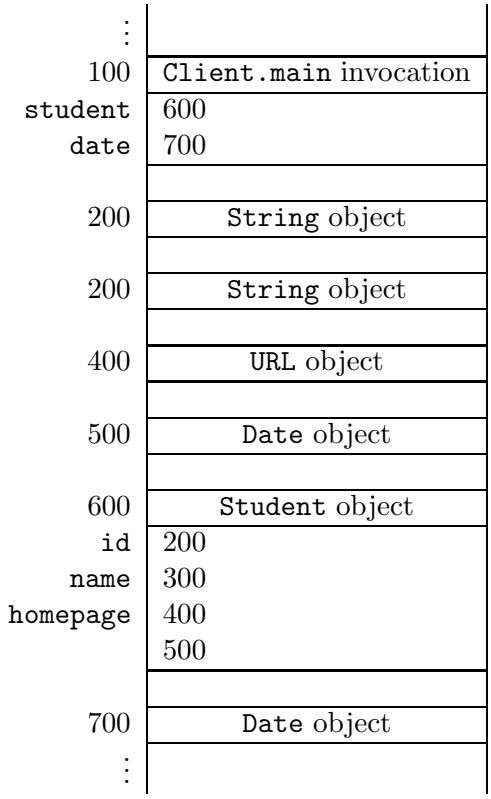
Also the composition relation can be reflected in UML class diagrams. For example, the fact that the classes `Student` and `Date` form a composition can be depicted as follows.



Memory diagrams can also be very helpful to reason about composition. For example, consider the code fragment introduced in Section 4.2.1. Once the execution reaches the end of line 3, memory can be depicted as follows.



Once we reach the end of line 4, the above memory diagram has changed into the one below.



Note that the local variable `date` refers to the new instance of the `Date` class at address 700, and *not* to the instance of the `Date` class at address 500. The latter `Date` object is "owned" by the `Student`. As a consequence, in line 6 the `Date` object "owned" by the `Student` object is *not* modified. Instead, its copy located at address 700 is changed.

4.2.3 The Attributes Section

A natural choice to represent the additional information, the date at which the student joined the university, is an attribute of type `Date`. Hence, we add the following declaration to our `Student` class.

```
1 private Date joinDate;
```

We strengthen the class invariant by adding `this.joinDate != null`.

4.2.4 The Constructors Section

According to the API, the class `Student` contains five constructors. As in Section 4.1.4, we exploit constructor chaining to avoid code duplication. Here, we focus on the constructor to which all other constructors delegate: the four-parameter constructor with the following header.

```
public Student(String id, String name, URL homepage, Date joinDate)
```

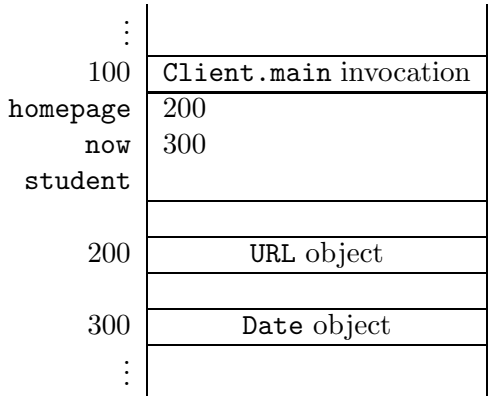
We focus on the fourth parameter. Before implementing the constructor, we first consider the following fragment of client code.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Date now = new Date();
3 Student student = new Student("123456789", "Jane Smith", homepage, now);
4 final long HOUR_IN_MILLISECONDS = 60 * 60 * 1000;
5 now.setTime(now.getTime() + HOUR_IN_MILLISECONDS);
```

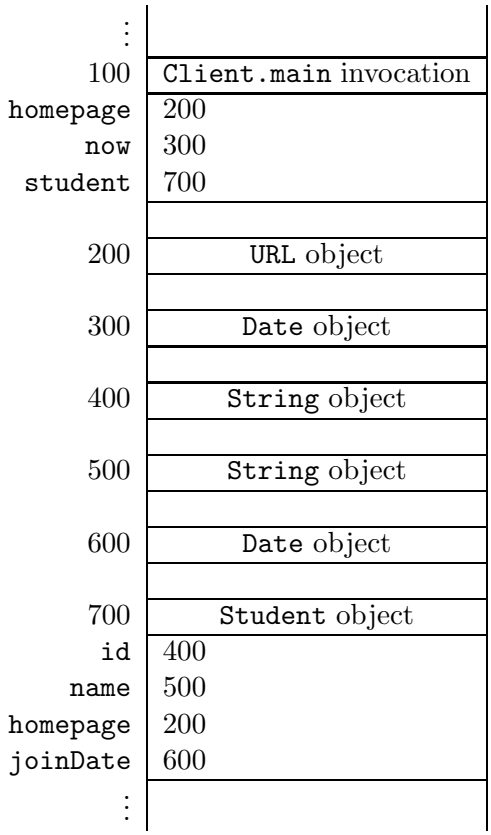
The above example shows that we cannot initialize the attribute `joinDate` to (the address of) the `Date` object (referred to by) `now`. Towards a contradiction, assume that the attribute `joinDate` *is* initialized to the `Date` object `now`. In that case, the client has access to the `Date` object "owned" by the `Student` object by means of its local variable `now`. Hence, the `Student` object does *not* have exclusive access to its `Date` object, which contradicts that the classes `Student` and `Date` form a composition.

Instead of initializing the attribute `joinDate` to the `Date` object passed as an argument, we initialize the attribute to a copy of the passed `Date` object. This copy is the `Date` object which is "owned" by the `Student` object. Note that the client does not have access to this copy.

Consider the above snippet of client code. Assume that the memory can be depicted as follows once we reach the end of line 2.



When we reach the end of line 3, memory can be depicted as follows.



Note that the `Student` object refers to a `Date` different from the one known to the client. Hence, in line 5 of the above code snippet the `Date` object at address 300 is modified, *not* the `Date` object at address 600 which is "owned" by the `Student` object.

Let us return to our attempt to implement the four-parameter constructor. From the above discussion we can conclude that we have to initialize the `joinDate` attribute to a copy of the passed `Date` object. Rather than doing the copying in the constructor, we delegate this to the mutator of the `joinDate` attribute. Hence, the four-parameter constructor can be implemented as follows.

```
1 public Student(String id, String name, URL homepage, Date joinDate) throws
      IllegalArgumentException
2 {
3    this.setId(id);
4    this.setName(name);
5    this.setHomepage(homepage);
6    this.setJoinDate(joinDate);
7 }
```

4.2.5 The Methods Section

Next, we only discuss the methods of the `Student` class which are either new or modified.

Accessor

As we already discussed in Section 4.2.1, the accessor of the `joinDate` attribute has to return a copy of the `Date` object "owned" by the `Student` object. Unfortunately, the `Date` class does not have a copy constructor. However, it has a constructor that is very similar to a copy constructor. The constructor

```
Date(long time)
```

creates a `Date` object which represents the date which is the given number of milliseconds (captured by the parameter `time`) after "the epoch", namely January 1, 1970, 00:00:00 GMT. Hence, using this constructor in combination with the method `getTime`, which we already saw in Section 4.2.1, we can implement the accessor for the `joinDate` attribute as follows.

```
1 public Date getJoinDate()
2 {
3    return new Date(this.joinDate.getTime());
4 }
```

Mutator

Since the API of the `Student` class does not contain a mutator for the `joinDate` attribute, we make the mutator private. Recall that the four-parameter constructor delegates the copying of the argument of type `Date` to the mutator. We use the same technique as used in the accessor to copy the `Date` object.

```
1 private void setJoinDate(Date joinDate)
2 {
3    this.joinDate = new Date(joinDate.getTime());
4 }
```

The `toString` Method

Since the class contains some additional information, namely the date the student joined the university, we want to reflect this additional information in the string representation returned by the `toString` method. In particular, we add one line to the string representation. This line contains the date at which the student joined the university. This date is represented as

- the month, formatted as two digits with leading zeros as necessary, and
- the year, formatted as at least four digits with leading zeros as necessary,

separated by a '/'. To format the date properly, we exploit the `format` method of the `String` class as follows. The specifiers `%1$tm` and `%1$tY` represent the month and year, respectively.

```
1  public String toString()
2  {
3     StringBuffer result = new StringBuffer();
4     result.append(this.getId());
5     result.append(":");
6     result.append(this.getName());
7     if (this.getHomepage() != null)
8     {
9        result.append("\n");
10       result.append(this.getHomepage().toString());
11    }
12    result.append("\n");
13    result.append(String.format("%1$tm/%1$tY", this.getJoinDate()));
14    return result.toString();
15 }
```

The code of the entire `Student` class can be found by following this link.

4.3 Implementing Aggregation Using Collections

4.3.1 What is a Collection?

Assume we want to modify our `Student` class and keep track of two additional pieces of information. First of all, we want to record the student's gpa for each year of study (i.e., for first year, second year, third year and fourth year). Secondly, we want to keep track of the courses the student has taken. The API of the modified `Student` class can be found at this link.[8]

If a course is represented by an instance of the `Course` class,[9] then a `Student` may have zero, one of more `Courses`. The `Student` class has a collection of `Courses`. A collection is a special type of aggregation where the number of components may vary.

[8] To simplify the `Student` class a little, we do not keep track of the name, URL and join date.

[9] The API of the `Course` class can be found at this link.

In the following snippet of client code, we create a `Student` with ID `123456789`, we add two courses, and finally print all courses.

```
Student student = new Student("123456789");
final int FIRST = 1020;
final int SECOND = 1030;
student.addCourse(new Course(FIRST));
student.addCourse(new Course(SECOND));
Iterator<Course> iterator = student.iterator();
while (iterator.hasNext())
{
   output.println(iterator.next());
}
```

The complete client code can be found at this link.

4.3.2 The Attributes Section

The API of the `Student` class contains a single public attribute named `NUMBER_OF_YEARS`. As we can see in the API, this attribute is static and it is a constant of type `int`. In the API, we can also find its value, which is four. Hence, we introduce the following declaration and initialization.

```
public static final int NUMBER_OF_YEARS = 4;
```

As an implementer, we have to decide how to represent the additional information: the four gpa's and the collection of courses. Since the constructors of the modified class have the same headers[10] as the those of the original class, we look at the methods for inspiration.

Let us first have a look at all those methods that manipulate the gpa's.

```
public double getGpa(int year)
public void setGpa(int year, double gpa)
```

The method `getGpa` returns the student's gpa of the given `year`. For example, `student.getGpa(2)` returns the gpa of `student` for the second year. The `setGpa` method sets the gpa of the given `year` to the given `gpa`. For example, `student.setGpa(2, 8.15)` sets the gpa of the second year of `student` to 8.15.

Given the above method headers, we may decide to represent each gpa by a `double` or an instance of its wrapper class `Double`. To represent the four gpa's we can use, for example, the following four attributes.

```
private double first;
private double second;
private double third;
private double fourth;
```

[10] The headers are not exactly the same, since we do not consider the student's name, home page and join date.

However, this choice will lead to the use of conditionals in the implementation of the methods `getGpa` and `setGpa` which renders code that is error prone and not scalable. To avoid the use of conditionals (simplifying the implementation of `getGpa` and `setGpa`), we can represent the four gpa's as a collection. In particular, we can use either a `List`, a `Set` or a `Map`. Since the collection of gpa's may contain duplicates (the student may, for example, have the exact same gpa in first and second year), a `Set` is not appropriate for representing the collection. Let us represent the collection of gpa's as a `List` of `Double`s.[11] To the attribute section of the `Student` class we add the following declaration.

```
1 private List<Double> gpas;
```

Note that `List`is an interface. As we already discussed in Section 1.6.1, using interfaces may make our code more versatile.

Let us also have a look at all those methods that manipulate the collection of courses.

```
public void addCourse(Course course)
public Iterator<Course> iterator()
```

The `addCourse` method adds the given `course` to the collection of courses taken by the student. The `iterator` method returns an `Iterator` object. The latter object can be used to enumerate the courses taken by the student using the methods `next` and `hasNext`.

Each course is represented by an instance of the `Course` class. Since the collection of courses does not contain duplicates (we do not keep track of the number of times the student takes a course), a `Set` is more appropriate than a `List`. Hence, we also add the following declaration to the attribute section of the `Student` class.

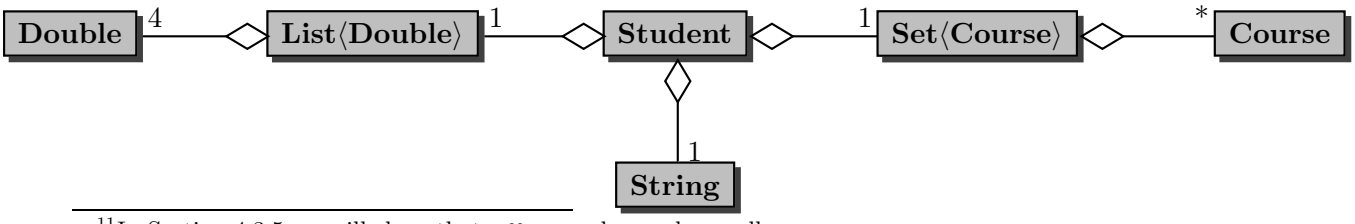
```
1 private Set<Course> courses;
```

Recall that there is a difference between `null`, which represents no object at all, and an object that represents an empty list or set. Since we want the attributes `gpas` and `courses` to always refer to a `List` object and a `Set` object, respectively, we strengthen the class invariant with the following.

```
1 this.gpas != null && this.courses != null
```

We leave it to the reader to verify that all public constructors establish this class invariant and that all public methods maintain this class invariant.

The classes discussed above are related as follows.



[11] In Section 4.3.5 we will show that a `Map` can be used as well.

The $*$ in the above class diagram indicates the multiplicity of the aggregation. It represents the number of `Course` objects that each `Student` object has: each `Student` has a variable number of `Courses`.

4.3.3 The Constructors Section

In the API of the `Student` class we find the following two constructor headers.

```
public Student(String id)
public Student(Student student)
```

The first constructor creates a `Student` with the given `id`, an empty collection of courses, and initializes all four gpa's to zero. The second one is a copy constructor. Neither the first one can delegate to the second one, nor the second one can delegate to the first one.

The one-parameter constructor has to initialize the attribute `id` to the argument of the constructor, the attribute `gpas` to a list of four zeroes, and the attribute `courses` to an empty set. Since `List` and `Set` are interfaces, they cannot be instantiated. Hence, we have to create instances of classes that implement these interfaces. The `List` interface is implemented by classes such as `ArrayList` and `LinkedList`, and the `Set` interface is implemented by classes such as `HashSet` and `TreeSet`. Let us pick `ArrayList` and `HashSet`. Then the one-parameter constructor can be implemented as follows.

```
1  public Student(String id)
2  {
3     this.setId(id);
4     List<Double> gpas = new ArrayList<Double>(Student.NUMBER_OF_YEARS);
5     for (int i = 0; i < Student.NUMBER_OF_YEARS; i++)
6     {
7        gpas.add(0.0);
8     }
9     this.setGpas(gpas);
10    this.setCourses(new HashSet<Course>());
11 }
```

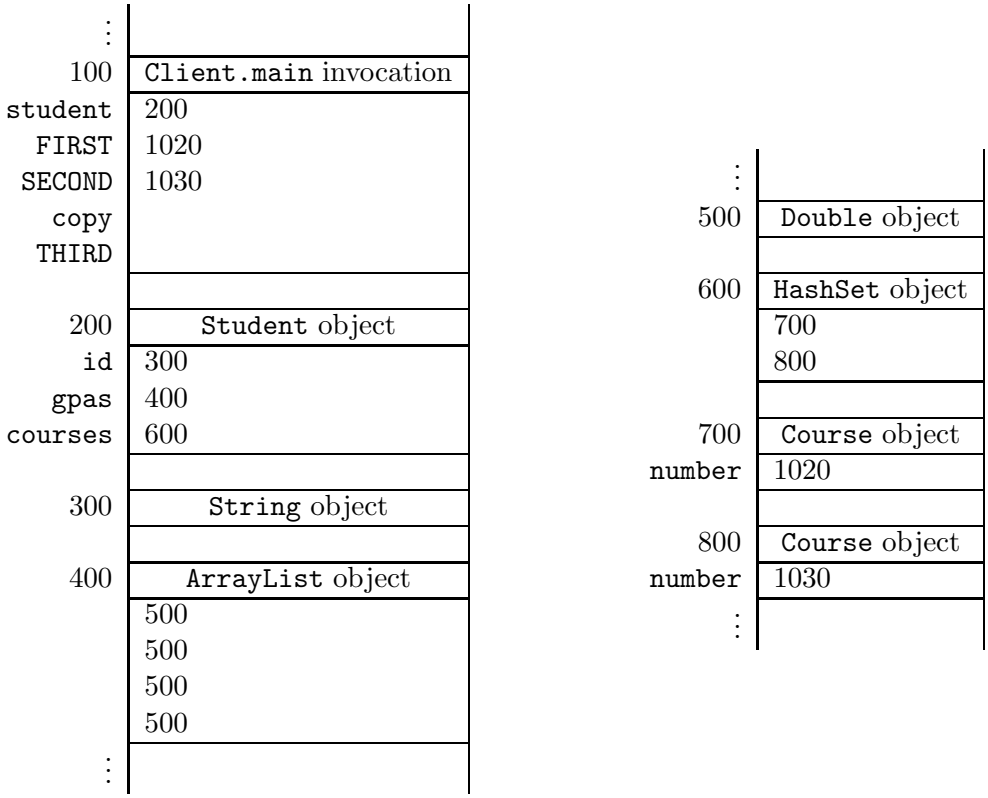
We create an empty list in line 4 and add four zeroes to that list in line 5–8. In line 10 we create an empty set.

The copy constructor can be implemented in several different ways. To compare the different implementations, we will consider the following snippet of client code.

```
1  Student student = new Student("123456789");
2  final int FIRST = 1020;
3  final int SECOND = 1030;
4  student.addCourse(new Course(FIRST));
5  student.addCourse(new Course(SECOND));
6  Student copy = new Student(student);
7  final int THIRD = 2011;
```

```
8 student.addCourse(new Course(THIRD));
```

After the execution has reached the end of line 5, memory can be depicted as follows.



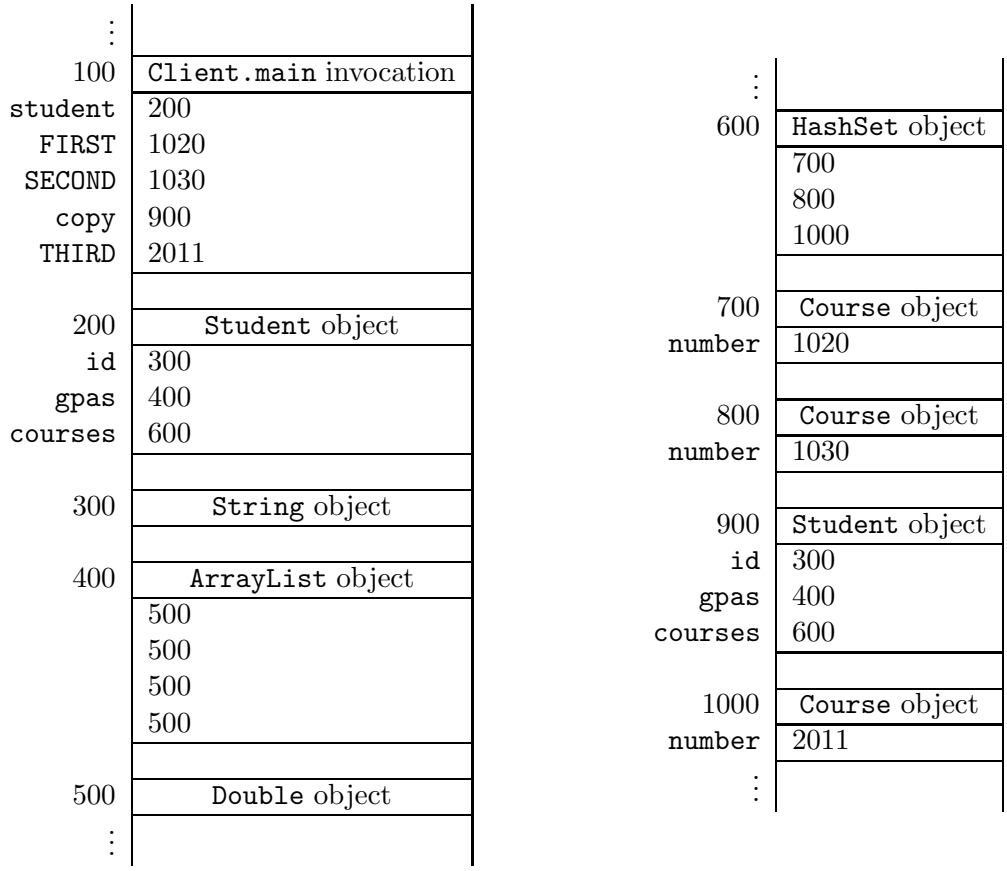
Since the class `Double` implements the pattern "one instance per state", the above memory diagram contains only a single `Double` object.

Alias

We can implement the copy constructor in such a way that the attribute `gpas` of `this` object and the attribute `gpas` of the `student` object are merely aliases, that is, they both refer to the same object. The attribute `courses` can be initialized similarly. This leads to the following implementation of the copy constructor.

