

# 1

## A General Introduction to Programming

### WHAT YOU WILL LEARN IN THIS CHAPTER:

---

- The key steps in a programming process
- The different types of programming errors
- The key principles of software testing
- The different types of software maintenance
- The key principles of structured programming

### **WROX.COM CODE DOWNLOADS FOR THIS CHAPTER**

The wrox.com code downloads for this chapter are found at [www.wrox.com/go/beginningjavaprogramming](http://www.wrox.com/go/beginningjavaprogramming) on the Download Code tab. The code is in the Chapter 1 download and individually named according to the names throughout the chapter.

Developing good and correct software is a very important challenge in today's business environment. Given the ubiquity and pervasiveness of software programs into our daily lives, the impact of faulty software is now bigger than ever. Software errors have caused flight crashes, rocket launch errors, and power blackouts, to name a few examples. Hence, it is important to design high-quality, error-free software programs. This chapter covers the fundamental concepts of programming. First, it elaborates on the programming process. The next section provides a sneak preview of object-oriented programming. This is followed by a short discussion on programming errors. The basic principles of software testing and software maintenance are also discussed. The chapter concludes by giving some recommendations relating to structured programming. You will revisit many of these ideas in future chapters, with a more hands-on approach.

## THE PROGRAMMING PROCESS

A *program* (also referred to as an application) is a set of instructions targeted to solve a particular problem that can be unambiguously understood by a computer. To this end, the computer will translate the program to the language it understands, which is machine language consisting of 0s and 1s. Computers execute a program literally as it was programmed, nothing more and nothing less. Programming is the activity of writing or coding a program in a particular programming language. This is a language that has strict grammar and syntax rules, symbols, and special keywords. People who write programs are commonly referred to as programmers or application developers. The term software then refers to a set of programs within a particular business context.

An example of a programming exercise is a program that calculates the body mass index (BMI) of a person. The BMI is calculated by dividing a person's weight in kilograms by the square of his or her height in meters. A person is considered overweight if his or her BMI is over 25. A BMI calculator program then requires the weight and height as inputs and calculates the associated BMI as the output. This is illustrated in Figure 1-1. This BMI example is used to demonstrate the steps in the software development cycle.

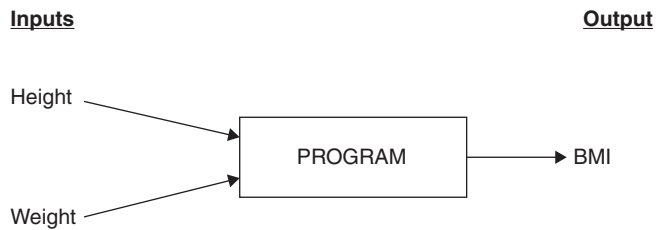


FIGURE 1-1

Programs are typically written using a step-by-step approach, as follows:

1. Requirements gathering and analysis
2. Program design
3. Program coding
4. Translation to machine language
5. Testing and debugging
6. Deployment
7. Maintenance

Because our environment is continuously evolving, software, too, is often continually reviewed and adapted. Therefore, these steps are often represented as a cycle, as shown in Figure 1-2, rather than as a ladder.

The first step is to make sure you understand the problem in sufficient detail. This means analyzing the problem statement carefully so you fully grasp all the requirements that need to be fulfilled by the software program. This may involve Q&A sessions, interviews, and surveys

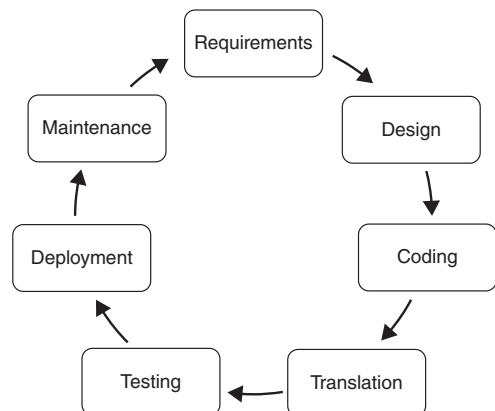


FIGURE 1-2

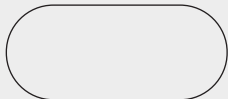


with business experts who have the necessary subject matter expertise. Even if you are programming for yourself, taking the time upfront to consider all the demands you want your program to meet will limit the amount of changes required later in the process. At the end of this step, it is important to know what the input to the program will receive and what output it should give. In the BMI example, you will need to know whether the height will be measured in meters or feet and the weight in kilos or pounds. You would also want to determine whether the output should be just the BMI results or also a message stating whether or not the person is overweight.

