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.

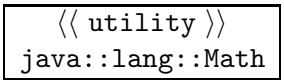
1.1.2 Why are Utilities Needed?

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`).

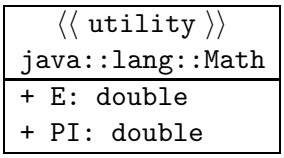
It should be noted that most, if not all, classes that make up a real-life application are *not* utilities. We start with utilities because they are simpler than non-utilities. Moreover, the skills we will acquire in implementing them apply as-is to non-utility classes.

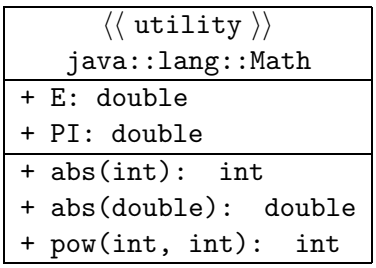
1.1.3 UML and Memory Diagrams

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.

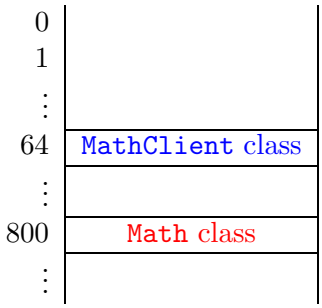


If additional features need to be exposed then a second (and a third) box can be added to hold the attributes (and methods) of the utility, as shown below. The + sign appearing before a feature indicates that the feature is public.

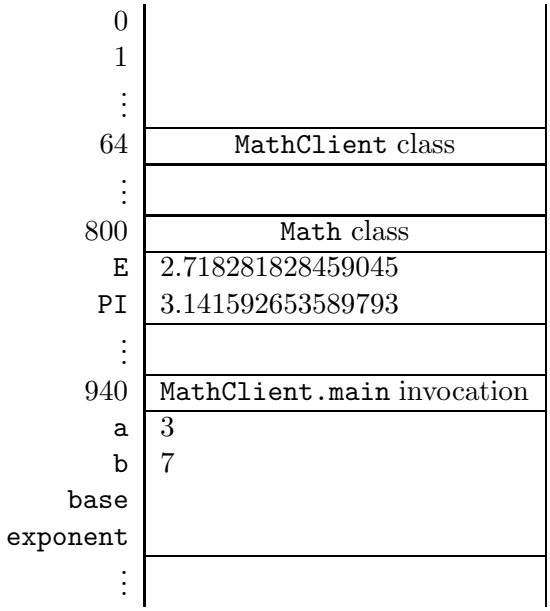




In addition to UML diagrams, we sometimes use memory diagrams to show how different classes reside in memory. For example, the memory diagram below contains two class blocks representating the client class named `MathClient` residing in memory beginning at address `64` and the `Math` utility at address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

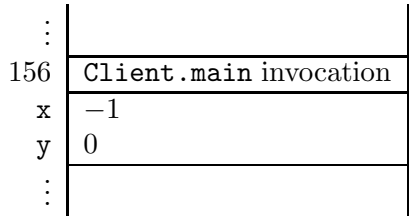
1.3.1 Passing Arguments By Value

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

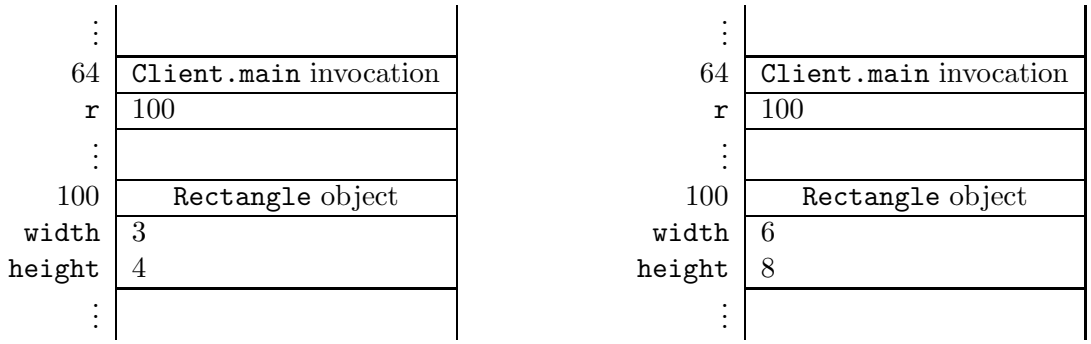
after an invocation (but the object to which it refers might). Let us look at an example that illustrates this point: assume that there is a utility called `Doubler` with the method:

```
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)
```

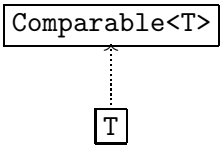
This proposal is clearly not typesafe because it does guarantee that the list elements are comparable with `x`. The implementer will not be able to invoke `compareTo` on `x` without casting it first, and such a cast may lead to a runtime error.

```
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)
```

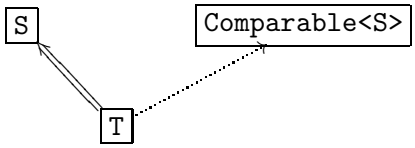
In this case, the type parameter `T` is restricted: only a class like `Date` that implements the interface `Comparable<Date>` can be used. This restriction can roughly be captured by the following UML-like diagram.



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)
```

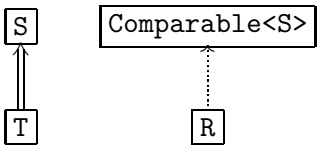
Also in this case, the type parameter `T` is restricted. This case is less restrictive than the previous one. The `?` is called a *wildcard*. The pattern `? super T` is matched by any super class of `T` or `T` itself. In this case, the restriction can roughly be captured by the following UML-like diagram.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