```
1 public Student(Student student)
2 {
3    this.setId(student.getId());
4    this.setGpas(student.getGpas());
5    this.setCourses(student.getCourses());
6 }
```

Once the execution of the above snippet of client code reaches the end of line 8, memory can be depicted as follows.



Note that the values of the attributes `gpas` and `courses` of both `Student` objects are the same. As a consequence, both `student` and `copy` have three courses.

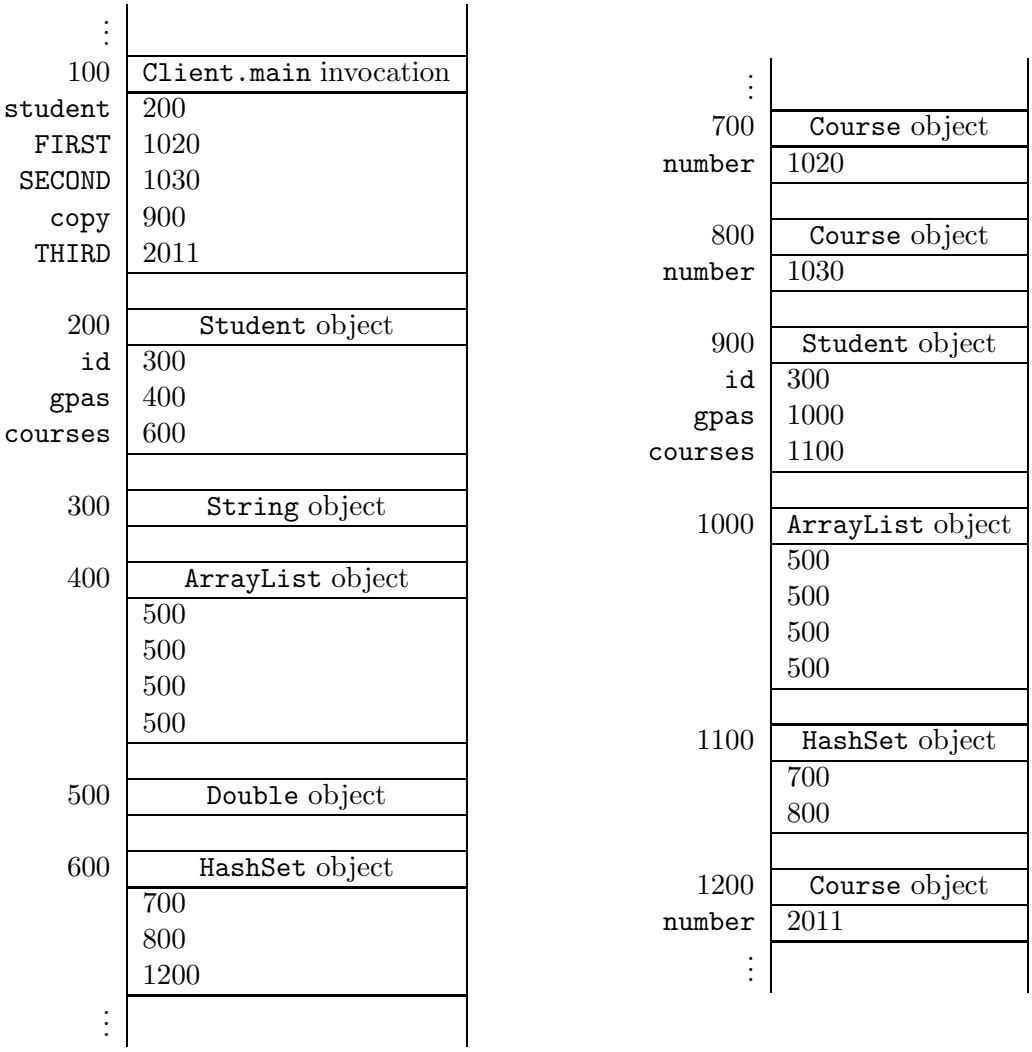
Shallow Copy

As an alternative, we can implement the copy constructor in such a way that the attribute `gpas` of the `student` object is a shallow copy of the attribute `gpas` of `this` object. The attribute `courses` can be initialized similarly. This leads to the following implementation of the copy constructor.

```
1 public Student(Student student)
2 {
3    this.setId(student.getId());
4    this.setGpas(new ArrayList(student.getGpas()));
5    this.setCourses(new HashSet(student.getCourses()));
6 }
```

We use the copy constructors of the classes `ArrayList` and `HashSet`. From the class invariant we can deduce that neither `student.getGpas()` nor `student.getCourses()` returns null. As a consequence, neither copy constructor throws a `NullPointerException`.

Once the execution of the above snippet of client code reaches the end of line 8, memory can be depicted as follows.



This time the values of the attributes `gpas` and `courses` of both the `Student` objects are different. Observe also that `student` has three courses, but `copy` has only two. Finally, note that both `HashSet` objects contain the same `Course` objects at the addresses 700 and 800.

Deep Copy

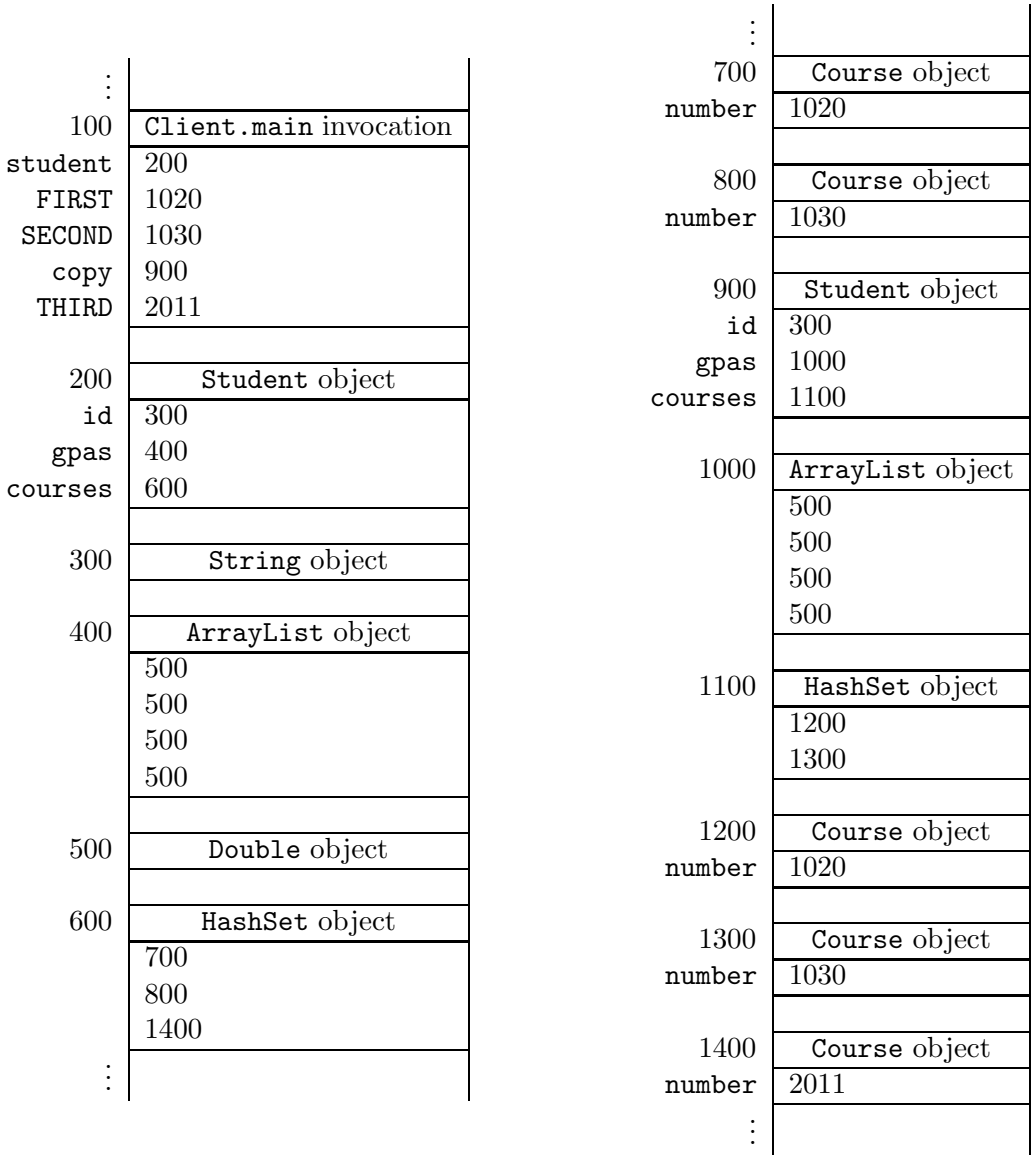
In our final implementation of the copy constructor, we ensure that the `courses` attribute of the `student` object is a deep copy of the `courses` attribute of `this` object. Since the class `Double` is immutable, it makes no sense to create a deep copy of the `gpas` attribute. Hence, we use a shallow copy instead.

```
1 public Student(Student student)
2 {
3    this.setId(student.getId());
4    this.setGpas(new ArrayList(student.getGpas()));
```

```
5     Set<Course> copy = new HashSet<Course>();
6     Iterator<Course> iterator = student.iterator();
7     while (iterator.hasNext())
8     {
9        copy.add(new Course(iterator.next()));
10    }
11    this.setCourses(copy);
12 }
```

In line 5, we create an empty collection. In line 6–10, we add a copy of each course, using the copy constructor of the `Course` class, to this collection.

Once the execution of the above snippet of client code reaches the end of line 8, memory can be depicted as follows.



This time, the `HashSet` objects contain different `Course` objects. If we were to execute also the following snippet

```
9  iterator = student.iterator();
10 Course course = iterator.next();
11 final int NEW = 1021;
12 course.setNumber(NEW);
```

only the `Course` object at address 700 would change to the following.

700	Course object
number	1021

4.3.4 The Methods Section

Here, we only discuss the methods which are new. That is, we consider the implementation of the following four methods.

```
public double getGpa(int year)
public void setGpa(int year, double gpa)
public void addCourse(Course course)
public Iterator<Course> iterator()
```

Recall that the gpa's are represented by a `List` of `Doubles`. Hence, returning the gpa of the second year amounts to returning the second element of the list. To retrieve this element from the list, we can simply delegate to the `get` method of the `List` interface. Since the index of the first element of a list is zero, the index of the gpa of the given `year` is `year` $-$ 1. Hence, the `getGpa` method can be implemented as follows.

```
1 public double getGpa(int year)
2 {
3    return this.getGpas().get(year - 1);
4 }
```

To implement the `setGpa` method, we delegate to the `set` method of the `List` interface as follows.

```
1 public void setGpa(int year, double gpa)
2 {
3    this.getGpas().set(year - 1, gpa);
4 }
```

The collection of courses is represented by a `Set` of `Courses`. Adding a given `course` to this collection can be implemented to simply delegating to the `add` method of the `Set` interface.

```
1 public void addCourse(Course course)
2 {
3    this.getCourses().add(course);
4 }
```

The `add` method of the `Set` interface returns a boolean. Note that we simply discard that returned boolean in the above method.

Finally, we implement the `iterator` method. This method returns an `Iterator` object that allows us to enumerate the courses. Again, we can simply delegate to the `Set` interface.

```
1 public Iterator<Course> iterator()
2 {
3    return this.getCourses().iterator();
4 }
```

Note that, when implementing the bodies of above methods, the implementer takes on the role of a client of the `List` and `Set` interfaces.

The interface `Iterable` contains the single method `iterator`. Since we have implemented the `iterator` method, our `Student` class implements `Iterable<Course>`. This is reflected in the header of the class `Student` as follows.

```
1 public class Student implements Comparable<Student>, Iterable<Course>
```

Implementing this interface allows a `Student` object to be the target of the "foreach" statement.[12] For example, line 6–10 of the client snippet presented in Section 4.3.1 can be replaced with

```
6 for (Course course : student)
7 {
8    output.println(course);
9 }
```

4.3.5 Beyond the Basics

Using a Map

Instead of a `List`, we can also use a `Map` to represent the four gpa's. The attribute maps years, represented by `Integer`s, to gpa's, represented by `Double`s. In our implementation we replace the declaration of a `List` with the following declaration of a `Map`.

```
1 private Map<Integer, Double> gpas;
```

To initialize this attribute, we replace line 4–9 of the one parameter constructor with

```
4 Map<Integer, Double> gpas = new HashMap<Integer, Double>();
5 for (int year = 1; year <= Student.NUMBER_OF_YEARS; year++)
6 {
7    gpas.put(year, 0.0);
8 }
9 this.setGpas(gpas);
```

Here, we only consider the copy constructor that makes a shallow copy. In that case, we replace line 4 with

```
4 this.setGpas(new HashMap<Integer, Double>(student.getGpas()));
```

In this setting the methods `getGpa` and `setGpa` can be implemented as

```
1 public double getGpa(int year)
2 {
3    return this.getGpas().get(year);
4 }
```

[12] This is something specific to Java.

and

```
1 public void setGpas(int year, double gpa)
2 {
3    this.getGpas().put(year, gpa);
4 }
```

Chapter 5

Implementing Inheritance

5.1 What is Inheritance?

Inheritance is a relation. It is also known as the *is-a* relation. It is a relation on classes. The relation captures that one class extends another class. The class that is extended is known as the *superclass* and the class that extends the superclass is known as the *subclass*. The subclass inherits certain features from the superclass. In particular, the subclass inherits the public non-static methods of its superclass.[1]

A golden rectangle is a rectangle made out of gold. Besides a width and a height, it also has a weight. The `GoldenRectangle` class extends the `Rectangle` class. It defines a special type of rectangle and adds a weight to each rectangle. Its API can be can be found at this link.

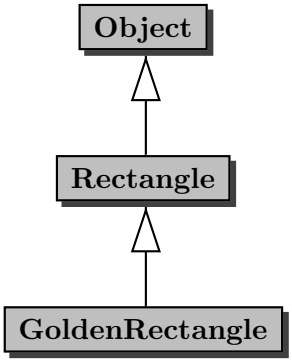
In the code snippet below, we create a `GoldenRectangle`. The `GoldenRectangle` class inherits the `scale` method of the `Rectangle` class. Phrased differently, since a `GoldenRectangle` is a `Rectangle`, we can `scale` it.

```
1 final int WIDTH = 3;
2 final int HEIGTH = 4;
3 final int WEIGHT = 80;
4 GoldenRectangle rectangle = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5 final int FACTOR = 3;
6 rectangle.scale(FACTOR);
```

5.2 UML and Memory Diagrams

The inheritance relation can be depicted in UML class diagrams. For example, the diagram

[1] It also inherits public attributes of its superclass. However, we declare all non-static attributes to be private and we access all public static attributes and all public static methods via the class(name). Hence, we can restrict our attention to public non-static methods.



captures the inheritance relation between the classes mentioned in the previous section.

Recall that the state of an object consists of its non-static attributes and their values. Although the private non-static attributes of a class are not inherited by its subclasses, they are part of the state of instances of those subclasses. For example, the `Rectangle` class contains private non-static attributes named `width` and `height`. Since our `GoldenRectangle` class is a subclass of the `Rectangle` class, the attributes `width` and `height` and their values are part of the state of a `GoldenRectangle` object.

Since a `GoldenRectangle` has a weight, we will introduce an attribute named `weight` to capture that additional information.

Consider the client code of the previous section. When we reach the end of line 4, memory can be depicted as follows.

⋮	
100	`Client.main` invocation
`WIDTH`	3
`HEIGHT`	4
`WEIGHT`	80
`rectangle`	
`FACTOR`	
200	`GoldenRectangle` object
`width`	3
`height`	4
`weight`	80
⋮	

Note that we colour the part of the state which corresponds to attributes of the superclass. This reflects that these attributes cannot be accessed directly.

5.3 The `GoldenRectangle` Class

As we already mentioned in Section 5.1, a `GoldenRectangle` is a `Rectangle` made of gold and, hence, our class `GoldenRectangle` extends the `Rectangle` class. To reflect that the `GoldenRectangle` class is a subclass of the `Rectangle` class, we use the following class header.

```
1 public class GoldenRectangle extends Rectangle
```

5.3.1 The Attributes Section

A `GoldenRectangle` is a `Rectangle` with some additional information, namely its weight (in grams). Based on the signature of the three parameter constructor and the signature of the accessor `getWeight`, we decide to represent this additional information by means of an attribute of type `int`. We declare this attribute as follows.

```
1 private int weight;
```

Note that we do *not* introduce attributes to capture the width and height of the golden rectangle. This information is already represented by attributes of the `Rectangle` class and, hence, is already part of the state of a `GoldenRectangle` object.

As class invariant, we use

```
1 this.weight >= 0
```

5.3.2 The Constructors Section

Recall that the state of an object is initialized in the constructors. Since the attributes `width` and `height` are part of the state of a `GoldenRectangle` object, in the snippet

```
1 final int WIDTH = 3;
2 final int HEIGHT = 4;
3 final int WEIGHT = 80;
4 GoldenRectangle rectangle = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
```

the attributes `width`, `height` and `weight` get initialized. But how can we initialize the attributes `width` and `height`? These attributes are private and, hence, *cannot* be accessed directly. However, within the body of a constructor we can delegate to a constructor of the superclass. Hence, within the body of the three parameter constructor of the `GoldenRectangle`, we can delegate to the two parameter constructor of its superclass `Rectangle`.

To delegate to a constructor of the superclass, we use the keyword `super`. In the three parameter constructor of the `GoldenRectangle` class, we delegate to the two parameter constructor of the superclass `Rectangle` using `super(width, height)`. In this way, we initialize the attributes `width` and `height`. To initialize the weight, we use the private mutator `setWeight`. This results in the following constructor implementation.

```
1 public GoldenRectangle(int width, int height, int weight)
2 {
3    super(width, height);
4    this.setWeight(weight);
5 }
```

Consider the above lines of client code. When executing line 4 of the client code, first a block of memory for the attributes of the `GoldenRectangle` class and its superclass `Rectangle` is allocated.

⋮	
100	`Client.main` invocation
`WIDTH`	3
`HEIGHT`	4
`WEIGHT`	80
`rectangle`	
200	`GoldenRectangle` object
`width`	
`height`	
`weight`	
⋮	

Next, the three parameter constructor of the `GoldenRectangle` class is invoked. The corresponding invocation block can be depicted as follows.

⋮	
800	`GoldenRectangle` invocation
`this`	200
`width`	3
`height`	4
`weight`	80
⋮	

Within the body of the three parameter constructor of the `GoldenRectangle` class, the two parameter constructor of the `Rectangle` class is invoked by means of `super(width, height)`. Recall that we implemented the two parameter constructor of the `Rectangle` class as follows.

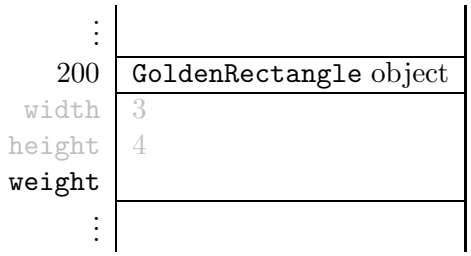
```
1 public Rectangle(int width, int height)
2 {
3    this.width = width;
4    this.height = height;
5 }
```

The invocation `super(width, height)` gives rise to an invocation block which can be depicted as follows.

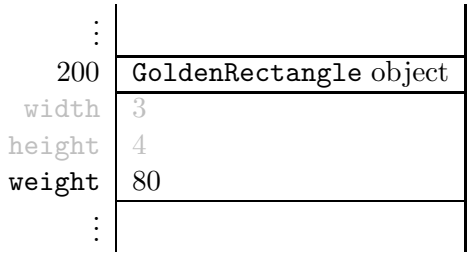
⋮	
900	`Rectangle` invocation
`this`	200
`width`	3
`height`	4
⋮	

Note that in the invocation `super(width, height)`, the parameter `this` is implicit.

Within the body of the two parameter constructor of the `Rectangle` class, we assign values to the attributes `width` and `height` of `this` object, that is, the object at address 200. Hence, once we reach the end of line 4 of the above constructor, the attributes `width` and `height` of the object at address 200 have the values 3 and 4, respectively.



Next, line 4 of the three parameter constructor of the `GoldenRectangle` class is executed. In this line, the `weight` attribute is initialized. Once the execution reaches the end of that line, the `GoldenRectangle` object can be depicted as follows.



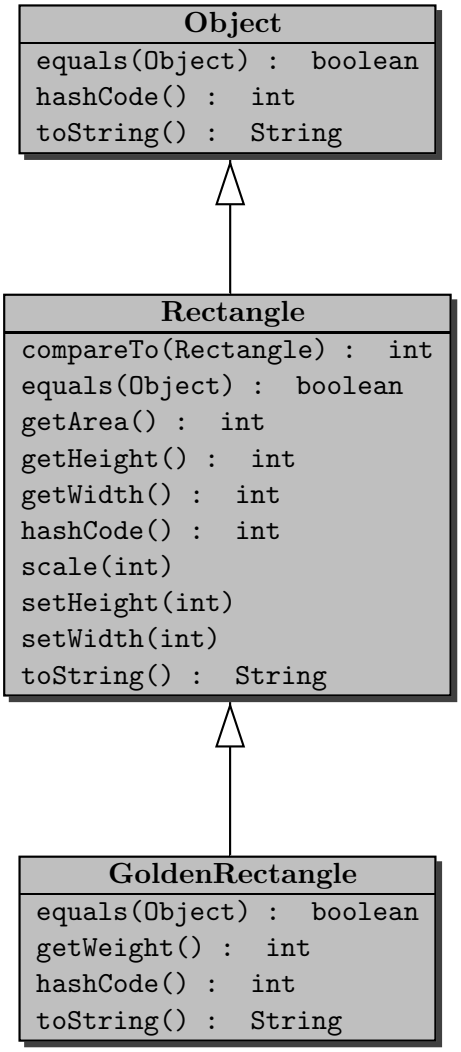
Next, we implement the copy constructor. It can be implemented in several different ways. For example, we can delegate to the copy constructor of the superclass.

```
1 public GoldenRectangle(GoldenRectangle copied)
2 {
3    super(copied);
4    this.setWeight(copied.getWeight());
5 }
```

The Java compiler requires that the `super(...)` statement be the first statement of the constructor.

5.3.3 The Methods Section

Before we implement the methods of our `GoldenRectangle` class, let us have a look at its superclasses `Object` and `Rectangle` and some of their methods.



Note that we override the methods `equals`, `hashCode` and `toString`. The method `getWeight` is new.

The `equals` Method

Two golden rectangles are the same if they have not only the same width and height, but also the same weight. To implement the `equals` method, we start with the usual skeleton.