Once you have a thorough understanding of the business problem, you can start thinking about ways to solve it using a computer program. In other words, which processing steps should take place on the input(s) in order to give the desired output(s)? The procedure needed to solve the problem is also often referred to as the *algorithm*. When working out an algorithm, common sense and creativity both play an important role. A first useful step in designing an algorithm is planning the application logic using pseudo-code or flowcharts. Pseudo-code is a type of structured English but without strict grammar rules. It is a user-friendly way of representing application logic in a sequential, readable format. It allows the problem statement to be broken into manageable pieces in order to reduce its complexity. Following is an example of pseudo-code for the BMI case. A flowchart represents the application in a diagram, whereby the boxes show the activities and the arrows the sequences between them. Table 1-1 presents an overview of the most important flowchart construction concepts. Figure 1-3 then gives an example of a flowchart for the BMI case. Both pseudo-code and flowcharts can be used concurrently to facilitate the programming exercise. A key advantage of flowcharts when compared to pseudo-code is that they are visual and thus easier to interpret.

```
ask user: height
ask user: weight
if height = 0 or weight = 0:
error: "Incorrect input values"
return to beginning (ask height and weight)
end if
x = weight / (height * height)
message: "Your BMI is ",x
```

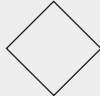



Table 1-1 is an overview of the most important flowchart modeling concepts.

**TABLE 1-1:** Key Flowchart Modeling Concepts

FLOWCHART SYMBOL	MEANING
	A terminator shows the start and stopping points of the program.
	An arrow shows the direction of the process flow.
	A rectangle represents a process step or activity.

*continues*

TABLE 1-1: (continued)

FLOWCHART SYMBOL	MEANING
	A diamond indicates a decision point in the process.
	This symbol represents a document or report.
	This rhombus represents data used as inputs/outputs to/from a process.
	This cylinder represents a database.

A next step is to code the program in a particular programming language. The choice of the language will depend on the programming paradigm and the platform adopted (such as hardware, operating system, or network).

Once the source code of the program has been written, it will be given to a translator to translate it to machine language (0s and 1s) so that it can be executed and solve the business problem.

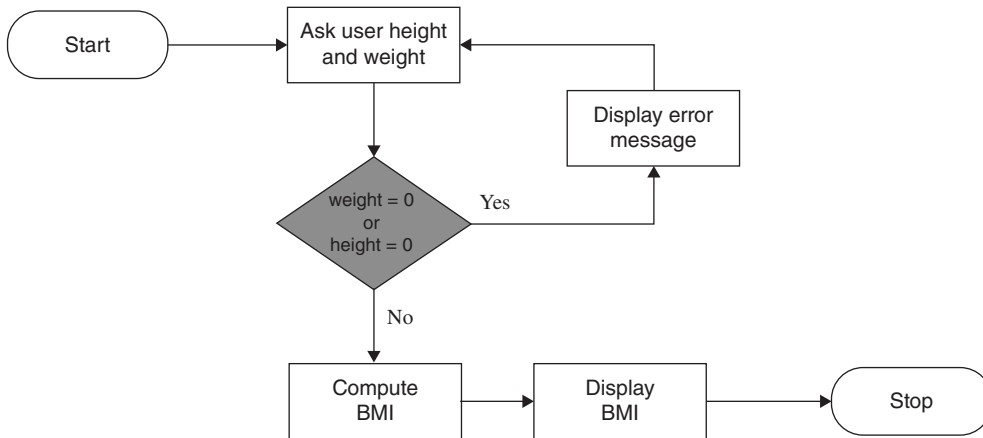


FIGURE 1-3

During application development, it is important that every program is intensively tested to avoid any errors. Often, in programming, errors are called *bugs*. Various types of errors exist and an entire chapter is devoted to this topic. Programming tools frequently have debugging facilities built in to

easily track bugs down and correct them. It is possible to debug your program without the use of such tools, but in either case, you should follow a structured and systematic review to be sure you've identified any bugs before your program is deployed.

Once a program has been thoroughly tested, it can be deployed. This means that the program will be brought into production and actively used to solve the business problem. Remember, users of your software don't usually understand as much about programming as you. Try to keep them in mind throughout the process to make this deployment step as seamless as possible.

Finally, programs should be maintained on an ongoing basis. There are many reasons for regular maintenance, namely correcting newly discovered bugs, accommodating changing user needs, preventing erroneous user input, or adding new features to existing programs.

It is important to note that programming is not a strict, sequential, step-by-step process. Quite to the contrary, it often occurs as an iterative process, whereby the original business problem is refined or even reformulated during the coding process.