```
1 public boolean equals(Object object)
2 {
3    boolean equal;
4    if (object != null && this.getClass() == object.getClass())
5    {
```

```
6        GoldenRectangle other = (GoldenRectangle) object;
7        equal = ???;
8     }
9     else
10    {
11       equal = false;
12    }
13 }
```

To check if `this` golden rectangle has the same weight as the `other` golden rectangle we can use

```
7 this.getWeight() == other.getWeight()
```

To check if both golden rectangles have the same width and height, we can delegate to the `equals` method of the `Rectangle` class as follows.[2]

```
7 super.equals(other)
```

Combing the two, we arrive at the following.

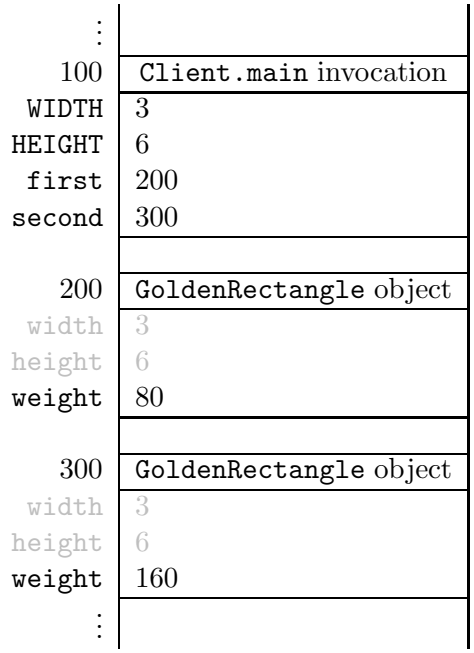
```
7 equal = super.equals(other) && this.getWeight() == other.getWeight();
```

Consider the following snippet of client code.

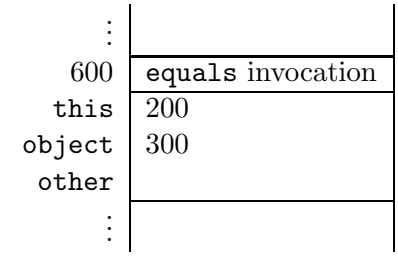
```
1 final int WIDTH = 3;
2 final int HEIGHT = 6;
3 final int WEIGHT = 80;
4 GoldenRectangle first = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5 GoldenRectangle second = new GoldenRectangle(WIDTH, HEIGHT, 2 * WEIGHT);
6 output.println(first.equals(second));
```

Once we reach the end of line 5, memory can be depicted as follows.

[2]Note that `super.` is used to invoke a method of the superclass, whereas `super(...)` is used to delegate to a constructor of the superclass.

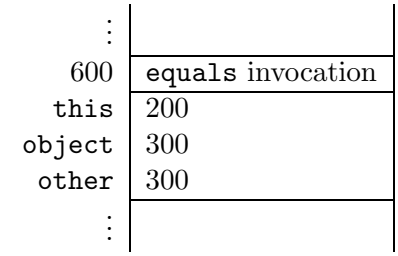


The invocation of the `equals` method in line 6 of the client code gives rise to the following invocation block.

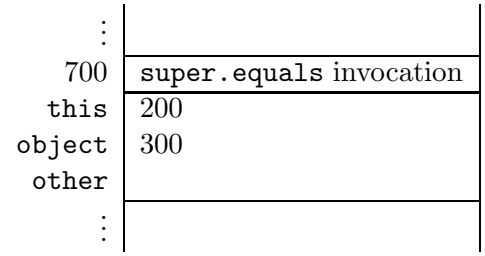


The invocation block contains the parameters `this` and `object` and the local variable `other`.

Now, let us consider the execution of the body of the `equals` method. Since `object` is different from `null` and both `this` and `object` refer to a `GoldenRectangle` object, line 6 and 7 of the `equals` method are executed. In line 6, the local variable `other` is initialized.



In line 7, the `equals` method of the superclass is invoked. This leads to the following invocation block.



Again, the invocation block contains the parameters `this` and `object` and the local variable `other`. Note that the parameter `this` is implicit in the invocation `super.equals(other)`. The `equals` method of the `Rectangle` class is invoked on the same object on which the `equals` method of the `ColouredRectangle` class is invoked.[3]

Let us consider the execution of the body of the `equals` method of the `Rectangle` class. Recall that we implemented the `equals` method in the `Rectangle` class 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.getWidth() == other.getWidth()) && (this.getHeight() ==
             other.getHeight());
8     }
9     else
10    {
11       equal = false;
12    }
13    return equal;
14 }
```

Since `object` is different from `null` and both `this` and `object` refer to a `GoldenRectangle` object, line 6 and 7 of the `equals` method are executed. In line 6, the local variable `other` is initialized to (the object at address) 300. In line 7, the width and height of `this` object, that is, the object at address 200, and of the `other` object, that is, the object at address 300, are compared.

In the next fragment of client code we compare a rectangle with a golden rectangle. Consider the following snippet of client code.

```
1  final int WIDTH = 3;
2  final int HEIGHT = 6;
3  final int WEIGHT = 80;
4  GoldenRectangle first = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5  Rectangle second = new Rectangle(WIDTH, HEIGHT);
```

[3] One may think of `super.equals(other)` as `this.super.equals(other)` where `super.equals` denotes the `equals` method of the `superclass`. Note, however, that `this.super.equals(other)` is not valid Java syntax.

```
6 output.println(first.equals(second));
7 output.println(second.equals(first));
```

Neither `first` is equal to `second`, nor `second` is equal to `first`, since `first` and `second` are instances of different classes and, hence, their `Class` objects (returned by the accessor `getClass`) are different. We leave it to the interested reader to verify that the `equals` method satisfies the usual properties. We will come back to the `equals` method and its properties in Section 5.7.

The `hashCode` Method

The `hashCode` method returns an integer. This integer can be thought of as an abstraction of the state of the object. As a consequence, the value returned by the `hashCode` method is usually defined in terms of the values of the attributes of the class and its superclasses. For our `GoldenRectangle` class, the attributes `width`, `height` and `weight` and their values make up the state. The `hashCode` method of the superclass `Rectangle` already takes the attributes `width` and `height` into account. In the `hashCode` method of the `GoldenRectangle` class, we combine the result of the `hashCode` method of the `Rectangle` class and the `weight` attribute. The two integers can be combined in many ways. In superclass `Rectangle` we simply add the width and the height. Here we combine them as follows.[4]

```
1 public int hashCode()
2 {
3    final int BASE = 37;
4    return super.hashCode() + BASE * this.getWeight();
5 }
```

Recall that the `hashCode` method and the `equals` method are closely related. Since we override the `equals` method and the `hashCode` method, we have to check if they satisfy the usual 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. We leave it to the interested reader to check that this is indeed the case.

The `toString` Method

The `toString` method returns a string representation of the golden rectangle. For example, the snippet of client code

```
1 final int WIDTH = 3;
2 final int HEIGHT = 6;
3 final int WEIGHT = 80;
```

[4]Let b be a natural number. We call b the *base*. The *polynomial* hashcode of the integers $\langle i_0, \ldots, i_n \rangle$ is defined by

$$\sum_{k=0}^{n} i_k b^k.$$

The base b is usually a prime number. The details are beyond the scope of this book.

```
4 GoldenRectangle rectangle = new GoldenRectangle(WIDTH, HEIGHT, WEIGHT);
5 output.println(rectangle);
```

produces the output

```
GoldenRectangle of width 3 and height 6 and weight 80
```

Note that part of this string, namely `Rectangle of width 3 and height 6`, is the same as the string representation returned by the `toString` method of the `Rectangle` class for a rectangle of width 3 and height 6. Hence, we can delegate to the `toString` method of the superclass to obtain this part. Therefore, we can implement the `toString` method as follows.

```
1 public String toString()
2 {
3    return "Golden" + super.toString() + " and weight " + this.getWeight();
4 }
```

The code of the `GoldenRectangle` class can be found by following this link.

5.4 The `PricingException` Class

Next we add the method

```
public double getPrice() throws PricingException
```

to our `GoldenRectangle` class. This method returns the price (in Canadian dollars) of the golden rectangle. The price is based on the weight of the golden rectangle and the current gold price. The method throws a `PricingException` if the gold price cannot be determined.

To determine the gold price, we utilize the method

```
public static double priceIn(String currencyCode)
  throws MalformedURLException, IOException, IndexOutOfBoundsException
```

of the `LondonGoldExchange` class. The API of this class can be found by following this link. The method returns the current price of one gram of gold in the given currency at the London Gold Exchange. The currency code for Canadian dollar is CAD. The method may throw three different types of exceptions. We refer the reader to the API of the `LondonGoldExchange` class for the conditions under which these exceptions are thrown. Since we want to hide these details from the client of our `getPrice` method, we introduce a new exception class, named `PricingException` which will be thrown whenever something goes wrong when determining the gold price.

First, we will implement the `getPrice` method. After that, we will implement the `PricingException` class.

The price of the golden rectangle is simply the product of its weight and the price of one gram of gold.

```
1 return this.getWeight() * LondonGoldExchange.priceIn("CAD");
```

Note that there is no need to introduce an attribute for the price, since it would be redundant. Also note that we should not cache the price, because the gold price varies over time.

Whenever the `priceIn` method throws an exception, our `getPrice` method throws a `PricingException`. This can be accomplished by catching the exceptions thrown by the `priceIn` method, and throwing instead a `PricingException`. This leads to the following implementation.

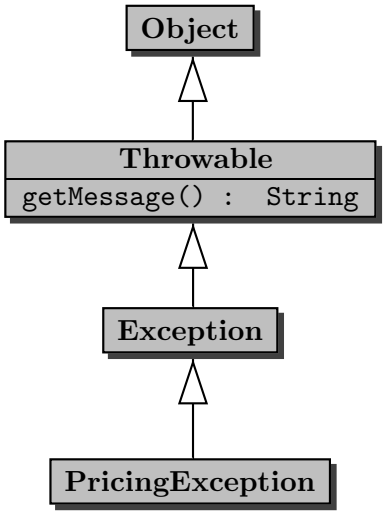
```
1  public double getPrice() throws PricingException
2  {
3     try
4     {
5        return this.getWeight() * LondonGoldExchange.getPrice("CAD");
6     }
7     catch (MalformedURLException e)
8     {
9        throw new PricingException("The gold price could not be determined");
10    }
11    catch (IOException e)
12    {
13       throw new PricingException("The gold price could not be determined");
14    }
15    catch (IndexOutOfBoundsException e)
16    {
17       throw new PricingException("The gold price could not be determined");
18    }
19 }
```

The `getPrice` method throws a `PricingException` if the gold price cannot be determined. Rather than throwing an instance of an already existing exception class, we decided to introduce a new exception class, `PricingException`, and to throw an instance this new class. Introducing our own exception class allows the client to easily detect that this exception is thrown by our `getPrice` method. Hence, this allows us to separate the ordinary code from the exception handling code.

Next, we implement the `PricingException` class. Its API can be can be found at this link. A `PricingException` is an exception and, hence, our class `PricingException` extends the class `Exception`. This is reflected in the header of our class.

```
1  public class PricingException extends Exception
```

Since the `PricingException` class extends the `Exception` class, it inherits all public non-static methods of the `Exception` class and its superclasses `Throwable` and `Object`. For example, it inherits the `getMessage` method of the `Throwable` class.



5.4.1 The Attributes Section

A `PricingException` is just a special type of exception, but contains no additional information. Therefore, we do not introduce any new attributes.

5.4.2 The Constructors Section

Recall that the private non-static attributes of the superclasses are part of the state. For example, the `Throwable` class contains a private non-static attribute named `detailMessage` of type `String`. This attribute contains the string which is returned by the `getMessage` method. Since our `PricingException` class is a subclass of the `Throwable` class, the attribute `detailMessage` and its value are part of the state of a `PricingException` object.

To initialize the attributes of the superclasses, such as `detailMessage`, we delegate to a constructor of the superclass.

```
1 public PricingException(String message)
2 {
3    super(message);
4 }
```

Note that the `Throwable` class does not contain a mutator `setMessage` and, hence, the attribute `detailMessage` can only be initialized by delegation to a constructor of the superclass `Exception`. That constructor of the superclass `Exception` in turn delegates to a constructor of its superclass, `Throwable`. In that constructor of the `Throwable` class the attribute `detailMessage` is initialized.

To implement the default constructor of our `PricingException` class, we simply delegate to the default constructor of its superclass.

```
1 public PricingException()
2 {
3    super();
4 }
```

5.4.3 The Methods Section

Our `PricingException` class contains no new public methods. Hence, we do not have to implement any. Note, however, that our class inherits methods from the classes `Object` and `Throwable` such as `equals` and `getMessage` (the class `Exception` contains no new public methods either).

The code of the `PricingException` class can be found by following this link.

5.5 The `ColouredRectangle` Class

Next, we consider a different extension of the `Rectangle` class. This time we add some colour to the rectangle. Its API can be can be found at this link. To reflect that our `ColouredRectangle` class extends the `Rectangle` class, we use the following class header.

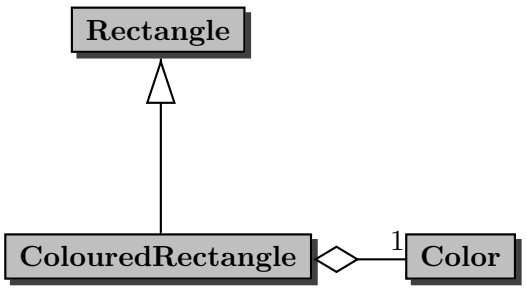
```
1 public class ColouredRectangle extends Rectangle
```

5.5.1 The Attributes Section

A `ColouredRectangle` is a `Rectangle` with some additional information, namely its colour. Based on the signature of the three parameter constructor and the signature of the accessor `getColour` and the return type of the mutator `setColour`, we decide to represent this additional information by means of the attribute of type `Color` (the class `Color` is part of the `java.awt` package). We declare the corresponding attribute as follows.

```
1 private Color colour;
```

The classes `ColouredRectangle`, `Rectangle` and `Color` are related as follows.



Note that this example combines inheritance and aggregation.

5.5.2 The Constructors Section

The API contains two constructors: a three parameter constructor and a two parameter constructor. Let us first consider the three parameter constructor. Again, we delegate to the superclass to initialize its part of the state.

```
1 public ColouredRectangle(int width, int height, Color colour)
2 {
```

```
3     super(width, height);
4     this.setColour(colour);
5  }
```

In the two parameter constructor, the width and height are set to the given `width` and `height`, and the colour is set to be white. In this case we can delegate to the three parameter constructor as follows.

```
1  public ColouredRectangle(int width, int height)
2  {
3     this(width, height, Color.WHITE);
4  }
```

5.5.3 The Methods Section

Of the methods of the `ColouredRectangle` class, we only discuss the `equals` method. Two coloured rectangles are the same if they not only have the same width and height, but also the same colour. To compare the width and the height, we delegate to the `equals` method of the superclass. This leads to the following implementation.

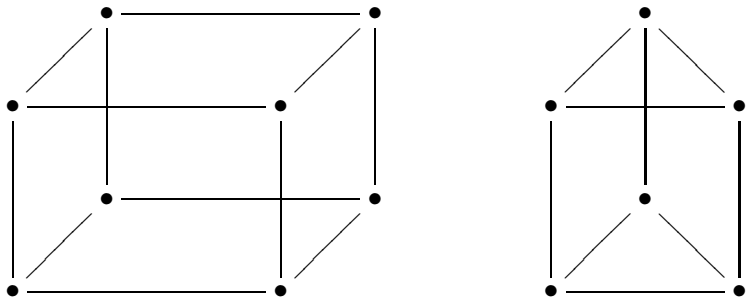
```
1  public boolean equals(Object object)
2  {
3     boolean equal;
4     if (object != null && this.getClass() == object.getClass())
5     {
6        ColouredRectangle other = (ColouredRectangle) object;
7        equal = super.equals(other) && this.getColour().equals(other.getColour()
           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

Note that we use the `equals` method of the `Rectangle` class (through inheritance) and the `equals` method of the `Color` class (through aggregation).

The code of the `ColouredRectangle` class can be found by following this link.

5.6 Implementing Abstract Classes

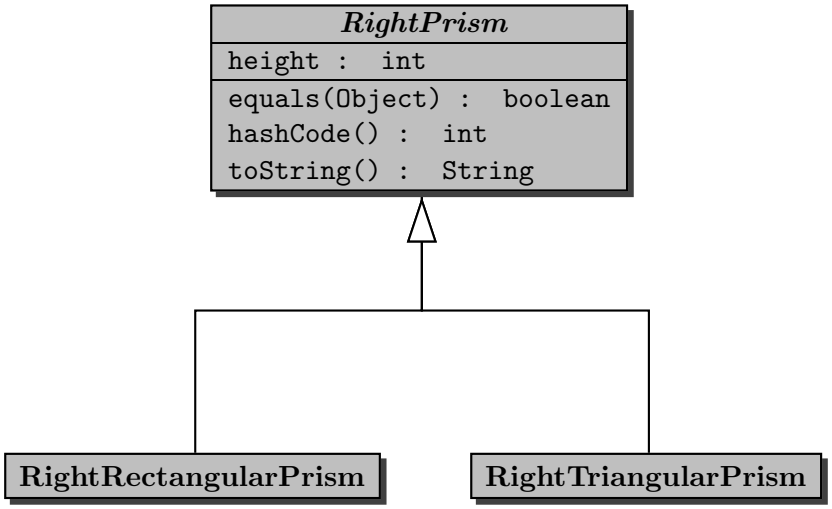
Abstract classes are often introduced to avoid code duplication. For example, consider the classes `RightRectangularPrism` and `RightTriangularPrism`.



The base of a right rectangular prism is a rectangle and the base of a right triangular prism is a triangle. In a right rectangular prism all angles are 90°. In a right triangular prism, apart from the six angles of the two triangles, all other angles are 90°. Both a right rectangular prism and a right triangular prism have a height. Hence, we may represent this information by the following attribute.

```
1 private int height;
```

Rather than duplicating this code in the `RightRectangularPrism` class and in the `RightTriangularPrism` class, we introduce a common superclass, named `RightPrism`, that contains the common code. Its API can be can be found at this link.



To prevent the client from creating an instance of the `RightPrism` class, we declare the class to be abstract.

```
1 public abstract class RightPrism
```

If the client were to attempt to create an instance of this abstract class

```
1 final int HEIGHT = 5;
2 RightPrism prism = new RightPrism(HEIGHT);
```

then a compile time error would occur

```
RightPrismClient.java:2: RightPrism is abstract; cannot be instantiated
        RightPrism prism = new RightPrism(HEIGHT);
                           ^
1 error
```

As we already mentioned, the `RightPrism` class contains the above declaration of the attribute `height`. The corresponding accessor and mutator are private.

Although the client cannot create an instance of the `RightPrism` class, we do add a constructor to the class so that subclasses can delegate to it to initialize the `height` attribute.

```
1  public RightPrism(int height)
2  {
3     super();
4     this.setHeight(height);
5  }
```

Note that we delegate to the default constructor of the superclass, the `Object` class. If a constructor does not explicitly invoke a superclass constructor, then the Java compiler automatically inserts `super()` and, hence, `super()` can be left out in the above constructor.[5]

The `RightPrism` class contains the methods `equals`, `hashCode` and `toString` which all depend on the `height` attribute. For example, 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        RightPrism other = (RightPrism) object;
7        equal = this.getHeight() == other.getHeight();
8     }
9     else
10    {
11       equal = false;
12    }
13 }
```

[5]If the Java compiler automatically inserts `super()` and the superclass does not have a default constructor, then a compile-time error will occur.

The code of the `RightPrism` class can be found by following this link.

Let us only have a look at the `RightRectangularPrism` class. Its API can be can be found at this link. This class extends the abstract `RightPrism` class which is reflected in the class header.

```
1 public class RightRectangularPrism extends RightPrism
```

As a consequence, the `RightRectangularPrism` class inherits methods from the `Object` class, such as `getClass`, and from the `RightPrism` class, namely `equals`, `hashCode`, and `toString`. Furthermore, the attribute `height`, although not inherited, is part of the state of a `RightRectangularPrism` object.

Because a right rectangular prism can be represented by its width, height and depth, and since the height is already represented in the superclass, we only need to introduce the following two attributes.

```
1 private int width;
2 private int depth;
```

The state of a `RightRectangularPrism` object, consisting of its `width`, `height` and `depth`, is initialized as follows.

```
1 public RightRectangularPrism(int width, int height, int depth)
2 {
3    super(height);
4    this.setWidth(width);
5    this.setDepth(depth);
6 }
```

Note that we delegate to the constructor of the `RightPrism` class.

In the `RightRectangularPrism` class we override the methods `equals`, `hashCode` and `toString`. For example, 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        RightRectangularPrism other = (RightRectangularPrism) object;
7        equal = super.equals(other)
8                && this.getWidth() == other.getWidth()
9                && this.getDepth() == other.getDepth();
10    }
11    else
12    {
13       equal = false;
14    }
15 }
```

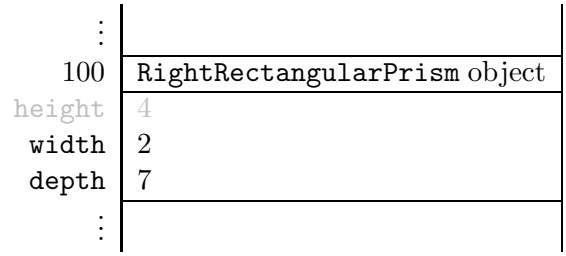
Note that we delegate to the `equals` method of the `RightPrism` class.

The code of the `RightRectangularPrism` class can be found by following this link.

5.7 Beyond the Basics

5.7.1 The `getVolume` Method

We want to add a `getVolume` method to the classes that represent right prisms such as `RightRectangularPrism` and `RightTriangularPrism`. Consider, for example, the `RightRectangularPrism` class. In this case, its volume is defined as the product of its height, width and depth. However, in the `RightRectangularPrism` we cannot compute this product since the attribute `height` is not accessible, as is reflected in the following object block.



Each right prism has a base area. However, the base area is computed differently for different types of right prisms. For example, for a right rectangular prism its base area is the product of its width and depth. Note that this product can be computed in the `RightRectangularPrism` class as follows.

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

The facts that each right prism has a base area but it cannot be computed in the `RightPrism` class, can be reflected by adding the following abstract method to the `RightPrism` class.

```
1 public abstract int getBaseArea();
```

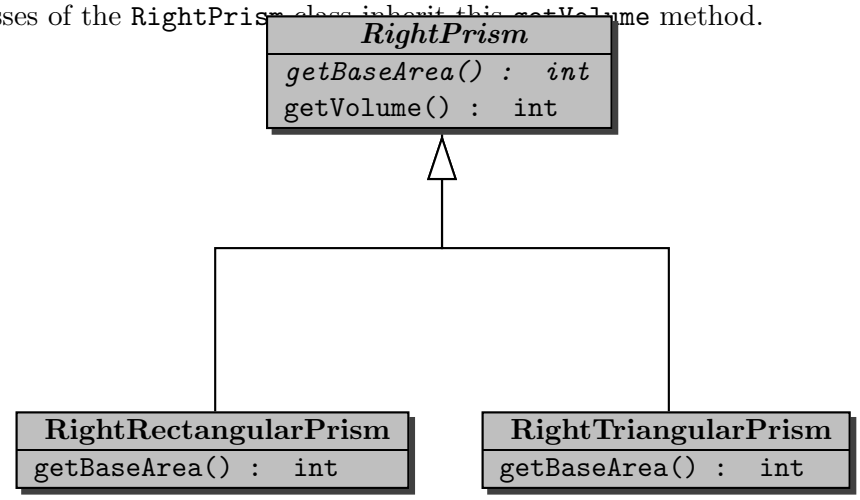
By introducing this method declaration, we enforce that each (non-abstract) subclass has to provide an implementation of the `getBaseArea` method. If we were to extend the `RightPrism` class and not implement the `getBaseArea` method, then we would get a compile time error such as

```
IncompleteRightPrism.java:1: IncompleteRightPrism is not abstract and
does not override abstract method getBaseArea() in RightPrism
public class IncompleteRightPrism extends RightPrism
       ^
1 error
```

Given that we have an (abstract) `getBaseArea` method in the `RightPrism` class, we can now implement the `getVolume` method in the `RightPrism` class as follows.

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

All subclasses of the `RightPrism` class inherit this `getVolume` method.



When we compile the `RightPrism` class, the invocation `this.getBaseArea()` in the body of the `getVolume` method is bound to the `getBaseArea` method of the `RightPrism` class. At run time, the `getBaseArea` method is invoked on an instance of a (non-abstract) subclass of the `RightPrism` class. As we already mentioned above, this subclass has to implement the `getBaseArea` method. Hence, at run time the invocation `this.getBaseArea()` in the body of the `getVolume` method is bound to the `getBaseArea` method of that subclass of the `RightPrism` class.

5.7.2 Preconditions

Recall that we implemented the `scale` method of the `Rectangle` class as follows.

```
1  /**
2   * Scale this rectangle with the given factor.
3   *
4   * @param factor scaling factor.
5   * @pre. factor >= 0
6   */
7  public void scale(int factor)
8  {
9     this.width *= factor;
10    this.height *= factor;
11 }
```

The `Factory` class contains a method

```
public static Rectangle getInstance()
```

which returns a `Rectangle` with random width and height.

A client uses the above classes in an app as follows.

```
1 Rectangle rectangle = Factory.getInstance();
2 rectangle.scale(0);
3 output.println(rectangle);
```

When the client runs the app, it does not produce the expected output

```
Rectangle with width 0 and height 0
```

After inspecting the API of our `Rectangle` class, the client blames us since the client has satisfied the precondition of our `scale` method, yet our method does not return the expected string.

How is it possible that the `scale` method does not return the expected string? Are we to blame?

After inspecting our `scale` method, we are convinced that it is correct. Hence, we are not to blame. However, the client is not to blame either. So, who is to blame?

The implementer of the `Factory` class also wrote the `HiddenRectangle` class. This class extends the `Rectangle` class and overrides the `scale` method. The precondition of the `scale(factor)` method of the `HiddenRectangle` class is `factor` > 0 and the method does nothing in case the precondition is not met (recall that as an implementer we can do whatever we want if the precondition is not met).

The implementer of the `Factory` class implemented the `getInstance` method in such a way that it returns a `HiddenRectangle` object which is-a `Rectangle`. As a consequence, at compile time, the method call `rectangle.scale(0)` is bound to the `scale` method of our `Rectangle` class. However, at run time, the method call `rectangle.scale(0)` is bound to the `scale` method of the `HiddenRectangle` class. Hence, it does not set the width and height to zero.

Note that the precondition of the `scale` method in the `HiddenRectangle` class strengthens the precondition of the `scale` method in the `Rectangle` class: `factor > 0` is stronger than `factor >= 0` since the former implies the latter. As a consequence, the `Rectangle` class guarantees that the `scale` method works as expected if the argument 0 is provided, whereas the `HiddenRectangle` class does not. However, a `HiddenRectangle` object is-a `Rectangle` and, hence, should behave like a `Rectangle`. In particular, its `scale` method should work as expected if the argument 0 is provided. Therefore, we blame the implementer of the `HiddenRectangle` class: the precondition of the `scale` method should not have been strengthened.

In a subclass, we can weaken the precondition as is shown in the following alternative implementation of the `HiddenRectangle` class.

```
1 public class HiddenRectangle extends Rectangle
2 {
3    /**
4     * Scale this rectangle with the given factor.
5     * If the factor is negative then its absolute value is used.
6     *
7     * @param factor scaling factor.
8     * @pre. true
9     */
```

```
10    public void scale(int factor)
11    {
12       super.scale(Math.abs(factor));
13    }
14 }
```

5.7.3 Postconditions

Similarly, it can be shown that the postcondition of a method cannot be weakened in a subclass. It can be strengthened though.

5.7.4 Exceptions

From the API of the `Rectangle` class we can conclude that the `scale` method does not throw an exception. Hence, in the snippet

```
1 public static void main(String[] args)
2 {
3    Rectangle rectangle = Factory.getInstance();
4    final int FACTOR = 3;
5    rectangle.scale(FACTOR);
6 }
```

we do not need to enclose the `scale` invocation in a try block. As we have seen above, the `getInstance` method may return an instance of a subclass of the `Rectangle` class. If the subclass overrides the `scale` method and the `getInstance` method returns an instance of the subclass, then the overridden `scale` method is invoked in line 5. As a consequence, the overridden `scale` method cannot throw an exception either.

If we were to override the `scale` method in the `HiddenRectangle` class and throw a `RectangleException` if the argument of the `scale` method were negative and we were to compile the `HiddenRectangle` class, then we would get the following error message.

```
HiddenRectangle.java:9: scale(int) in HiddenRectangle cannot override scale(int)
in Rectangle;
overridden method does not throw RectangleException
    public void scale(int factor) throws RectangleException
                ^
1 error
```

In summary, when we override a method, we cannot throw any exception that is not thrown by the method in the superclass.

As we have seen, the `getPrice` method of the `GoldenRectangle` class may throw an exception of type `PricingException`. If we override the `getPrice` method in a subclass of the `GoldenRectangle` class, then we can throw an exception provided that it is an instance of `PricingException` or one of its subclasses.

5.7.5 The `equals` Method Revisited

In our implementation of the `equals` method, we use the `getClass` method to check if the two objects are of the same type. Can `instanceof` be used instead? Assume we use

```
1   if (object != null && object instanceof ColouredRectangle)
```

instead of

```
1   if (object != null && this.getClass() == object.getClass())
```

in the `equals` method of the `ColouredRectangle` class. The `equals` method of the `Rectangle` class can be modified similarly. Now consider the following snippet of client code.

```
1 final int WIDTH = 3;
2 final int HEIGHT = 6;
3 ColouredRectangle first = new ColouredRectangle(WIDTH, HEIGHT, Color.RED);
4 Rectangle second = new Rectangle(WIDTH, HEIGHT);
5 output.println(first.equals(second));
6 output.println(second.equals(first));
```

On the one hand, because `second` is not an instance of `ColouredRectangle`, the invocation `first.equals(second)` returns false. On the other hand, since `first instanceof Rectangle` returns true and `first` and `second` have the same width and height, the invocation `second.equals(first)` returns true. Hence, this implementation gives rise to an `equals` method that is not symmetric. Since symmetry is part of the postcondition of the `equals` method of the `Object` class and, as we have seen above, the postcondition cannot be weakened, the `equals` method of the `ColouredRectangle` class should be symmetric as well. Since the implementation of the `equals` method using `instanceof` weakens the postcondition, it is incorrect.

Chapter 6

Implementing Graphical User Interfaces

6.1 Introduction

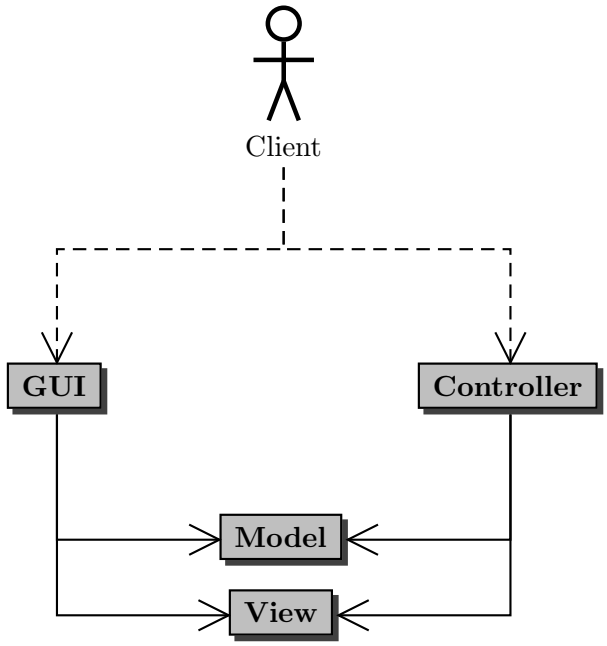
To see aggregation and inheritance in action, we implement a graphical user interface (GUI for short). This chapter is *not* about GUIs, but we do introduce some of the concepts that play a role in the implementation of GUIs. The focus of this chapter is aggregation and inheritance.

To implement a GUI, we exploit the model-view-controller (MVC) pattern. This pattern decouples the data, the graphical representation of the data, and the interactions of the client with the data. More precisely, the MVC pattern is based on the following three elements.

- The *model* represents the data and provides ways to manipulate the data.
- The *view* provides a graphical representation of the model on the screen.
- The *controller* translates the client's interactions with the view into actions that manipulate the view and the model. These interactions can be menu selections, button clicks, etc.

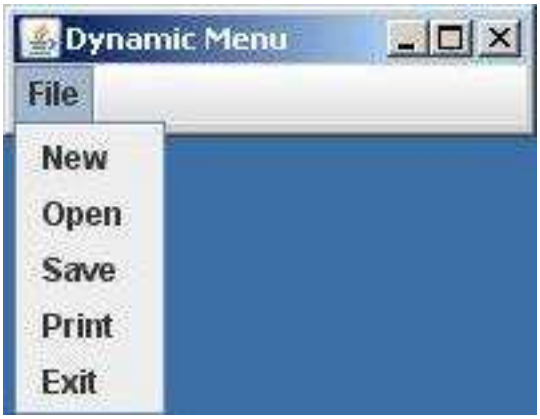
In our implementation, we introduce the classes `Model`, `View` and `Controller`. Furthermore, we develop an app, named `GUI`, which is launched by the client to run the GUI.

The client launches the app `GUI`. The app creates a `Model` and a `View`. Subsequently, the `View` creates a `Controller`. The interactions of the client with the GUI via the `View` are handled by the `Controller`. The `Controller` subsequently updates the `Model` and the `View`.

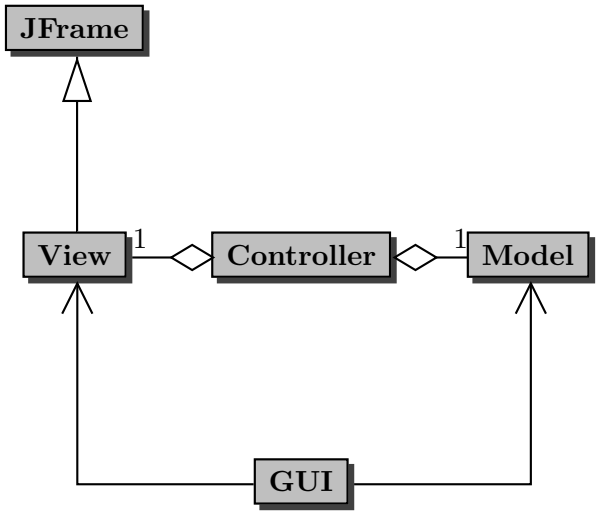


Since the `Controller` updates the `Model` and the `View`, it needs access to these objects. As a consequence, we will design the `Controller` class in such a way that it has a `Model` and it also has a `View`.

A GUI usually consists of a window with a title and a border. Furthermore, the window may contain components such as a menu bar, buttons, etcetera. A screenshot of the GUI we will develop can be found below.



By extending the `JFrame` class, which is part of the package `javax.swing`, our `View` class inherits more than three hundred methods from `JFrame` and its superclasses and more than one hundred attributes of `JFrame` and its superclasses are part of the state of a `View`.



We will see more examples of aggregation and inheritance when we implement the classes `Model`, `View` and `Controller`.

6.2 The Constructor Sections

Before implementing the classes `Model`, `View` and `Controller`, we first focus on their constructors. As we already mentioned, in the `GUI` app we create instances of the `Model` and `View` classes.

```
1 Model model = new Model(...);
2 View view = new View(...);
```

As we also mentioned, an instance of the `Controller` class is created in (the constructor of) the `View` class.

```
1 public View(...)
2 {
3    Controller controller = new Controller(...);
4 }
```

Since the `Controller` has a `Model` and has a `View`, the `Controller` class has attributes of type `Model` and `View`.

```
1 private Model model;
2 private View view;
```

These attributes are initialized in the constructor of the `Controller` class.

```
1 public Controller(Model model, View view)
2 {
3    this.setModel(model);
4    this.setView(view);
5 }
```

Note that we have to provide two arguments to the constructor of the `Controller` class in line 3 of the constructor of the `View` class, one of type `Model` and one of type `View`. The former can be passed as an argument to the constructor of the `View` class, whereas the latter is the object on which the constructor is invoked. This leads to the following constructor of the `View` class.

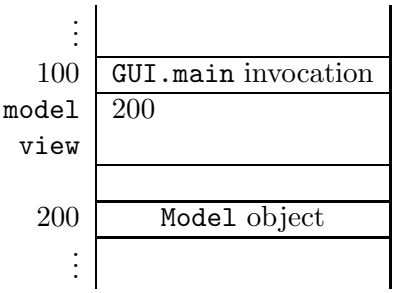
```
1 public View(Model model)
2 {
3    Controller controller = new Controller(model, this);
4 }
```

Now, we can return to the `GUI` app and fill in the arguments of the constructors.

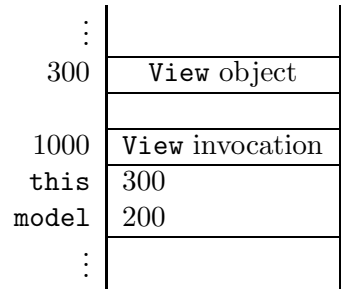
```
1 Model model = new Model();
2 View view = new View(model);
```

In summary, the `GUI` app first creates a `Model` and passes a reference to that `Model` to the constructor of the `View` class. In the constructor of the `View` class, that reference to the `Model` is passed on to the constructor of the `Controller` class together with a reference to the created `View`. Hence, the constructor of the `Controller` class receives references to the created `Model` and `View`.

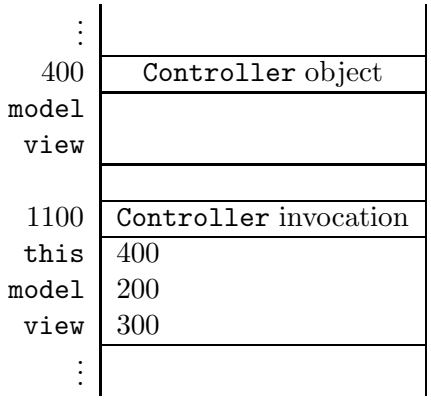
Once the execution reaches the end of line 1 of the `GUI` app, memory can be depicted as follows.



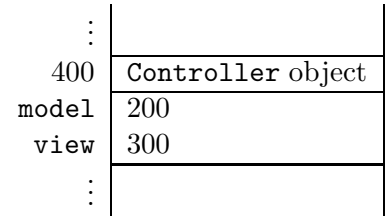
When executing line 2 of the `GUI` app, a memory block for the `View` object is allocated and the constructor of the `View` class is invoked.



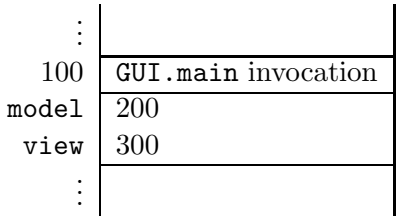
When executing the constructor of the `View` class, a memory block for the `Controller` object is allocated and the constructor of the `Controller` class is invoked.



Once we reach the end of the constructor of the `Controller` class, the `Controller` object can be depicted as follows.

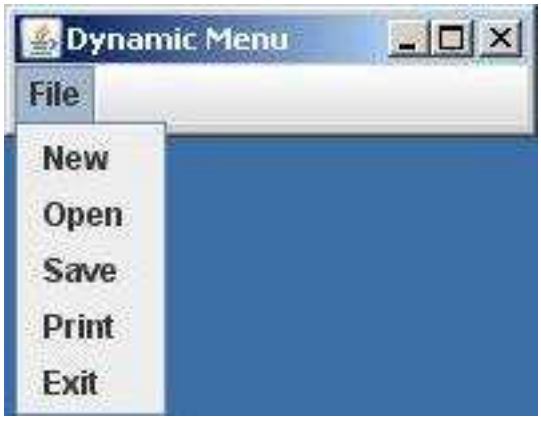


Once we reach the end of line 2 of the `GUI` app, its invocation block can be depicted as follows.



6.3 Dynamic Menu

We start with a rather simple GUI. The view consists of a menu bar. The menu bar has a single menu. The items of the menu are ordered in such a way that the most recently selected item is at the top of the menu. As a consequence, the order of the items may change as items are selected by the client. Assume that initially, when no items have been selected yet, the items are ordered as follows.



After the client has selected the Print item, the items are ordered as follows.

Below, we discuss the `Model`, `View`, and `Controller` classes and the `GUI` app.

6.3.1 The `Model` Class

The model contains the data of the GUI. In this case, we keep track of the order of the items of the menu. Furthermore, we provide the `Controller` with the method

```
public void select(String title)
```

to update the model when the item with the given `title` has been selected by the client. The method

```
public List<String> getTitles()
```

provides the `Controller` with the list of titles of the items of the menu. The complete API of the `Model` class can be found following this link.

The Attributes Section

To represent the order of the items of the menu, we use a list of the titles of the items. Hence, we introduce the following attribute.

```
private List<String> titles;
```

As class invariant, we introduce the following.

```
this.titles != null && this.titles contains no duplicates
```

The Constructors Section

The `Model` class only contains a one parameter constructor. In this constructor, we initialize the attribute `titles`.

```
public Model(List<String> titles)
{
   this.setTitles(titles);
}
```

The Methods Section

The accessor and mutator are implemented in the usual way. Note that only the accessor is public so that the `Controller` can invoke this method.

We have left to implement the `select` method. We have to move the given `title` to the beginning of the list. This can be accomplished by first removing the given `title` and by next inserting the given `title` at the beginning of the list.

```
public void select(String title)
{
   this.getTitles().remove(title);
   this.getTitles().add(0, title);
}
```

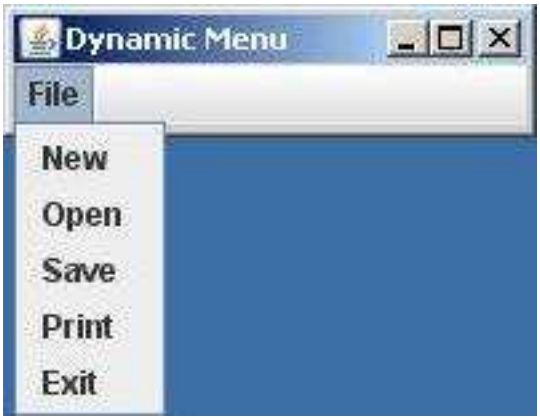
In the following snippet of client code, we create a model for our GUI.

```
List<String> titles = new LinkedList<String>();
titles.add("New");
titles.add("Open");
titles.add("Save");
titles.add("Print");
titles.add("Exit");
Model model = new Model(titles);
```

Once we have implemented the `Model` class, we can test its constructor and methods in the usual way. The code of the `Model` class can be found by following this link.

6.3.2 The `View` Class

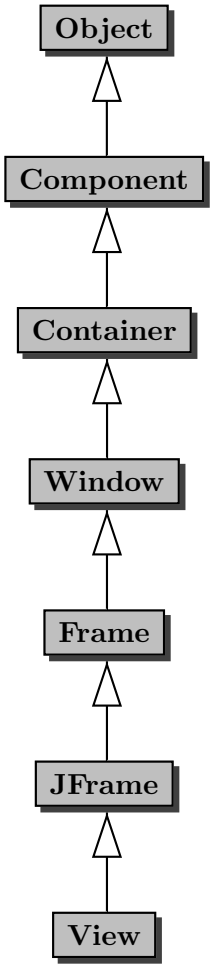
The view provides the graphical representation of the GUI. In this case, the GUI consists of a window with the title "Dynamic Menu" and a menu bar. The menu bar has a single menu, entitled "File". This menu has five items as shown below.



As we already mentioned in the introductory section of this chapter, the `View` extends `JFrame`. This is reflected in the class header as follows.