## OBJECT-ORIENTED PROGRAMMING: A SNEAK PREVIEW

In object-oriented (OO) programming, an application consists of a series of objects that ask services from each other. Each object is an instance of a class that contains a blueprint description of all the object's characteristics. Contrary to procedural programming, an object bundles both its data (which determines its state) and its procedures (which determines its behavior) in a coherent way. An example of this could be a student object having data elements such as ID, name, date of birth, email address, and so on, and procedures such as `registerForCourse`, `isPassed`, and so on. A key difference between OO and procedural programming is that OO uses local data stored in objects, whereas procedural programming uses global shared data that the various procedures can access directly. This has substantial implications from a maintenance viewpoint. Imagine that you want to change a particular data element (rename it or remove it). In a procedural programming environment, you would have to look up all procedures that make use of the data element and adapt them accordingly. For huge programs, this can be a very tedious maintenance exercise. When you're using an OO programming paradigm, you only need to change the data element in the object's definition and the other objects can keep on interacting with it like they did before, minimizing the maintenance.

OO programming is the most popular programming paradigm currently in use. Some examples of object-oriented programming languages are Eiffel, Smalltalk, C++, and Java.

The following code example demonstrates how to implement the BMI example in Java. Contrary to the procedural programming example, it can be clearly seen that the data (`weight`, `height`, and `BMI`) is bundled together with the procedures (`BMICalculator`, `calculate`, and `isOverweight`) into one coherent class definition.

```
public class BMICalculator {
    private double weight, height, BMI;

    public BMICalculator( double weight, double height ){
        this.weight = weight;
        this.height = height;
    }
}
```

```
public void calculate(){
    BMI = weight / (height*height);
}

public boolean isOverweight(){
    return (BMI > 25);
}
}
```

## PROGRAMMING ERRORS

A programming error is also referred to as a *bug*, and the procedure for removing programming errors is called *debugging*. Debugging usually has the following three steps:

1. Detect that there is an error.
2. Locate the error. This can be quite time consuming for big programs.
3. Solve the error.

Different types of programming errors exist and are explored in the following sections.

### Syntax/Compilation Errors

A syntax or compilation error refers to a grammatical mistake in the program. Examples are a punctuation error or misspelling of a keyword. These types of errors are typically caught by the compiler or interpreter, which will generate an error message. Consider the following Java example:

```
public void calculate(){
    BMI = weight / (height*height),
}
```

The statement that calculates the BMI should end with a semicolon (;) instead of a comma (,), according to the Java syntax rules. Hence, a syntax error will be generated and displayed. Syntax errors are usually easy to detect and solve.

### Runtime Errors

A runtime error is an error that occurs during the execution of the program. Consider the following piece of Java code to calculate the BMI:

```
public void calculate(){
    BMI = weight / (height*height);
}
```

If the user enters a value of 0 for height, a division by zero occurs. This creates a runtime error and will likely crash during execution. Another example of a runtime error is an infinite loop into which the program enters at execution. During the design of the program, it is important to think about possible runtime errors that might occur due to bad user input, which is where the majority of bugs will originate. These errors should be anticipated as much as possible using appropriate error-handling routines, as we will discuss later.

## Logic/Semantic Errors

Logic or semantic errors are the hardest to detect since the program will give an output and not generate an error. However, the output that is given is incorrect due to a formula being incorrectly programmed. Consider the BMI example again:

```
public void calculate(){
    BMI = (weight*weight) / height;
}
```

This routine is clearly erroneous since it calculates the BMI as  $(\text{weight} * \text{weight}) / \text{height}$  instead of  $\text{weight} / (\text{height} * \text{weight})$ . These errors cannot be detected by compilers or interpreters.

## PRINCIPLES OF SOFTWARE TESTING

In order to avoid software errors (and their impact), a program should be thoroughly tested for any remaining errors before it is brought into production. The main purpose of testing is verification and validation of the software build. Verification aims at answering the question as to whether the system was built correctly, whereas validation tries to determine whether the right system was built. The quicker an error is found during development, the cheaper it is to correct it. As illustrated in Figure 1-4, the cost of testing typically increases exponentially, whereas the cost of missed bugs decreases exponentially with the amount of testing conducted. The optimum testing resources can then be found where both curves intersect.

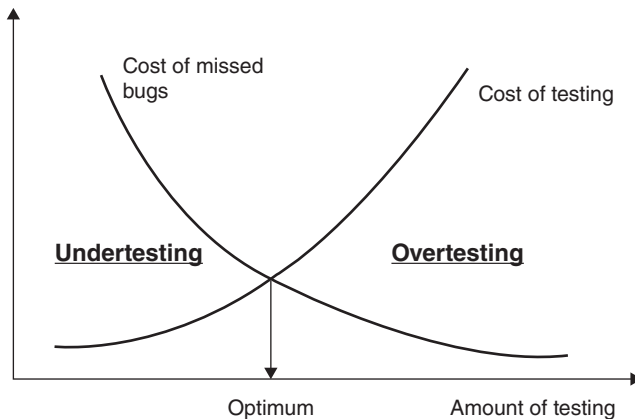


FIGURE 1-4

A first basic way of testing is to desk-check the program by using paper and pencil. The manual calculations and output can then be contrasted with the program calculations and output. It is especially important to consider extreme cases and see how the program behaves. Of course, this only works for small-scale programs; more sophisticated procedures might be needed for bigger programs.