```
1 public class View extends JFrame
```

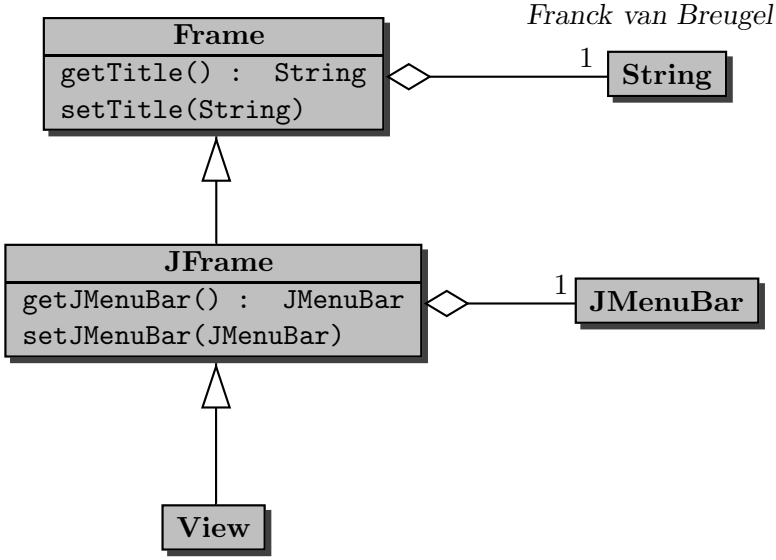
The corresponding inheritance hierarchy can be depicted as follows.



The Attributes Section

Our GUI has a title and a menu bar. However, the superclass `Frame` contains the attribute `title` of type `String` and the superclass `JFrame` contains an attribute `menuBar` of type `JMenuBar`, the accessor `getJMenuBar` and the mutator `setJMenuBar`.[1] Hence, the attributes `title` and `menuBar` are already part of the state of the `View` and the methods `getJMenuBar` and `setJMenuBar` are inherited by `View`. Therefore, we do not need to introduce any additional attributes.

[1]To be precise, the `JFrame` class contains the attribute `rootPane` of type `JRootPane` and the `JRootPane` class contains the attribute `menuBar` of type `JMenuBar`.

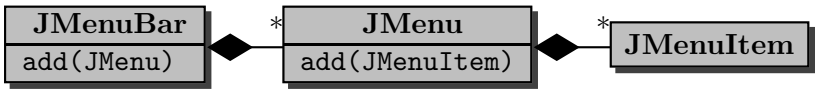


The Constructors Section

As we have already seen in Section 6.2, the constructor of the `View` class has the following skeleton.

```
1 public View(Model model)
2 {
3    Controller controller = new Controller(this, model);
4 }
```

Since our `View` class extends the `JFrame` class, we start the constructor by delegating to a constructor of the `JFrame` class. We use `super("Dynamic Menu")` to set the title of the frame to "Dynamic Menu." We have left to create the menu bar. A menu bar has a collection of menus and each menu has a collection of items.



We create a menu bar as follows.

```
1 JMenuBar bar = new JMenuBar();
```

We use the mutator `setJMenuBar` to initialize the menu bar of the `View`. We create a menu entitled "File" as follows.

```
1 JMenu menu = new JMenu("File");
```

The menu can be added to the menu bar as follows.

```
1 bar.add(menu);
```

We create a menu item entitled "New" as follows.

```
1 JMenuItem item = new JMenuItem("New");
```

The item can be added to the menu as follows.

```
1 menu.add(item);
```

In our implementation of the constructor, we delegate to the method

```
public void setMenu(List<String> titles)
```

which ensures that the first menu of the menu bar has items with the given `titles` in the corresponding order. The titles of the items of the menu can be obtained using `model.getTitles()`. Combining the above, we arrive at the following implementation of the constructor.

```
1 public View(Model model)
2 {
3    super("Dynamic Menu");
4    Controller controller = new Controller(this, model);
5    JMenuBar bar = new JMenuBar();
6    this.setJMenuBar(bar);
7    bar.add(new JMenu("File"));
8    this.setMenu(model.getTitles());
9 }
```

The Methods Section

Recall that the `setMenu` method ensures that the first menu of the menu bar has items with the given list of titles in the corresponding order. This method can implemented as follows.

```
1  public void setMenu(List<String> titles)
2  {
3     JMenu menu = this.getJMenuBar().getMenu(0);
4     menu.removeAll();
5     for (String title : titles)
6     {
7        JMenuItem item = new JMenuItem(title);
8        menu.add(item);
9     }
10 }
```

If we want to have a look at our `View`, we have to comment out line 4 of the constructor, since we have not yet implemented the `Controller`. The following snippet of client code creates a `Model` and a `View`.

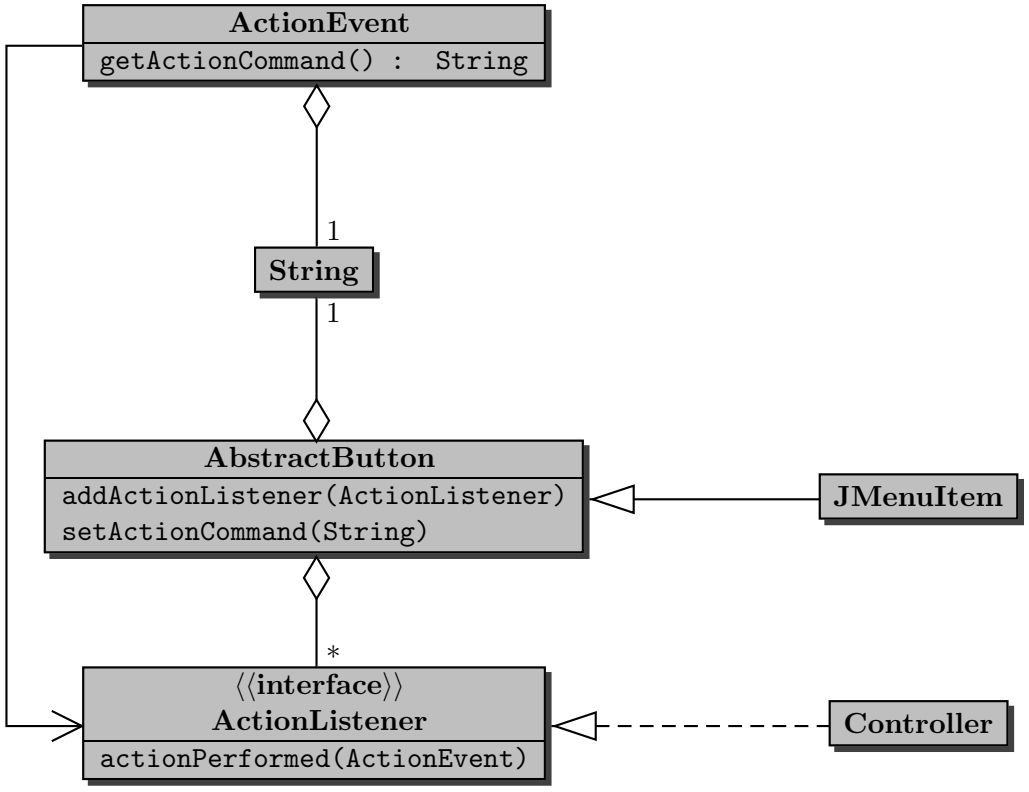
```
1 List<String> titles = new LinkedList<String>();
2 titles.add("New");
3 titles.add("Open");
4 titles.add("Save");
5 titles.add("Print");
```

```
6  titles.add("Exit");
7  Model model = new Model(titles);
8  View view = new View(model);
9  final int WIDTH = 200;
10 final int HEIGHT = 50;
11 view.setSize(WIDTH, HEIGHT);
12 view.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13 view.setVisible(true);
```

In line 11, the width and height (in the number of pixels) of the GUI is set using the method `setSize`, which is inherited from the `Window` class. In line 12, the behaviour of the GUI is defined when the box with the cross at the right upper corner is clicked. The method `setDefaultCloseOperation` is inherited from the `JFrame` class. The constant `JFrame.EXIT_ON_CLOSE` is used to specify that the GUI exits when the box with the cross at the right upper corner is clicked. The method `setVisible`, inherited from the `Window` class, makes the GUI visible.

6.3.3 Event-Driven Programming

Recall that the `Controller` translates the client's interactions, with the `View` into actions that manipulate the `View` and the `Model`. For our GUI, the client can interact with the `View` by selecting one of the items of the menu. The following class diagram provides an overview of the classes and methods that play a role.



Of the above classes we only have to implement the `Controller` class. The other classes are part of the Java standard library. We discuss the implementation of the `Controller` class in the next section. Below we discuss the additions that have to be made to the `View` class to signal the interactions of the client with the `View` to the `Controller`

Whenever the client interacts with the view (by, for example, selecting a menu item), the operating system and the Java virtual machine ensure that the `actionPerformed` method of the action listeners of the components (for example, of the menu item) are invoked. In our GUI, whenever the client selects a menu item, the operating system and the Java virtual machine ensure that the `actionPerformed` method of the `Controller` is invoked. From the above diagram we can conclude that an `AbstractButton` has `ActionListener`s and that a `JMenuItem` is an `AbstractButton` and, hence, also has `ActionListener`s. Furthermore, we can also deduce from the above diagram that the `Controller` class implements the `ActionListener` interface and, hence, a `Controller` is an `ActionListener`.

According to the above diagram, an `AbstractButton` (and, hence, a `JMenuItem`) and an `ActionEvent` have an action command, which is of type `String`. As we will see, this action command provides a link between the `Controller` and the `View`.

As we already mentioned above, whenever the client interacts with the view (by, for example, selecting a menu item), the operating system and the Java virtual machine ensure that the `actionPerformed` method of the action listeners of the components (for example, of the selected menu item) are invoked. Each invocation of the `actionPerformed` method is provided with a single argument of type `ActionEvent`. The Java virtual machine ensures that the action command of this `ActionEvent` is the same as the action command of the component (for example, of the menu item). In our GUI, whenever the client selects a menu item, the operating system and the Java virtual machine ensure that the `actionPerformed` method of the `Controller` is invoked with an `ActionEvent`, whose action command is the action command of the menu item, as its argument.

Within the `View` class, we have to

- set the action command of each menu item, and
- add the `Controller` as an `ActionListener` of each menu item.

In our GUI, we use the title of each menu item as its action command. We set the action command of each menu item using the mutator `setActionCommand`. Hence, we add to the `setMenu` method the following.

```
1 item.setActionCommand(title);
```

In the `setMenu` method we also add an `ActionListener` to each menu item, by adding a parameter named `actionListener` of type `ActionListener` to the signature of the `setMenu` method and by adding the following to the body of this method.

```
1 item.addActionListener(actionListener);
```

Hence, we arrive at the following `setMenu` method.

```
1 public void setMenu(List<String> titles, ActionListener actionListener)
2 {
```

```
3     JMenu menu = this.getJMenuBar().getMenu(0);
4     menu.removeAll();
5     for (String title : titles)
6     {
7        JMenuItem item = new JMenuItem(title);
8        item.setActionCommand(title);
9        item.addActionListener(actionListener);
10       menu.add(item);
11    }
12 }
```

Recall that we invoke the `setMenu` method in the constructor of the `View` class. Since we have changed the signature of the `setMenu` method, by adding a parameter of type `ActionListener`, we also have to modify the invocation of the `setMenu` method. We replace line 8 of the constructor with

```
1 this.setMenu(model.getTitles(), controller);
```

The API of the `View` class can be found by following this link and the code of the `View` class can be found by following this link.

6.3.4 The `Controller` Class

As we already saw in the above class diagram, the `Controller` class implements the `ActionListener` interface (so that we can use the `Controller` as the action listener of each menu item). This is reflected by the following class header.

```
1 public class Controller implements ActionListener
```

In Section 6.2 we already discussed the attributes and the constructor of the `Controller` class. We have left to introduce the methods of this class. Apart from the usual accessors and mutators, the `Controller` class contains a single method, namely the single method of the `ActionListener` interface: `actionPerformed`.

Whenever the client selects a menu item, the `Controller` has to modify the `Model` and `View`. In particular, the `Controller` has to invoke the `select` method on the `Model` with the title of the selected item as its argument. Furthermore, the `Controller` has to invoke the `setMenu` method on the `View`. The latter invocation takes two arguments: the list of titles of the menu items and the action listener to be associated with the menu items. The former information can be obtained from the `Model` using the `getTitles` accessor. The latter is the `Controller` itself.

As we already mentioned above, the `actionPerformed` method takes an `ActionEvent` object as its single argument. For our GUI, this `ActionEvent` contains the title of the menu item that has been selected as its action command. Hence, the `actionPerformed` method can be implemented as follows.

```
1 public void actionPerformed(ActionEvent event)
2 {
```

```
3     this.getModel().select(event.getActionCommand());
4     this.getView().setMenu(this.getModel().getTitles(), this);
5  }
```

The API of the `Controller` class can be found by following this link and the code of the `Controller` class can be found by following this link.

Next, we show how the `actionPerformed` method of the `Controller` class can be invoked and, hence, be tested in an app. As we have seen above, the `actionPerformed` method takes an `ActionEvent` as its single argument. To invoke the `actionPerformed` method, we need to create an `ActionEvent` object. According to the API of the `ActionEvent` class, its simplest constructor takes the following three arguments:

- `source` – the component that originated the event;
- `id` – an integer that identifies the event;
- `command` – a string that specifies the action command associated with the event.

Since we create the `ActionEvent` in the app, it has no originating component (such as a menu item). However, we cannot use null for the `source`, because the constructor throws an exception in that case. Hence, we provide a generic `Object` as the first argument. The API provides a number of constants that can be used for the second argument. We use the constant `ActionEvent.ACTION_PERFORMED` which indicates that a meaningful action occured. The third and final argument is the action command. For our GUI, the action command is the title of one of the menu items. Hence, we can create an `ActionEvent` object as follows.

```
1  new ActionEvent(new Object(), ActionEvent.ACTION_PERFORMED, title)
```

In the app below, we create a `Model`, `View` and `Controller` in the usual way. Then we print the list of titles, randomly select a title, and print the selected title. Now we can invoke the `actionPerformed` method, mimicking that the item with the given title has been selected. Finally, we print the list of titles so that we can check if the invocation of the `actionPerformed` method has the desired effect.

```
1  List<String> titles = new LinkedList<String>();
2  titles.add("New");
3  titles.add("Open");
4  titles.add("Save");
5  titles.add("Print");
6  titles.add("Exit");
7  Model model = new Model(titles);
8  View view = new View(model);
9  Controller controller = new Controller(view, model);
10 output.println(model.getTitles());
11 Random random = new Random();
12 String title = titles.get(random.nextInt(titles.size()));
13 output.println("Selected title: " + title);
```

```
14 controller.actionPerformed(new ActionEvent(new Object(), ActionEvent.
       ACTION_PERFORMED, title));
15 output.println(model.getTitles());
```

6.3.5 The `GUI` App

We have already discussed all the ingredients of the GUI app. Here, we simply show the body of its `main` method.

```
1  List<String> titles = new LinkedList<String>();
2  titles.add("New");
3  titles.add("Open");
4  titles.add("Save");
5  titles.add("Print");
6  titles.add("Exit");
7  Model model = new Model(titles);
8  View view = new View(model);
9  final int WIDTH = 200;
10 final int HEIGHT = 50;
11 view.setSize(WIDTH, HEIGHT);
12 view.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13 view.setVisible(true);
```

The code of the `GUI` app can be found by following this link.

6.4 Shuffling the Items

We add a button to our GUI. The button is emtitled "Shuffle."

When the client presses the button, the items of the menu are shuffled. For example, assume that the menu items are ordered as follows.

After the client has pressed the button, the order of the items may have changed as follows.

Next, we discuss the modifications to the classes `Model`, `View` and `Controller`.

6.4.1 The `Model` Class

Recall that the `Model` has a list of titles. When the client presses the button, the `Controller` should shuffle this list. Hence, we add the method

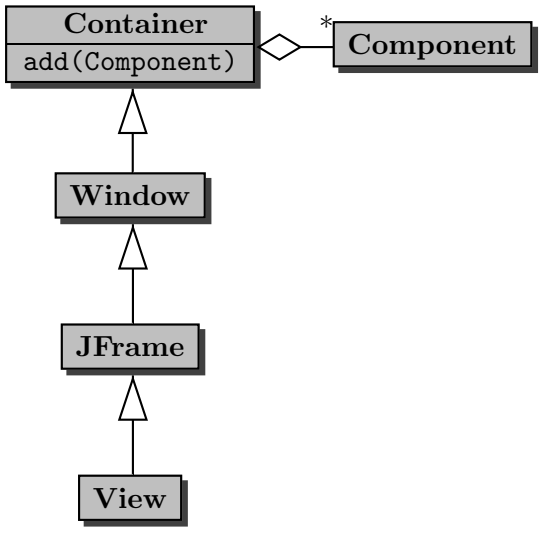
```
public void shuffle()
```

to shuffle the list of titles. To implement this method, we delegate to the utility `Collections`, which is part of the package `java.util`, as follows.

```
1 public void shuffle()
2 {
3    Collections.shuffle(this.getTitles());
4 }
```

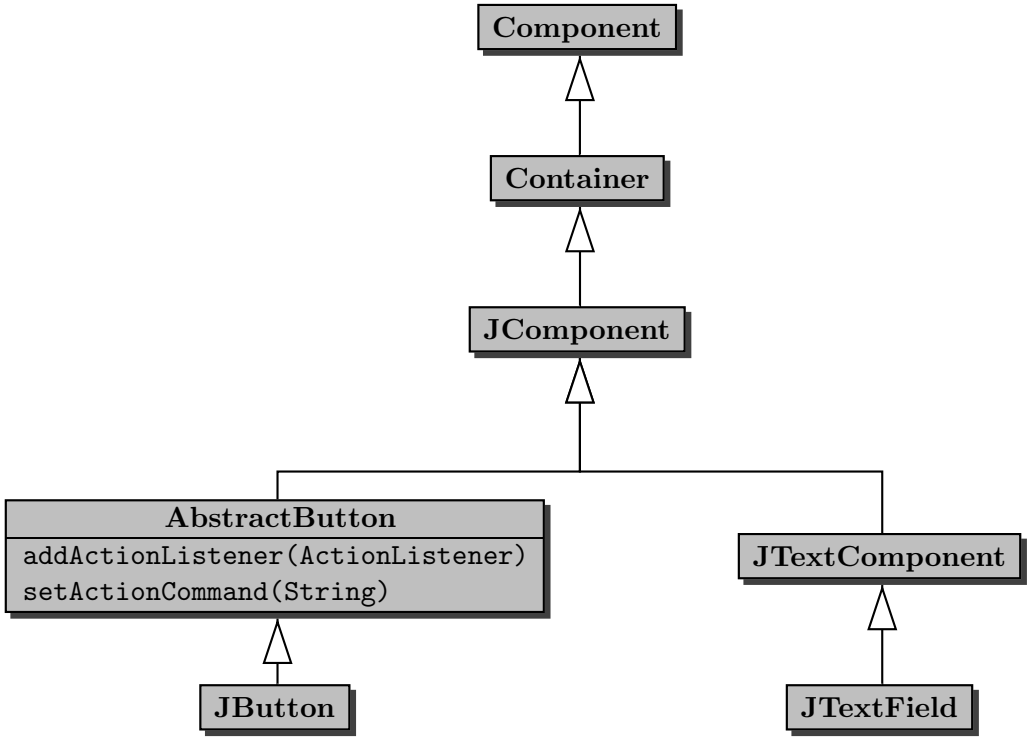
6.4.2 The `View` Class

The modified `View` has a button. This button can be represented by a `JButton` object. We may be tempted to introduce an attribute of type `JButton`. However, consider the following class diagram.



Note that a `Container` has a collection of `Component`s. This collection contains the components (such as buttons and text fields) that are part of the container. Note that our `View` is a `Container`. As a consequence, the collection of `Component`s is part of the state of the `View`.

Also consider the following class diagram.



Note that a `JButton` is a `Component` and, hence, can be added to the above mentioned collection of `Components`. Hence, rather than adding an attribute of type `JButton` to our `View`, we create a `JButton` object and add it to the collection of `Components`, which is part of the state of our `View`, using the `add` method inherited by our `View` class from the `Container` class.

As we did before for the menu items, we have to

- set the action command of the button, and
- add the `Controller` as an `ActionListener` of the button.

Recall that the action command is shared by the `Component` (the `JButton` in this case), which is part of the `View`, and the `ActionEvent`, which is provided as an argument to the `actionPerformed` method of the `Controller`. Hence, the action command is shared by the `View` and the `Controller`. Rather than arbitrarily placing it in either the `View` class or the `Controller` class, we introduce a new class `ActionCommands` that contains the action command.

```
1 public class ActionCommands
2 {
3    public static final String SHUFFLE = "Shuffle";
4 }
```

To the constructor of the `View` class we add the following.

```
1 JButton button = new JButton("Shuffle");
2 this.add(button);
3 button.addActionListener(controller);
```

```
4 button.setActionCommand(ActionCommands.SHUFFLE);
```

6.4.3 The `Controller` Class

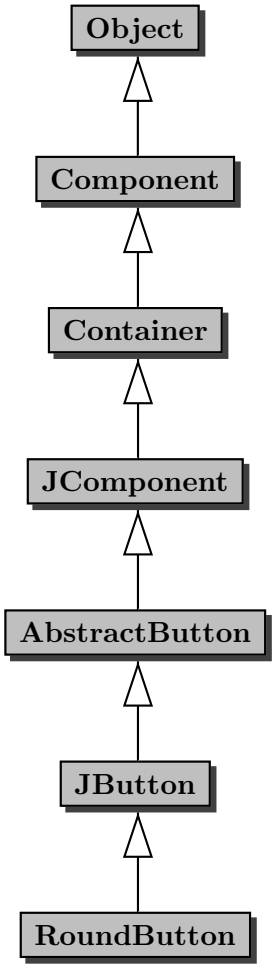
As we already mentioned above, whenever the client presses the button, the `Controller` should invoke the `shuffle` method on the `Model`. Hence, we modify the `actionPerformed` method to the following.

```
1  public void actionPerformed(ActionEvent event)
2  {
3     String action = event.getActionCommand();
4     if (action.equals(ActionCommands.SHUFFLE))
5     {
6        this.getModel().shuffle();
7     }
8     else
9     {
10       this.getModel().select(action);
11    }
12    this.getView().setMenu(this.getModel().getTitles(), this);
13 }
```

6.5 Beyond the Basics

6.5.1 A Round Button

We replace the rectangular button to shuffle the menu items with a round button. Rather than developing a `RoundButton` class from scratch, we extend the `JButton` class.



Our `RoundButton` class inherits more than four hundred methods from `JButton` and its superclasses and more than one hundred attributes of `JButton` and its superclasses are part of the state of a `RoundButton`. To express that the `RoundButton` class is a subclass of the `JButton` class, we use the following class header.

```
1 public class RoundButton extends JButton
```

In our `RoundButton` class, we only need to provide a constructor (since constructors are not inherited) and override the methods `paintBorder` and `contains`. The former method paints the button's border. Since the border of a `RoundButton` is round, whereas the border of a `JButton` is rectangular, the method needs to be overridden. The latter method defines the shape of the button. Because a `RoundButton` is round, whereas a `JButton` is rectangular, also this method needs to be overridden.

The constructor of our `RoundButton` class takes the title of the button as its single argument. Since the `RoundButton` class extends the `JButton` class, we first delegate to a constructor of the superclass to initialize the state. Furthermore, we have to set the `contentAreaFilled` attribute to false so that the area of the `JButton`, which is rectangular, is not filled.

```
1 public RoundButton(String title)
```

```
{
   super(title);
   this.setContentAreaFilled(false);
}
```

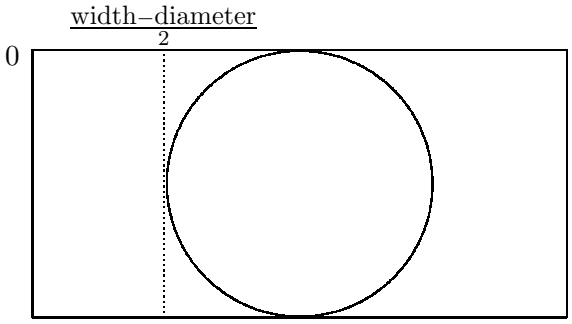
The method `paintBorder` paints the border of the button on the given `Graphics` object. This `Graphics` object is passed to the method by the Java virtual machine. Since our `RoundButton` is a `JButton` and a `JButton` has a width and a height, we take the diameter of the button to be the minimum of that width and height.

To draw a circle, we use the method

```
public void drawOval(int x, int y, int width, int height)
```

where `x` and `y` are the x- and y-coordinate of the left upper corner of the oval to be drawn and `width` and `height` are width and height of the oval to be drawn.

If the width is greater than the height, then the left upper corner has coordinates $(\frac{\text{width}-\text{diameter}}{2}, 0)$.



Otherwise, the left upper corner has coordinates $(0, \frac{\text{height}-\text{diameter}}{2})$. Both cases can be unified into one: the left upper corner has coordinates $(\frac{\text{width}-\text{diameter}}{2}, \frac{\text{height}-\text{diameter}}{2})$. Note that if the width is greater than the height, then the diameter is equal to the height and, hence, $\frac{\text{height}-\text{diameter}}{2}$ equals 0.

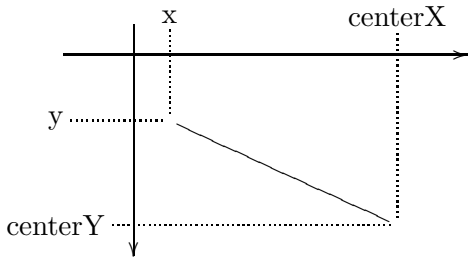
Since we want to draw a circle, we use the diameter of the circle as the width and height of the oval.

Hence, the `paintBorder` method can be implemented as follows.

```
public void paintBorder(Graphics canvas)
{
   int diameter = Math.min(this.getWidth(), this.getHeight());
   canvas.drawOval((this.getWidth() - diameter) / 2, (this.getHeight() -
       diameter) / 2, diameter, diameter);
}
```

The method `contains` checks whether the point specified by the given x- and y-coordinate falls within the circle that outlines the round button. The operating system and the Java virtual machine provide the x- and y-coordinate to the `contains` method.

The point (x, y) is within the circle if and only if the distance from (x, y) to the centre of the circle is smaller than or equals to the radius of the circle (the radius is half the diameter). Let (centerX, centerY) be the coordinates of the centre of the circle.



The distance from (x, y) to (centerX, centerY) is

$$\sqrt{(\mathrm{x} - \mathrm{centerX})^2 + (\mathrm{y} - \mathrm{centerY})^2}$$

Hence, the `contains` method can be implemented as follows.

```
1 public boolean contains(int x, int y)
2 {
3    int diameter = Math.min(this.getWidth(), this.getHeight());
4    double centerX = this.getWidth() / 2.0;
5    double centerY = this.getHeight() / 2.0;
6    return Math.sqrt(Math.pow(centerX - x, 2.0) + Math.pow(centerY - y, 2.0))
        <= diameter / 2.0;
7 }
```

Chapter 7

Recursion

7.1 Introduction

Let us implement the method

```
/**
 * Prints the given number of *s.
 *
 * @param number of *s to be printed.
 * @pre. n >= 0
 */
public static void stars(int n)
```

of the class `Print`. An obvious way to implement this method is the following.

```
public static void stars(int n)
{
   for (int i = 0; i < n; i++)
   {
      System.out.print('*');
   }
}
```

However, there are other ways to implement this method. Obviously, if `n` is zero, then we do not have to print anything at all. Otherwise, we have to print at least one `*`. This leads us to the following skeleton.

```
public static void stars(int n)
{
   if (n == 0)
   {
      // do nothing
   }
```

```
7     else
8     {
9        System.out.print('*');
10
11    }
12 }
```

What remains to be done at line 10 is printing `n` $-$ 1 `*`s. In order to accomplish that, we can delegate to the `stars` method:

```
10 Print.stars(n - 1);
```

As usual, when we invoke a method, such as `stars`, we have to make sure that its precondition is satisfied. If we reach line 10, then we have that `n` > 0 and, hence, `n` $- 1 \geq 0$. So the precondition of `stars` is indeed satisfied at line 10.

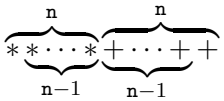
Note that the `stars` method is invoked within the body of the `stars` method. This makes it a so-called *recursive* method. The invocations of the method within itself are known as *recursive invocations*. For example, the invocation `Print.stars(n - 1)` on line 10 is a recursive invocation.

Within the body of a recursive method we find one or more *base cases* and one or more *recursive cases*. In a base case, there is no need to delegate to the method itself, whereas a recursive case contains one or more recursive invocations. The above recursive method `stars` has a single base case, namely when `n` is zero. In that case, line 5 is executed. The method also has a single recursive case, namely when `n` is greater than zero. In that case, line 9 and 10 are executed.

The method `starsAndPlusses` prints a number of `*`s —this number is given as an argument— followed by the same number of `+`s.

```
1  /**
2   * Prints the given number of *s followed by the given number of +s.
3   *
4   * @param the number of *s and +s to be printed.
5   * @pre. n >= 0
6  */
7  public static void starsAndPlusses(int n)
```

When implementing a method recursively, we first determine base cases, i.e. those cases in which we do not need to delegate to the method itself. If `n` is zero, we do not have to do anything. This is an example of a base case. Now assume that `n` is greater than zero. In that case we have to print the following.



Note that the substring starting from the second `*` and ending with the one but last `+` can be printed by delegating to `Print.starsAndPlusses(n - 1)`. Hence, to print `n` `*`s followed by `n` `+`s we can

1. print a `*`,
2. print `n` − 1 `*`s followed by `n` − 1 `+`s by delegating to `Print.starsAndPlusses(n - 1)`, and
3. print a `+`.

Combining the above leads us to the following.

```
1  public static void starsAndPlusses(int n)
2  {
3     if (n == 0)
4     {
5        // do nothing
6     }
7     else
8     {
9        System.out.print('*');
10       Print.starsAndPlusses(n - 1);
11       System.out.print('+');
12    }
13 }
```

As in the previous example, we have to make sure that the precondition of the method `starsAndPlusses` is satisfied when we invoke it in line 10. If we reach line 10, then we have that $n > 0$ and, hence, $n - 1 \geq 0$. So the precondition is satisfied. The method has a single base case. In that case line 5 is executed. The method also has a single recursive case. In that case line 9–11 are executed.

To simplify matters a little, all methods in this chapter are static. Note, however, that recursion can also be used to implement non-static methods.

In the remainder of this chapter, we will provide numerous examples of recursive methods. We will start with simple examples, such as the `stars` method presented above, to illustrate recursion. To implement these methods, using iteration, i.e. using for-, while- and do-loops, may be simpler than using recursion. However, when we come to more intricate examples, the strength of recursion will become apparent.[1]

7.2 Proving Correctness and Termination

First, we present a strategy to prove recursive methods correct and a technique to prove that recursive methods terminate. Subsequently, we apply these to prove the correctness and termination of the recursive methods introduced in the previous section.

The correctness proof of a recursive method is split into two parts.

[1] If a method can be implemented using recursion, then it can also be implemented using iteration. Conversely, if a method can be implemented using iteration, then it can also be implemented using recursion. So, what should we use, recursion or iteration? Generally, we choose the technique that gives rise to the simplest code. We will violate this guideline at the beginning of this chapter, since we want to illustrate recursion by means of simple examples and we are not aware of any simple examples for which recursion gives rise to simpler code than iteration.

- For each base case, we prove that it is correct.
- For each recursive case, we assume that the recursive invocations are correct and we prove under that assumption that the recursive case is correct as well.

The termination proof of a recursive method consists of two steps.

1. We define the size of each invocation of the method. The size has to be a natural number (i.e. a non-negative integer).
2. We prove that each recursive invocation has a smaller size than the original invocation.

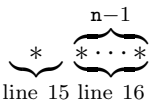
Our strategy to prove correctness can only be used in combination with a proof of termination using the above proof technique.[2]

The method `stars` of the class `Print` prints a number of `*`s.

```
1  /**
2   * Prints the given number of *s.
3   *
4   *  @param the number of *s to be printed.
5   *  @pre. n >= 0
6   */
7  public static void stars(int n)
8  {
9     if (n == 0)
10    {
11       // do nothing
12    }
13    else
14    {
15       System.out.print('*');
16       Print.stars(n - 1);
17    }
18 }
```

Let us first prove the method correct.

- In the base case (line 11), when `n` equals zero, we do nothing, which is equivalent to printing zero `*`s.
- Consider the recursive case (line 15 and 16). Assume that the recursive invocation `Print.stars(n - 1)` is correct, i.e. it prints `n - 1` `*`s. Then the recursive case (line 15 and 16) prints `n` `*`s.



[2] Our correctness proof and our termination proof can be combined into a proof that for each invocation of the method, the invocation is correct and terminates by strong induction on the size of the invocation.

Next, let us prove termination.

1. We define the size of the invocation `Print.stars(n)` by

   $$\text{size}(\texttt{Print.stars(n)}) = \texttt{n}.$$

   From the precondition we can conclude that the size of an invocation is a natural number.

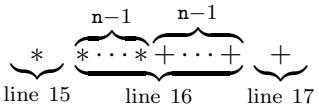
2. The size of the recursive invocation `Print.stars(n - 1)` is $\texttt{n} - 1$, which is smaller than `n`, the size of the original invocation `Print.stars(n)`.

The method `starsAndPlusses` prints a number of `*`s followed by the same number of `+`s.

```
1  /**
2   * Prints the given number of *s followed by the given number of +s.
3   *
4   * @param the number of *s and +s to be printed.
5   * @pre. n >= 0
6   */
7  public static void starsAndPlusses(int n)
8  {
9     if (n == 0)
10    {
11       // do nothing
12    }
13    else
14    {
15       System.out.print('*');
16       Print.starsAndPlusses(n - 1);
17       System.out.print('+');
18    }
19 }
```

Again, we first prove the correctness of the recursive method.

- In the base case (line 11), when `n` equals zero, we do nothing, which is equivalent to printing zero `*`s followed by zero `+`s.
- Consider the recursive case (line 15–17). Assume that the recursive invocation `Print.starsAndPlusses(n - 1)` is correct, i.e. it prints `n - 1` `*`s followed by `n - 1` `+`s. Then the recursive case (line 15–17) prints `n` `*`s followed by `n` `+`s.



Termination of this method can be proved in the same way as the previous method.

7.3 Recursive Methods that Return an Integer

Recall that the factorial of n, often denoted as $n!$, is defined as

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots (n-1) \times n.$$

Let us implement the method

```
1 /**
2  * Returns the factorial of the given number.
3  *
4  * @param a number whose factorial is returned.
5  * @pre. n >= 1
6  * @return n!
7  */
8 public static int factorial(int n)
```

of the `Math` class using recursion. Note that, in contrast to all recursive methods that we have developed so far, this method returns a result.

If `n` is one, then `n!` is one. This is a base case. Otherwise, `n` is greater than one. In that case,

$$\texttt{n}! = 1 \times 2 \times \cdots \times (\texttt{n}-1) \times \texttt{n} = (\texttt{n}-1)! \times \texttt{n}.$$

To compute $(\texttt{n}-1)!$ we can delegate to the method itself using `Math.factorial(n - 1)`. Hence, this is a recursive case. Combining the above, we arrive at the following recursive method.

```
1  public static int factorial(int n)
2  {
3     int factorial;
4     if (n == 1)
5     {
6        factorial = 1;
7     }
8     else
9     {
10       factorial = Math.factorial(n - 1) * n;
11    }
12    return factorial;
13 }
```

Note that if we arrive at line 10, then $\texttt{n} \geq 2$ and, hence, $\texttt{n} - 1 \geq 1$ and, hence, the precondition for the invocation `Math.factorial(n - 1)` is satisfied.

Let us prove the method `factorial` correct.

- In the base case (line 6), when `n` is one, we assign the variable `factorial` the value of `n!` and, hence, this case is correct.

- Consider the recursive case (line 10). Assume that the recursive invocation `Math.factorial(n - 1)` is correct, i.e. it returns (`n`−1)!. Then we assign (`n`−1)!×`n` = `n`! to the variable `factorial` in line 10. Hence, this case is correct as well.

Next, we show that the method terminates.

1. We define the size of the invocation `Math.factorial(n)` by

   $$\text{size}(\texttt{Math.factorial(n)}) = \texttt{n}.$$

   From the precondition we can conclude that the size of an invocation is a natural number.

2. The size of the recursive invocation `Math.factorial(n - 1)` is `n` − 1, which is smaller than `n`, the size of the original invocation `Math.factorial(n)`.

7.4 Recursive Methods for Lists

The method `contains` of the class `Collections` checks if a element is part of a list.

```
1  /**
2   * Tests if the given list contains the given element.
3   *
4   * @param element the element.
5   * @pre. element != null
6   * @param list the list.
7   * @pre. list != null
8   * @return true if the given list contains the given element, false otherwise.
9   */
10 public static <T> boolean contains(T element, List<T> list)
```

Obviously, an empty list, i.e. one of size zero, does not contain the given element. This is a base case. If the first element of the list is the element for which we are looking, then we have found it right away. This is a base case as well. Otherwise, the list is nonempty and the first element is not the one for which we are looking. Hence, we have to look in the rest of the list for the element. This can be done be delegating and, hence, this is a recursive case.

To obtain the rest of the list (without destroying the original list), we can make a shallow copy as follows.

```
List<T> rest = new ArrayList<T>(list.subList(1, list.size()));
```

The method `subList` returns a view of a portion of the list. In the above case, it returns a view of the portion starting at index 1 and ending just before index `list.size()`, that is, all but the first element (the element at index 0) of the list. We use the copy constructor of the `ArrayList` class to copy this portion.

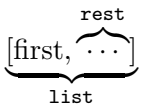
We can implement the `contains` method as follows.

```
1  public static <T> boolean contains(T element, List<T> list)
2  {
3     boolean contains;
4     if (list.size() == 0)
5     {
6        contains = false;
7     }
8     else
9     {
10       if (element.equals(list.get(0))
11       {
12          contains = true;
13       }
14       else
15       {
16          List<T> rest = new ArrayList<T>(list.subList(1, list.size()));
17          contains = Collections.contains(element, rest);
18       }
19    }
20    return contains;
21 }
```

In line 17, we have that `element != null` and `rest != null` and, hence, the precondition of `contains` is satisfied.

The correctness of the `contains` method can be proved as follows.

- In the first base case (line 6), `list` is empty. Obviously, an empty list does not contain `element` and, hence, setting `contains` to false is correct.
- In the second base case (line 12), `list` is nonempty and the first element of `list` is equal to `element`. In this case we have found the element and, hence, setting `contains` to true is correct.
- When we reach the recursive case (line 16–17), we know that `list` is nonempty and the first element of `list` is not equal to `element`. We assume that the recursive invocation `Collections.contains(element, rest)` is correct, i.e. it returns whether `element` is contained in `rest`.

  Since the first element of `list` is not equal to `element`, `list` contains `element` if and only if `rest` contains `element`. Therefore,

```
contains = Collections.contains(element, rest);
```

is correct.

We can prove that the `contains` method terminates as follows.

1. We define the size of the invocation `Collections.contains(element, list)` by

   $$\text{size}(\texttt{Collections.contains}(\texttt{element}, \texttt{list})) = \texttt{list.size}().$$

   Obviously, `list.size()` is a natural number.

2. Since `rest` is obtained by removing the first element of `list`, `rest.size()` is smaller than `list.size()`. Hence, the size of the recursive invocation `Collections.contains(element, rest)` is smaller than the size of the original invocation `Collections.contains(element, list)`.

The method `isConstant` of the class `Collections` checks if a list is constant, i.e. if all its elements are equal.

```
/**
 * Tests if the given list is constant, that is, if all elements are the same.
 *
 * @param list the list.
 * @pre. list != null and list does not contain null
 * @return true if the given list is constant, false otherwise.
 */
public static <T> boolean isConstant(List<T> list)
```

If the list is empty, then all elements are equal and, hence, the list is constant. If the list contains a single element, then the list is constant as well. Both are base cases. If the first and the second element are not equal, then the list is not constant. This is yet another base case. Otherwise, the list contains at least two elements and the first and second element are equal. In that case, the list is constant if and only if the sublist starting with the second element is constant. The latter fact can be determined by invoking the method recursively on the rest of the sublist. Therefore, the `isConstant` method can be implemented as follows.

```
public static <T> boolean isConstant(List<T> list)
{
   boolean constant;
   if (list.size() <= 1)
   {
      constant = true;
   }
   else
   {
```

```
10      if (!list.get(0).equals(list.get(1)))
11      {
12         constant = false;
13      }
14      else
15      {
16         List<T> rest = new ArrayList<T>(list.subList(1, list.size()));
17         constant = Collections.isConstant(rest);
18      }
19   }
20   return constant;
21 }
```

We leave it to the reader to check that the precondition of `isConstant` is satisfied at line 17.

Let us prove the `isConstant` method correct.

- In the first base case (line 6), the list is either empty or contains a single element. In both cases all elements are equal and, hence, the assignment `constant = true` is correct.
- In the second base case (line 12), the first and second element are not equal and, therefore, the list is not constant, so the assignment `constant = false` is correct.
- Let us consider the recursive case (line 16–17). Assume that the recursive call is correct, i.e. it returns whether `rest` is constant. If we reach the recursive case, we know that the list has at least two elements and the first element is equal to the second element. Hence, `list` is constant only if `rest` is constant. Therefore, the assignment `constant = Collections.isConstant(rest)` is correct.

We can prove that the method `isConstant` terminates in the same way as we proved the termination of the `contains` method.

The `minimum` method of the `Collections` class returns the minimum of a nonempty list of integers.

```
1  /**
2   * Returns the minimum element of the given list.
3   *
4   * @param list the list.
5   * @pre. list != null and list.size() > 0 and list does not contain null
6   * @return the minimum element of the given list.
7   */
8  public static int minimum(List<Integer> list)
```

If the list contains a single integer, then obviously that integer is the minimum. This is a base case. Assume that the list contains more than one element. Then the minimum is the minimum of

- the first element,

- the minimum of the rest of the list.

The latter can be obtained by a recursive call.

```
1  public static int minimum(List<Integer> list)
2  {
3    int minimum;
4    if (list.size() == 1)
5    {
6       minimum = list.get(0);
7    }
8    else
9    {
10      List<Integer> rest = new ArrayList<Integer>(list.subList(1, list.size()))
            ;
11      minimum = Math.min(list.get(0), Collections.minimum(rest));
12   }
13   return minimum;
14 }
```

Note that `rest != null`. Since `list` does not contain null, `rest` does not either. If we reach line 10, `list.size()` is greater than one and, hence, `rest.size()` is greater than zero. Therefore, precondition for the invocation of `minimum` in line 10 is satisfied.

The correctness of the `minimum` method can be proved as follows.

- In the base case (line 6), the list has a single element, which is the minimum.
- For the recursive case (line 10–11), assume that the recursive invocation `Collections.minimum(rest)` is correct, i.e. it returns the minimum of `rest`. As we already mentioned above, in this case the minimum of `list` is the minimum of the first element and the minimum of the rest of the list.

The proof of termination is similar to the one presented above.

In all the recursive methods for lists presented above, the method is recursively invoked on a copy of the list with the first element removed. Rather than removing the first element, we can also reduce the size of the list in other ways. For example, we can split a list in two as follows.

```
int middle = list.size() / 2;
List<T> left = new ArrayList<T>(list.subList(0, middle));
List<T> right = new ArrayList<T>(list.subList(middle, list.size()));
```

As an example, let us consider the `minimum` method again. In the recursive case we can split `list` into `left` and `right`. Since the minimum element of `list` is the minimum of

- the minimum element of `left` and
- the minimum element of `right`

we can implement the `minimum` method also as follows.

```
1  public static int minimum(List<Integer> list)
2  {
3    int minimum;
4    if (list.size() == 1)
5    {
6       minimum = list.get(0);
7    }
8    else
9    {
10      int middle = list.size() / 2;
11      List<Integer> left = new ArrayList<Integer>(list.subList(0, middle));
12      List<Integer> right = new ArrayList<Integer>(list.subList(middle, list.
           size()));
13      minimum = Math.min(Collections.minimum(left), Collections.minimum(right))
           ;
14   }
15   return minimum;
16 }
```

Note that, since `list.size()` is greater than one when we reach line 10, the lists `left` and `right` are nonempty. This is needed to verify that the preconditions for `Collections.minimum(left)` and `Collections.minimum(right)` are satisfied. The details are left to the reader.

To modify the above correctness proof, we only have to revisit the recursive case (line 10–13). In that case, we assume that the recursive invocations `Collections.minimum(left)` and `Collections.minimum(right)` are correct, i.e. return the minimum element of the left and the right part of `list`, respectively. As we already mentioned above, the minimum element of `list` is the minimum of the minimum element of `left` and the minimum element of `right`. As a consequence, the assignment of line 13 is correct.

Termination can be proved as follows.

1. We define the size of the invocation `minimum(list)` by

   $$\text{size}(\texttt{minimum}(\texttt{list})) = \texttt{list.size}()$$

2. When we reach line 10 `list.size()` is greater than one. As a consequence, `left.size()` and `right.size()` are smaller than `list.size()`. Hence, the size of the recursive invocations `Collections.minimum(left)` and `Collections.minimum(right)` are smaller than the size of the original invocation `Collections.minimum(list)`.

Chapter 8

Arrays

8.1 Arrays

8.1.1 What is an Array?

To motivate why we might be interested in using arrays, let us implement an app that creates a collection of doubles. We will keep track of the number of milliseconds it takes to create the collection. In the snippet below we use a `LinkedList` to represent the collection.