Static testing procedures test the program not by executing it, but by inspecting and reviewing the code and performing detailed walk-throughs. It is aimed at verification. On the other hand, dynamic testing procedures test the program by executing it with carefully selected test cases. It is thus more related to validation. The test cases can be chosen according to a *white box* or *black box* strategy. In white box testing, they are selected by thorough inspection of the source code of the program; for example, by making sure all the branches in an `if-then-else` selection are covered, boundary conditions for loops are verified, and so on. One popular approach is to intentionally inject faults in the source code, which then need to be tracked down in a follow-up step. Black box testing considers the program as a black box and does not inspect its internal source code. One example is a procedure that tries to test all possible input parameter combinations. It is especially important to also test what happens when impossible values are entered (such as a negative value for weight and height, value of 0 for height, missing value for gender, and so on). Obviously, this becomes computationally infeasible in case many inputs are present and intelligent sampling procedures could be adopted to test as many useful input combinations as possible.

Software development typically has two phases of testing. *Alpha testing* is done internally by the application developers before the software is brought to the market. In *beta testing*, the software is given to a selected target audience and errors are reported back to the development team.

## SOFTWARE MAINTENANCE

Software is always dynamically evolving, even after delivery. Maintenance is the activity of adjusting the program after it has been taken into production. This is done to boost its performance, solve any remaining errors, and/or accommodate new user requirements. Maintenance typically consumes a large part of the overall software development costs (up to 70% or more according to some estimates). This can be partly explained by the fact that much of the software people work with today is relatively old (legacy software) and has been maintained on an ongoing basis. This section covers the four main types of maintenance. They are categorized according to their intended goals.

### Adaptive Maintenance

Adaptive software maintenance refers to modifying a program to accommodate changes in the environment (both hardware and software). An example of this is a new Windows release with new features added (which can also be used by the program) and old features removed (which can no longer be used by the program).

### Perfective Maintenance

This refers to enhancing a program to support new or changed user requirements. Consider again the BMI example. When the user wants to be able to enter height in feet units and weight in pound units, this is a perfective maintenance operation.

### Corrective Maintenance

Corrective maintenance aims at finding errors during runtime and fixing them. A further distinction can be made here between emergency fixes (which need to be solved as quickly as possible due to their critical relevance) and routine debugging (which is less urgent).



## Preventive Maintenance

Preventive maintenance aims at increasing software maintainability in order to prevent future errors. A popular example here was the Y2K problem, where companies massively anticipated date calculation errors in their software programs at the end of the previous century. Another example concerns the transition of many countries from their own independent currency toward the Euro. One important activity to facilitate preventive maintenance is documentation. This means that the application is extended with various comments that are not executed by the compiler, but that indicate the meaning of the various data elements, procedures, and operations in order to facilitate future maintenance.

Among the four types of maintenance, perfective maintenance typically takes the main share of all maintenance efforts (it can even be more than 50%), followed by adaptive, corrective, and preventive maintenance.

The major causes of maintenance problems are unstructured code, lack of documentation, excessive user demand for changes, lack of user training and understanding, and high user turnover. Many organizations have standard procedures for maintenance, which typically start with the formal filing of a change request specifying the modifications needed to the software. Depending on the severity of the request and the change management strategy adopted by the organization, these change requests can be grouped and dealt with at fixed time stamps, or treated immediately.

## PRINCIPLES OF STRUCTURED PROGRAMMING

To finish this introductory chapter, this section discusses some of the basic principles of structured programming.

A first important concept is *stepwise refinement*. Programs should be designed using a top-down strategy where the problem statement is subdivided into smaller, more manageable subproblems. These subproblems can be further broken down into smaller subproblems until each piece becomes easy to solve. This strategy should decrease the program development time and its maintenance cost.

*Documentation* is another important concept. It provides invaluable clarification for complex programming statements, which will again facilitate future maintenance operations. Every programming language offers facilities to include documentation lines that are ignored by the compiler or interpreter but can be easily read and understood by programmers.

Also of key importance is to assign meaningful names to programming concepts such as variables. Instead of naming a variable *i* or *j* without any explicit interpretation, it is much better to use *student* or *course*, which immediately indicate their meanings.

By incorporating these principles into your programs, you will improve your own work and at the same time make it possible for others (or even yourself—it's not always easy to remember what you meant by `varX` months later) to update and continue using your software. After all, the goal is to create something useful that people will want to keep using.

That being said, let's immerse ourselves further into the wonderful world of Java programming and continue with the next chapter!