```
1 output.print("Provide the size of the collection: ");
2 final int SIZE = input.nextInt();
3 long start = System.currentTimeMillis();
4 List<Double> collection = new LinkedList<Double>();
5 for (int i = 0; i < SIZE; i++)
6 {
7    collection.add(new Double(1.0));
8 }
9 output.println(System.currentTimeMillis() - start);
```

If we run the above snippet for different sizes of the collection, we get the following times (in milliseconds).

SIZE	time
100000	28
1000000	267
10000000	13348

Next, we do the same but this time we exploit arrays. We only have to changes line 4 and 7 of the above snippet. We replace them with the following.

```
4 double[] collection = new collection[SIZE];
```

```
7 collection[i] = 1.0;
```

We will discuss the above two lines in detail later in this section. For now, we are merely interested in the output that the modified app produces.

SIZE	time
100000	20
1000000	128
10000000	5035

Comparing the two tables, we can conclude that the app which uses the array takes less time than the one which uses the `LinkedList`.

Arrays are objects, but of a special type. Like other objects, we declare arrays and we create them (but in a special way). An array has a single attribute, which we will discuss later. In principle, we can invoke methods on arrays but we hardly ever do. Arrays also have some special features that ordinary objects do not possess.

As we have seen in the above example, we can exploit an array to implement a collection. We can declare an array of `double`s named `collection` as follows.

```
1 double[] collection;
```

The type of the variable `collection` is `double[]`. The `[]` denotes that we are dealing with an array. The `double` denotes that the elements stored in the array are of type `double`. The type `double` is known as the base type of the array.

To create an object of type `double[]`, we use the keyword `new`. At the time of creation, we have to fix the length of the array. For example,

```
1 collection = new double[SIZE];
```

creates an array of `double`s of length `SIZE` (and assigns it to the variable `collection`).

The attribute `length` contains the length of the array. This attribute is final which reflects that the length of an array is fixed. The snippet

```
1 output.println(collection.length);
```

prints the length of the array `collection`.

The elements of the array can be found in the so-called cells of the array. Each cell has an index. The index of the first cell is 0. The second one has index 1, etcetera. The index of the last cell of the array `collection` is `SIZE` $-$ 1. Initially, each array cell contains the default value of the base type of the array. Since the base type of the array `collection` is `double` and the default value for `double` is 0.0, each cell of `collection` contains 0.0 when the array is created.

The elements of an array can be accessed via their indices. For example, to print the first element and last element of the array `collection` we can use

```
1 output.println(collection[0]);
2 output.println(collection[SIZE - 1]);
```

If we provide an invalid index, that is, one that is smaller than 0 or one that is greater than or equal to the length of the array, then an `ArrayIndexOutOfBoundsException` is thrown. For example,

```
1 output.println(collection[-1]);
```

and

```
1 output.println(collection[SIZE]);
```

both give rise to such an exception.

To change the content of an array cell, we can assign it a new value. For example,

```
1 collection[0] = 1.0;
```

assigns 1.0 to the first cell of the array `collection`. Also in this case an `ArrayIndexOutOfBoundsException` is thrown if the used index is invalid.

In the above example, we have used an array the base type of which is a primitive type, namely `double`. However, the base type of an array can also be non-primitive. For example, we can declare and create an array of `Double`s as follows.

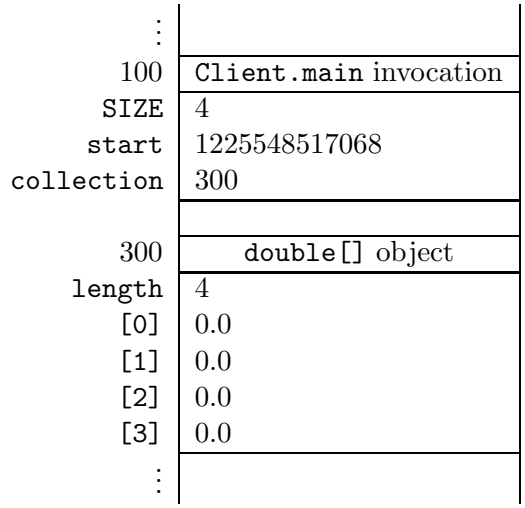
```
1 Double[] collection = new Double[SIZE];
```

Initially, each cell of this array contains the default value of the type `Double` which is null.

8.1.2 Memory Diagrams

Recall that arrays are objects. As a consequence, in our memory diagrams each array will have its own block. In that block we find its single attribute `length` and its value. Furthermore, the block also contains the cells of the array and their values.

For example, if the execution reaches the end of line 4 of the array version of the code snippet of the previous section and the user entered four, then memory can be depicted as follows.

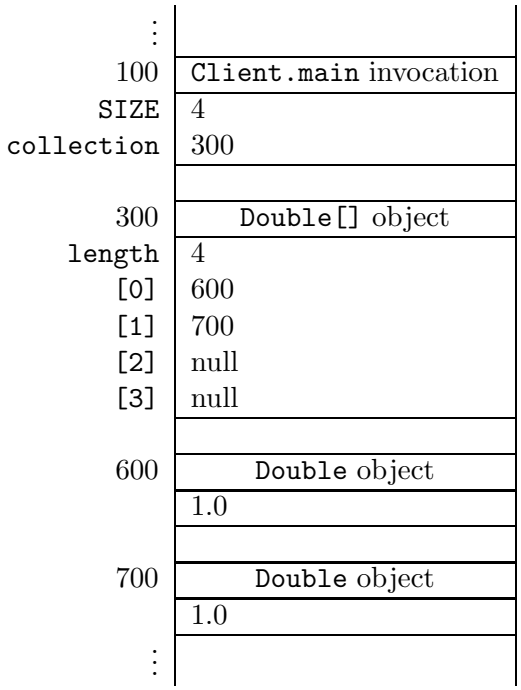


Let us consider another example.

```
1 output.print("Provide the size of the collection: ");
2 final int SIZE = input.nextInt();
3 Double[] collection = new Double[SIZE];
4 for (int i = 0; i < SIZE; i++)
5 {
```

```
6     collection[i] = new Double(1.0);
7 }
```

Once the execution reaches the end of line 6 for the second time, memory can be depicted as follows.



8.2 Implementing a List by means of an Array

8.2.1 Introduction

In this section we implement the `List` interface by means of an array. The API of the `List` interface can be found at this link.

The `List` interface is generic. This is reflected by the type parameter `T` in `List<T>`. The type parameter captures the type of the elements of the list. To use the `List` interface, the client has to provide a non primitive type as argument. For example, `List<Double>` represents a list containing `Double` objects.

Note that the `List` interface extends the `Iterable` interface. As a consequence, when we implement the `List` interface we not only have to provide an implementation for each method specified in the `List` interface, but also for the single method `iterator` specified in the `Iterable` interface.

Note also that our `List` interface is different from the interface `java.util.List`. To simplify matters, our `List` interface contains fewer methods and some of the methods are specified slightly differently.

We provide two implementations of the `List` interface: `BoundedArrayList` and `UnboundedArrayList`. In the former implementation, the maximum size of the list is bound, whereas in the latter imple-

mentation the list can grow unboundedly. Both classes are generic. The APIs of the classes can be found at this link and this link.

Like we used a `LinkedList` in Section 8.1.1, we can use a `BoundedArrayList` as follows.

```
output.print("Provide the size of the collection: ");
final int SIZE = input.nextInt();
long start = System.currentTimeMillis();
List<Double> collection = new BoundedArrayList<Double>(SIZE);
for (int i = 0; i < SIZE; i++)
{
   collection.add(new Double(1.0));
}
output.println(System.currentTimeMillis() - start);
```

If we run the above snippet for different sizes of the collection, we get the following times (in milliseconds).

SIZE	time
100000	23
1000000	131
10000000	5092

Note that the `BoundedArrayList` is more efficient than the `LinkedList`.

8.2.2 The Class Header

The class header is identical to the one found in the API.

```
public class BoundedArrayList<T> implements List<T>
```

The type parameter `T` captures the type of the elements of the list. The type parameter `T` can be used within the body of the class as a non primitive type. For example, we can declare a variable named `element` of type `T` as follows.

```
T element;
```

8.2.3 The Attributes Section

The API of the `BoundedArrayList` class contains a single public attribute named `CAPACITY`. As we can see in the API, this attribute is static and it is a constant of type `int`. In the API, we can also find its value, which is 100. Hence, we introduce the following declaration and initialization.

```
public static final int CAPACITY = 100;
```

To store the elements of the list, we introduce an array. The first element of the list is stored in the first cell of the array, the second element of the list is stored in the second cell of the array, etcetera. The elements of the list are of type `T`. Hence, it seems natural to introduce an array the base type of which is `T`. However, Java does not allow for arrays the base type of which is a type

parameter.[1] Since each class extends the `Object` class, each object is-an `Object`. Hence, an array of `Object`s can store elements of any type. Therefore, we introduce the following array to store the elements of the list.

```
1 private Object[] elements;
```

If the list contains n elements, then these elements are stored in the cells with indices $0, \ldots, n-1$. The cells with an index greater than or equal to n contain no element, that is, their values are null.

We also keep track of the size of the list by introducing the following attribute.

```
1 private int size;
```

To ensure that all the elements stored in the array are of type `T`, we will maintain the following class invariant.

```
1 /*
2  * class invariant:
3  * this.size >= 0 &&
4  * this.size <= this.elements.length &&
5  * for all 0 <= i < this.size : this.elements[i] != null && this.elements[i]
       instanceof T &&
6  * for all this.size <= i < this.elements.length : this.elements[i] == null
7  */
```

8.2.4 The Constructors Section

The API of the `BoundedArrayList` class contains three constructors. Both the default constructor and the constructor that takes a capacity as its argument give rise to an empty list, that is, the `size` is zero and all the cells of the array `elements` contain null. As we will see below, we can implement the former by delegating to the latter. In the latter constructor, we set the `size` to zero and we create an array of `Object`s of length `capacity` as follows.

```
1 public BoundedArrayList(int capacity)
2 {
3    this.size = 0;
4    this.elements = new Object[capacity];
5 }
```

Note that the cells of the array all contain null, since upon creation all cells of an array contain the default value of the base type of the array and null is the default value of `Object`.

The default constructor can simply delegate to the above implemented constructor as follows.

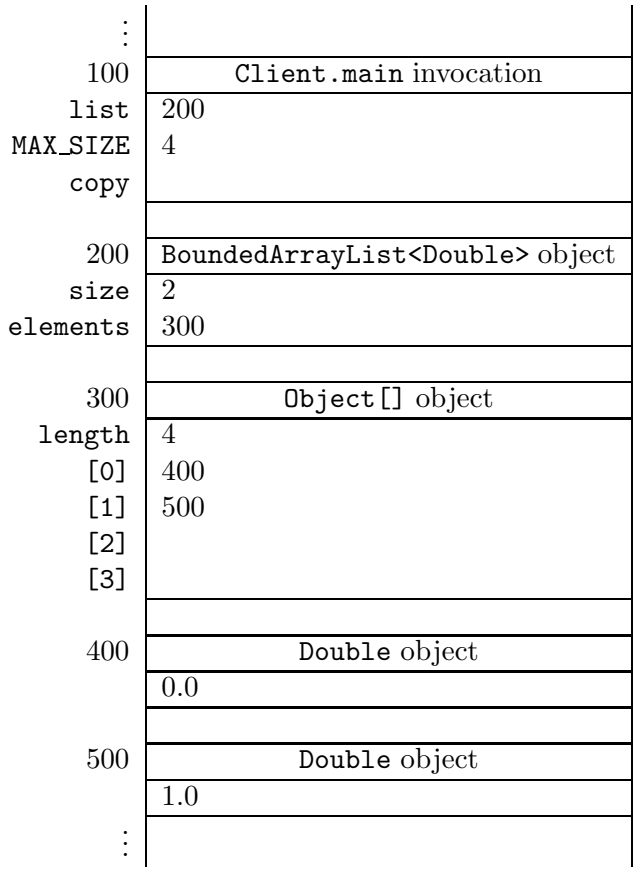
```
1 public BoundedArrayList()
2 {
3    this(BoundedArrayList.CAPACITY);
4 }
```

[1] The reasons why Java disallows such arrays is beyond the scope of these notes.

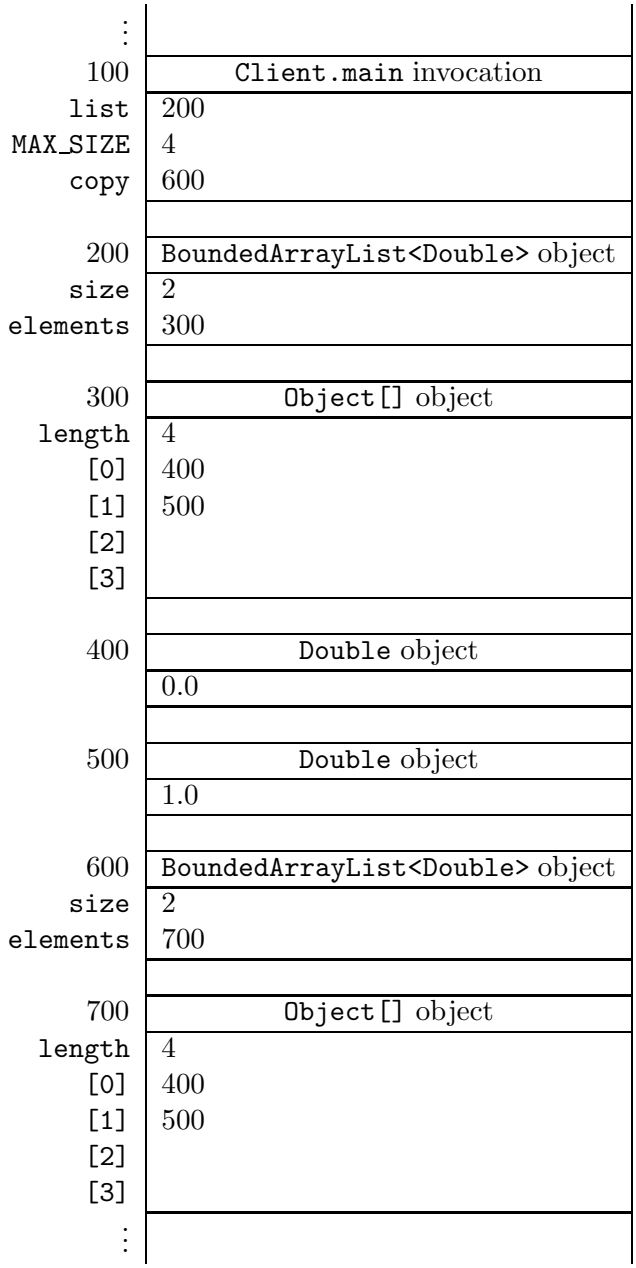
Before implementing the copy constructor, let us consider the following snippet of client code.

```
1 final int MAX_SIZE = 4;
2 List<Double> list = new BoundedArrayList<Double>(MAX_SIZE);
3 list.add(new Double(0.0));
4 list.add(new Double(1.0));
5 List<Double> copy = new BoundedArrayList<Double>(list);
```

When the execution reaches the end of line 4, memory can be depicted as follows.



After the execution reaches the end of line 5, our memory diagram looks as follows.



Note that the arrays at the addresses 300 and 700 contain the same `Double` objects, namely the ones at the addresses 400 and 500. This implies that we have to create a new array in the copy constructor and copy the content of the original array to the new one. Hence, we can implement the copy constructor as follows.

```
1 public BoundedArrayList(BoundedArrayList list)
2 {
3    this.size = list.size;
4    this.elements = new Object[list.elements.length];
```

```
5     for (int i = 0; i < list.size; i++)
6     {
7        this.elements[i] = list.elements[i];
8     }
9  }
```

From the class invariant we can conclude that the above code fragment does not throw an `ArrayIndexOutOfBoundEx`

Rather than creating a new array and copying the content from the old one to the new one ourselves, we can also delegate to the `Arrays` class, which is part of the `java.util` package. The static method `copyOf(original, length)` returns a new array and copies the values of the `original` array to the new one, truncating or padding with default values (if necessary) so the returned copy has the specified `length`. Exploiting this method, we can implement the copy constructor as follows.

```
1  public BoundedArrayList(BoundedArrayList list)
2  {
3     this.size = list.size;
4     this.elements = Arrays.copyOf(list.elements, list.elements.length);
5  }
```

8.2.5 The Methods Section

The accessor `getSize` can be implemented as follows.

```
1  public int getSize()
2  {
3     return this.size;
4  }
```

The method `get(index)` returns the element of the list with the given `index`. This corresponds to the element of the array `elements` with the given `index`. Note that the return type of the method `get` is `T` whereas the base type of the array `elements` is `Object`. Hence, we have to cast. If the given index is invalid, that is, it is smaller than zero, or it is greater than or equal to the size of the list, then the method `get` throws an `IndexOutOfBoundsException`.

```
1  public T get(int index) throws IndexOutOfBoundsException
2  {
3     if (index >= 0 && index < this.size)
4     {
5        return (T) this.elements[i];
6     }
7     else
8     {
9        throw new IndexOutOfBoundsException("Index is invalid");
10    }
```

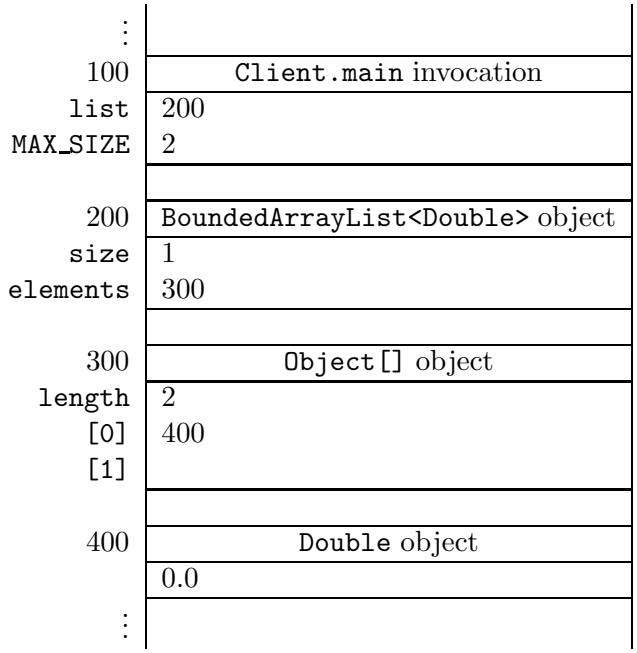
```
11 }
```

Note that we can conclude from the class invariant that the cast in line 5 will never fail.

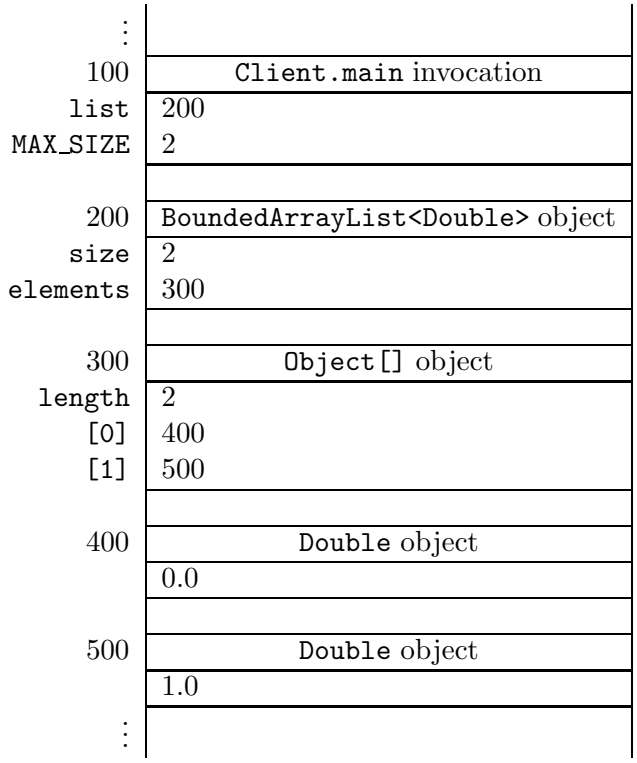
Before implementing the `add` method, let us consider the following client snippet.

```
1 final int MAX_SIZE = 2;
2 List<Double> list = new BoundedArrayList<Double>(MAX_SIZE);
3 list.add(new Double(0.0));
4 list.add(new Double(1.0));
5 list.add(new Double(2.0));
```

Once we reach the end of line 3, memory can be depicted as follows.



In line 4 we add another `Double` object to the collection. We add a reference to a `Double` object to the first available free cell, that is, the first cell that contains null. The index of this cell is the value of the attribute `size`. Furthermore, the attribute `size` has to be incremented by one to reflect that an element has been added to the collection. Since the element has been added successfully, the methods returns true. Once we reach the end of line 4, memory can be depicted as follows.



In line 5 we attempt to add yet another `Double` object to the collection. However, since there is no free cell available, the method simply returns false.

We can implement the `add` method as follows.

```
1  public boolean add(T element)
2  {
3     boolean added;
4     if (this.size == this.elements.length)
5     {
6        added = false;
7     }
8     else
9     {
10       this.elements[this.size] = element;
11       this.size++;
12       added = true;
13    }
14    return added;
15 }
```

The `contains(element)` method checks whether a given `element` is part of the list. This method can be implemented as follows.

```
1  public boolean contains(T element)
```

```
{
   boolean found = false;
   for (int i = 0; i < this.size; i++)
   {
      found = found || this.elements[i].equals(element);
   }
   return found;
}
```

The variable `found` keeps track whether we have found `element`. This is captured by the following loop invariant.

$$\texttt{found} == \exists 0 \leq j < \texttt{i} : \texttt{this.elements}[j].\texttt{equals}(\texttt{element})$$

From the class invariant we can conclude that the above method does not throw an `ArrayIndexOutOfBoundsExcept`

Once we have found `element`, we can exit the loop. This can be accomplished by changing the condition of the loop as follows.

```
public boolean contains(T element)
{
   boolean found = false;
   for (int i = 0; i < this.size && !found; i++)
   {
      found = found || this.elements[i].equals(element);
   }
   return found;
}
```

Now that we have modified the condition of the loop, we can further simplify the body of the loop as follows.

```
public boolean contains(T element)
{
   boolean found = false;
   for (int i = 0; i < this.size && !found; i++)
   {
      found = this.elements[i].equals(element);
   }
   return found;
}
```

The first part of the `remove(element)` method is (almost) the same as the `contains(element)` method: we try to locate the given `element`. If the element is not found then the method simply returns false. Otherwise, the element has to be removed from the list and true has to be returned. Assume that we have found the element at index i_f in the array. Then we have to move the elements at the indices $i_f + 1$, $i_f + 2$, ... one cell back. This can be accomplished by copying the

element from cell $i_f + 1$ to cell i_f, copying the element from cell $i_f + 2$ to $i_f + 1$, ..., and removing the element from the last cell (by setting its value to null).

```
public boolean remove(T element)
{
   boolean found = false;
   int i;
   for (i = 0; i < this.size && !found; i++)
   {
      found = this.elements[i].equals(element);
   }
   if (found)
   {
      for (; i < this.size; i++)
      {
         this.elements[i - 1] = this.elements[i];
      }
      this.elements[this.size - 1] = null;
      this.size--;
   }
   return found;
}
```

Note that we declare the variable `i` outside the scope of the first loop so that we can employ it in the second loop as well. Assume that we have found the element at index i_f. Then `i` is equal to $i_f + 1$ when exiting the first loop.

The following is a loop invariant for the second loop.

$\forall i_f < j <$ `i` : `this.elements`$[j]$ has been copied to `this.elements`$[j-1]$

Two lists are the same if they contain the same elements in the same order. Hence, the `equals` method can be implemented as follows.

```
public boolean equals(Object object)
{
   boolean equal;
   if (object != null && this.getClass() == object.getClass())
   {
      BoundedArrayList other = (BoundedArrayList) object;
      equal = this.size() == other.size();
      for (int i = 0; i < this.size() && equal; i++)
      {
        equal = equal && this.elements[i].equals(other.elements[i]);
      }
   }
   else
```

```
14     {
15        equal = false;
16     }
17     return equal;
18  }
```

For the above loop, we have the following loop invariant.

$$\texttt{equal} \;==\; \texttt{this.size}() == \texttt{other.size}() \;\&\&\; \forall 0 \leq j < \texttt{i} : \texttt{this.elements}[j].\texttt{equals}(\texttt{other.elements}[j])$$

The hash code of a list `list` is defined as

$$\sum_{0 \leq j < \texttt{list.getSize}()} 31^{\texttt{list.getSize}()-1-j} \times \texttt{list.get}(j).\texttt{hashCode}()$$

We can implement the `hashCode` method as follows.

```
1  public int hashCode()
2  {
3     final int BASE = 31;
4     int hashCode = 0;
5     for (int i = 0; i < this.size; i++)
6     {
7        hashCode = BASE * hashCode + this.elements[i].hashCode();
8     }
9     return hashCode;
10 }
```

A loop invariant for the above loop is

$$\texttt{hashCode} == \sum_{\texttt{i} < j < \texttt{this.size}} 31^{j} \times \texttt{this.elements}[j].\texttt{hashCode}()$$

Consider the following snippet of client code.

```
1  final int MAX_SIZE = 4;
2  List<Double> list = new BoundedArrayList<Double>(MAX_SIZE);
3  list.add(new Double(0.0));
4  list.add(new Double(1.0));
5  list.add(new Double(2.0));
6  output.println(list);
```

In line 6 the `toString` method is invoked implicitly. The above code fragment gives rise to the following output.

```
[0.0, 1.0, 2.0]
```

We can implement the `toString` method as follows.

```
1  public String toString()
2  {
3     StringBuffer representation = new StringBuffer("[");
4     if (this.size != 0)
5     {
6        for (int i = 0; i < this.size - 1; i++)
7        {
8           representation.append(this.elements[i]);
9           representation.append(", ");
10       }
11       representation.append(this.elements[this.size - 1]);
12    }
13    representation.append("]");
14    return representation.toString();
15 }
```

8.2.6 The `iterator` Method

According to the API of the `Iterable` interface, the method `iterator` returns an object of type `Iterator`. Since `Iterator` is an interface we cannot create instances of it. Hence, we have to return an instance of a class that implements the `Iterator` interface. Therefore, we develop a class that implements the `Iterator` interface. We call the class `BoundedArrayListIterator`. The `iterator` method of the `BoundedArrayList` class will be implemented in such a way that it returns an instance of the `BoundedArrayListIterator` class.

The `BoundedArrayListIterator` Class

To specify that our class implements `Iterator<T>`, we introduce the following class header.

```
1  public class BoundedArrayListIterator<T> implements Iterator<T>
```

For our class to implement `Iterator<T>`, we have to implement the following methods

```
boolean hasNext()
T next()
void remove()
```

according to the API of the `Iterator` interface. Recall that, by default, all methods specified in an interface are public.

The Attributes Section

Like any other class, we start with choosing the attributes. Let us first have a look at a small fragment of client code that utilizes the `BoundedArrayListIterator` class. The snippet

```
List<Double> list = new BoundedArrayList<Double>();
list.add(new Double(0.0));
list.add(new Double(1.0));
Iterator<Double> iterator = list.iterator();
while (iterator.hasNext())
{
   output.println(iterator.next());
}
```

produces the output

```
0.0
1.0
```

Recall that the `iterator` method will be implemented in such a way that it returns an instance of the `BoundedArrayListIterator` class. Hence, the variable `iterator` refers to a `BoundedArrayListIterator` object. This object needs to have access to the elements of the array `elements` of the `BoundedArrayList` object `list`. Hence, we introduce an attribute `elements` of type `Object[]`. Furthermore, we have to keep track which element to return next. For that purpose, we introduce an attribute `next` of type `int` that contains the index of the cell of the array `elements` which is returned when the method `next` is invoked. Hence, we declare the following two attributes.

```
private Object[] elements;
private int next;
```

Since `next` is used as an index of the array `elements`, we maintain the following class invariant.

```
/*
 * class invariant: this.next >= 0 &&
 * for all 0 <= i < this.elements.length : this.elements[i] instanceof T
 */
```

The Constructor Section

As we already mentioned above, the `elements` attribute of the `BoundedArrayListIterator` object should have access to the elements of the list, which are captured by the `elements` attribute of the `BoundedArrayList` object. The `BoundedArrayList` object can pass this information to the `BoundedArrayListIterator` object as an argument of the constructor. Hence, the header of the constructor of the `BoundedArrayListIterator` class is

```
public BoundedArrayListIterator(Object[] elements)
```

Before implementing this constructor, let us implement the `iterator` method of the `BoundedArrayList` class first.

```
1 public Iterator<T> iterator()
2 {
3    return new BoundedArrayListIterator<T>(this.elements);
4 }
```

Since the `BoundedArrayListIterator<T>` class implements the `Iterator<T>` interface, the return statement of the `iterator` method is compatible with the return type of the method.

Let us get back to the implementation of the constructor. We implement it as follows.

```
1 public BoundedArrayListIterator(Object[] elements)
2 {
3    this.elements = elements;
4    this.next = 0;
5 }
```

The Methods Section

We have left to implement the following methods.

```
boolean hasNext()
T next()
void remove()
```

To test if the iterator has a next element, we have to check if the cell of the array `elements` with index `next` is filled. To ensure that we do not get an `ArrayIndexOurOfBoundsException`, we have to verify that the index `next` is valid, that is, is smaller than `elements.length` (from the class invariant we already know that it is greater than or equal to zero). Hence, we can implement the `hasNext` method as follows.

```
1 public boolean hasNext()
2 {
3    return this.next < this.elements.length && this.elements[this.next] != null
        ;
4 }
```

Recall that if the left hand side of the conjunction, `this.next < this.elements.length`, evaluates to false, then the right hand side of the conjunction, `this.elements[this.next] != null`, is not evaluated in Java.

The `next` method returns the next element. If no such element exists, then a `NoSuchElementException` is thrown. This method can be implemented as follows.

```
1 public T next() throws NoSuchElementException
2 {
3    if (this.hasNext())
4    {
```

```
5        T element = (T) this.elements[this.next];
6        this.next++;
7        return element;
8     }
9     else
10    {
11       throw new NoSuchElementException("Iterator has no more elements");
12    }
13 }
```

According to the API of the `Iterator` interface, the `remove` method is optional. The API also specifies that an `UnsupportedOperationException` has to be thrown if the method is not supported. To keep the class as simple as possible, we decided not to support the method. Hence, the `remove` method can be implemented as follows.

```
1  public void remove() throws UnsupportedOperationException
2  {
3     throw new UnsupportedOperationException("Iterator does not support remove")
          ;
4  }
```

8.2.7 Unbounded Size

Whereas the maximum size of a `BoundedArrayList` is fixed, an `UnboundedArrayList` can grow without any limit. In the `UnboundedArrayList` class, the only method that needs to be implemented differently from the `BoundedArrayList` class is the `add` method. In case there is no free cell available for the element to be added to the list, we simply double the length of the array. That is, we create a new array of twice the length and copy all elements from the original array to the new one.[2]

```
1  public boolean add(T element)
2  {
3     if (this.size == this.elements.length)
4     {
5        this.elements = Array.copyOf(this.elements, this.elements.length * 2);
6     }
7     this.elements[this.size] = element;
8     this.size++;
9     return true;
10 }
```

[2]Doubling the length of the array is much more efficient than adding a single cell to the array. The details are beyond the scope of these notes.

8.3 Command Line Arguments

The main method of an app has the following header.

```
1 public static void main(String[] args)
```

The method has a single parameter of type `String[]`, an array of `String`s. This array contains the so called command line arguments.

The next app uses the command line arguments.

```
1  public class StudentClient
2  {
3     public static void main(String[] args)
4     {
5        PrintStream output = System.out;
6
7        if (args.length == 0)
8        {
9           output.println("Provide a student ID as the command line argument");
10       }
11       else
12       {
13          final String VALID = "\\d{9}";
14          if (args[0].matches(VALID))
15          {
16             Student student = new Student(args[0]);
17             .
               .
               .
18
19          }
20          else
21          {
22             output.println("Provide a valid student ID as the command line
                  argument");
23          }
24       }
25    }
26 }
```

In line 7 we check if any command line arguments are provided. In line 13 and 15, the first command line argument is used.

When running the app, the ID 123456789 can be provided as a command line argument as follows.

```
java StudentClient "123456789"
```

8.4 Beyond the Basics

8.4.1 The `getSize` Method

The attribute `size` is redundant since it can be computed from the attribute `elements` as follows.

```
1 public int getSize()
2 {
3    int size = 0;
4    while (size < this.elements.length && this.elements[size] != null)
5    {
6       size++;
7    }
8    return size;
9 }
```

This implies that we do not have to introduce the attribute `size`. However, the attribute `size` somewhat simplifies our code and makes it more efficient as we will see below.

Let us determine the running time[3] of the above `getSize` method. First, let us estimate the number of elementary operations that are performed at each line of the method.

line	number
3	3
4	11
6	3
8	2

Next, we determine how often each line is executed. Obviously, line 3 and 8 are only executed once. How often line 4 and 6 are executed depends on the number elements that are stored in the array `elements`. Assume that this array contains n elements. Then line 4 is executed $n + 1$ times and line 6 is executed n times. Hence, we estimate the total number of elementary operations to be

$$3 + (n + 1) \times 11 + n \times 3 + 2 = 14n + 16.$$

Therefore, (an estimate of) the number of elementary operations performed by the above `getSize` method can be represented by the function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = 14n + 16,$$

where n is the number of elements stored in the array `elements`.

Proposition 1 $f \in O(n)$.

Proof According to Definition 3, we have to show that

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \leq M : 14n + 16 \leq F \times n.$$

[3]We refer the reader to Appendix A for an introduction to running time analysis.

First, we have to pick a particular "minimal size" M and a particular "factor" F. We pick $M = 16$ and $F = 15$. Then it remains to show that

$$\forall n \geq 16 : 14n + 16 \leq 15n.$$

Let n be an arbitrary number with $n \geq 16$. Then

$$14n + 16 \leq 14n + n = 15n.$$

□

Hence, we say that the running time of the above `getSize` method is $O(n)$, where n is the number elements of the list.

Let us also determine the running time of the `getSize` method of Section 8.2.5. We estimate the number of elementary operations executed in line 3 to be 3. This line is only executed once. Hence, (an estimate of) the number of elementary operations performed by the `getSize` method can be represented by the function $g : \mathbb{N} \to \mathbb{N}$ defined by

$$g(n) = 3,$$

where n is the number of elements stored in the array `elements`.

Proposition 2 $g \in O(1)$.

Proof According to Definition 3, we have to show that

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \leq M : 3 \leq F \times 1.$$

First, we have to pick a particular "minimal size" M and a particular "factor" F. We pick $M = 1$ and $F = 3$. Then it remains to show that

$$\forall n \geq 1 : 3 \leq 3 \times 1$$

which is obviously the case. □

Hence, we say that the running time of the `getSize` method is $O(1)$.

The implementation presented in Section 8.2.5 only performs a constant number of elementary operations, no matter how many elements the list contains. However, if we were not to introduce the (redundant) attribute `size`, then we would have to implement the `getSize` method as described above. In that case, the number of elementary operations would depend on the number elements of the list. This would be less efficient, especially if the number of elements of the list is large.

8.4.2 The `contains` and `remove` Methods

As we already mentioned in Section 8.2.5, the first part of the `remove` method is almost identical to the `contains` method. To avoid code duplication, we can introduce an auxiliary method that consists of the common part.

The method `find(element)` tries to find an index at which the given `element` can be found in the list. If no such index exists, then the method returns a special value different from any valid index, namely -1.

```
1  private int find(T element)
2  {
3     int index = -1;
4     for (int i = 0; i < this.size && index == -1; i++)
5     {
6        if (this.elements[i].equals(element))
7        {
8           index = i;
9        }
10    }
11    return index;
12 }
```

We have that

$$\begin{array}{ll} & (\texttt{index} == -1 \ \&\& \ \forall 0 \leq j < \texttt{i} : !\texttt{this.element}[j].\texttt{equals}(\texttt{element})) \\ || & (\texttt{index} != -1 \ \&\& \ \texttt{this.element}[\texttt{index}].\texttt{equals}(\texttt{element})) \end{array}$$

is a loop invariant for the above loop.

Using the `find` method, we can implement the `contains` method as follows.

```
1  public boolean contains(T element)
2  {
3     return this.find(element) != -1;
4  }
```

We can also use the `find` method to implement the `remove` method.

```
1  public boolean remove(T element)
2  {
3     int index = this.find(element);
4     if (index != -1)
5     {
6        for (int i = index + 1; i < this.size; i++)
7        {
8           this.elements[i - 1] = this.elements[i];
9        }
10       this.elements[this.size - 1] = null;
11       this.size--;
12    }
13    return index != -1;
14 }
```

8.4.3 The `BoundedArrayListIterator` as an Inner Class

The `BoundedArrayListIterator` class is only used in the `BoundedArrayList` class. To ensure that only the `BoundedArrayList` class can make use of the `BoundedArrayListIterator` class, we can define the latter class as a private inner class of the former class as follows.

```
1  public class BoundedArrayList<T> implements List<T>
2  {
3     private int size;
4     private Object[] elements;
5
6     ⋮
7
8     private class BoundedArrayListIterator<T> implements Iterator<T>
9     {
10       ⋮
11    }
12 }
```

Note that the class `BoundedArrayListIterator` is private and, hence, will not show up in the API. Also note that the class `BoundedArrayListIterator` is defined within the scope of the `BoundedArrayList` class and, hence, is not visible outside the latter class.

Appendix A

Running Time Analysis

We use $\mathbb{N}$ to denote the set $\{0, 1, 2, \ldots\}$ of natural numbers. Let $f : \mathbb{N} \to \mathbb{N}$ be a function from natural numbers to natural numbers. We will associate such a function with a piece of Java code. This function will capture the answer to the question "How many elementary operations are performed at most when executing the Java code for an input of a given size?"

Consider the following method.

```
1  /**
2   * Returns the factorial of the given number.
3   *
4   * @param n a number.
5   * @pre. n > 0
6   * @return the factorial of the given number.
7   */
8  public static long factorial(int n)
9  {
10    long factorial = 1;
11    int i = 1;
12    while (i <= n)
13    {
14       factorial *= i;
15       i++;
16    }
17    return factorial;
18 }
```

Let us first estimate how many elementary operations are performed at each line of the above method.

line	number
10	2
11	2
12	4
14	4
15	3
17	2

For example, in line 14 we

- look up the value of `i`,
- look up the value of `factorial`,
- multiply these two values, and
- assign the result of the multiplication to `factorial`.

We will come back to the fact that these are just estimates. Next, we determine how often each line is executed. Obviously, line 10, 11 and 17 are executed once. How often line 12, 14 and 15 are executed depends on the value of `n`. Assume the value of the variable `n` is the integer n. Then, line 12, 14 and 15 are executed n times. Therefore, we estimate the total number of elementary operations to be

$$2 + 2 + n \times 4 + n \times 4 + n \times 3 + 2 = 11n + 6.$$

Hence, (an estimate of) the total number of elementary operations can be captured by a function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = 11n + 6,$$

where n represents the value of `n`.

The actual number of elementary operations that are performed depends on many factors. For example, some computers can assign a value to a variable of type `long` in a single operation whereas other computers may need two. Hence, estimates like

$$f_1(n) = 14n + 12$$

and

$$f_2(n) = 7n + 3$$

are equally reasonable. The big O notation provides us with an abstraction that can capture all these estimates. The functions f, f_1 and f_2 are all elements of $O(g)$, where the function $g : \mathbb{N} \to \mathbb{N}$ is defined by

$$g(n) = n.$$

Why are f, f_1 and f_2 elements of $O(g)$? For that we need the formal definition of the big O notation.

Definition 3 *Let* $f : \mathbb{N} \to \mathbb{N}$ *and* $g : \mathbb{N} \to \mathbb{N}$ *be functions. Then* $f \in O(g)$ *if*

$$\exists M \in \mathbb{N} : \exists F \in \mathbb{N} : \forall n \geq M : f(n) \leq F \times g(n),$$

That is, we can find a particular "minimal size" M and a particular "factor" F such that for all inputs of size n that are greater than or equal to the "minimal size," we have that $f(n) \leq F \times g(n)$.

The above introduced function g, which assigns to n the value n, is often simply denoted by n and, hence, we often write $f \in O(n)$ instead of $f \in O(g)$.

Proposition 4 $f \in O(n)$.

Proof To prove that $f \in O(n)$ we need to pick a particular M and a particular F. We pick $M = 6$ and $F = 12$. Then it remains to show that

$$\forall n \geq 6 : 11n + 6 \leq 12n,$$

that is, $11n + 6 \leq 12n$ for all $n \geq 6$. Let n be an arbitrary natural number with $n \geq 6$. Then

$$11n + 6 \leq 11n + n = 12n.$$

Hence, the above is true. □

Similarly, we can prove that $f_1 \in O(n)$. This time we pick $M = 12$ and $F = 15$. It remains to prove that

$$\forall n \geq 12 : 14n + 12 \leq 15n,$$

which is trivially true. To prove that $f_2 \in O(n)$, we pick $M = 8$ and $F = 3$. In this case, it remains to prove that

$$\forall n \geq 8 : 7n + 3 \leq 8n,$$

which is obviously true.

Let us consider another Java snippet.

```
1  /**
2   * Sorts the given array.
3   *
4   * @param a an array of integers.
5   * @pre. a != null && a.length > 0
6   */
7  public static void sort(int[] a)
8  {
9     for (int i = 0; i < a.length; i++)
10    {
11       int min = i;
12       for (int j = i + 1; j < a.length; j++)
```

```
13          {
14              if (a[j] < a[min])
15              {
16                  min = j;
17              }
18          }
19          int temp = a[i];
20          a[i] = a[min];
21          a[min] = temp;
22      }
23  }
```

Again, let us first estimate how many elementary operations are performed at each line of the above method.

line	number
9	10
11	2
12	10
14	8
16	2
19	4
20	6
21	4

Next, we determine how often each line is executed. Line 9, 11, 19, 20 and 21 are executed n times, where n is the length of the array. Line 12 and 14 are executed

$$(n-1)+(n-2)+\cdots+1=\sum_{i-1}^{n-1} i=\frac{n(n-1)}{2}$$

times.[1]

[1]We can prove that for all $n \geq 1$, $\sum_{i-1}^{n-1} i = \frac{n(n-1)}{2}$ by induction on n. In the base case, when $n = 1$, we have that $0 = \frac{1\times 0}{2}$. In the inductive step, let $n > 1$ and assume that $\sum_{i-1}^{n-2} i = \frac{(n-1)(n-2)}{2}$ (the induction hypothesis). Then

$$\begin{aligned}
\sum_{i-1}^{n-1} i &= (\sum_{i-1}^{n-2} i) + (n-1) \\
&= \frac{(n-1)(n-2)}{2} + (n-1) \quad \text{[induction hypothesis]} \\
&= \frac{n^2-3n+2+2n-2}{2} \\
&= \frac{n^2-n}{2} \\
&= \frac{n(n-1)}{2}.
\end{aligned}$$

Note that we do not know how often line 16 is executed. However, we do know that line 16 is executed at most $\frac{n(n-1)}{2}$ times. An upperbound of the total number of elementary operations that is performed is

$$\begin{aligned} & n \times 10 + n \times 2 + \frac{n(n-1)}{2} \times 10 + \frac{n(n-1)}{2} \times 8 + \frac{n(n-1)}{2} \times 2 + n \times 4 + n \times 6 + n \times 4 \\ = \quad & 10n^2 + 16n. \end{aligned}$$

This can be captured by a function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = 10n^2 + 16n.$$

To abstract from the estimates, we again exploit the big O notation and show that $f \in O(n^2)$, where we use n^2 to denote the function that maps n to n^2.

Proposition 5 $f \in O(n^2)$.

Proof To show that $f \in O(n^2)$, we pick $M = 16$ and $F = 11$. It remains to prove that

$$\forall n \geq 16 : 10n^2 + 16n \leq 11n^2.$$

Let $n \geq 16$. Then

$$10n^2 + 16n \leq 10n^2 + n^2 = 11n^2.$$

That concludes the proof. □