Chapter 1: Foundations

Java is a powerful object-oriented programming language introduced by Sun Microsystems in 1995, which has built-in support to create programs with a graphical user interface (GUI), utilize the Internet, create client-server solutions, and much more. Programs written in Java can run, without change, on any of the common computer operating systems Windows 95/NT, Macintosh, and Unix. A variant of Java programs called applets can be embedded inside a web page and execute on the computer that is viewing the page, automatically and in a secure environment.

As a language, Java is closely related to C++, which is also object-oriented but retains a lot of idiosyncrasies inherited from its predecessor language C. Java has removed the inconsistent elements from C++, is exclusively object-oriented, and can be considered a modern version of C++.[1] Because of its logical structure Java has quickly become a popular choice as a teaching language,[2] and because of its extensive Internet support and the promise of writing programs once and using them on every operating system Java is becoming more and more accepted in industry.

There are a variety of commercial Integrated Development Environments (IDE) available to create Java programs, such as *Visual J++* from Microsoft or *Java Workshop* from SUN Microsystems. This text focuses on using the basic Java Developer's Toolkit (JDK), which is available free of charge over the Internet at `www.javasoft.com`. That toolkit contains the Java compiler, the Java Virtual Machine (JVM) necessary to run Java programs, extensive documentation, and a variety of utilities helpful for creating Java programs. The only additional tool you need is a convenient text editor to create Java source code such as *Programmer's File Editor* (for Windows 95/98/NT), *BBEdit Lite* (for Macintosh), or *Emacs* (for Unix). With these tools everything that is necessary to create Java programs can be downloaded for free from the Internet.[3]

This chapter presents the fundamentals of programming in Java. Section 1.1 introduces the Java compiler and Virtual Machine and shows the basic steps involved in creating, compiling, and executing a Java program. Section 1.2 defines basic data types, assignments, and arithmetic, while section 1.3 describes control structures such as conditional execution and loops. Section 1.4 describes the data types `String` and `StringBuffer`, which are used to handle text. Section 1.5 illustrates how to define constants and how to obtain output including simple formatting options. The last section, which is optional, provides a case study dealing with theoretical and numerical aspects involved in computing prime numbers.

We are not introducing object-oriented programming until chapter 3 but even the simplest Java program is a class rather than a program, so we *are* using object-oriented programming even if we do not formally define it until later.

---

[1] Java does have disadvantages. For example, programs written in Java are generally slower than those in C++ and it is difficult to accomplish system-level tasks in Java.
[2] Java compilers and tools are available for free, an important consideration for academic and student budgets.
[3] Appendix A and B discuss how to download and install the JDK and the various tools. If you have not installed the JDK on your computer, you may want to read these appendices before continuing.

Quick View

Here is a quick overview of the topics that will be covered in this chapter.

1.1. The Java Compiler and the Java Virtual Machine
Compiling a Java Program or Class; Executing a Java Program or Class; The Java Virtual Machine (JVM)

1.2. Data Types, Assignments, and Arithmetic
Basic Data Types and Representation; Declaring Variables and Storing Values; Arithmetic with Numeric Types; Shortcuts and Type-Casting; Logic and Comparison

1.3. Basic Program Control
Conditional Execution: `if` and `switch`; Loop Control: `for`, `while`, and `do`

1.4. Strings
What is a `String`; `String` Operations; `StringBuffer` and Its Operations

(*) 1.5. Output and Constants
Simple Screen Output; Formatting Decimal Numbers; Named Constants

(**) Case Study: Java Primes and the Prime Number Theorem

(*) This section is optional but recommended.
(**) This section is optional

1.1. The Java Compiler and the Java Virtual Machine

To create a Java program, you first create the *source code*, a text file containing your program written according to the Java language specifications. You can use any text editor for that such as *Notepad* (on Windows), *SimpleText* (on Macintosh), or *vi* (on Unix), or – much preferred – one of the programs recommended in appendix B. There are a few basic principles that must be followed:

Basic Java Programming Guidelines

Every Java program must follow these guidelines:

- *Java is case sensitive, i.e. the word* `Program` *is different from* `program`.
- *Curly brackets* `{` *and* `}` *are used to group statements together.*
- *An executable Java program must contain at least the following lines as a framework:*

```
public class Name
{  public static void main(String args[])
   {  ... program code ...
   }
```

```
    }
```

- *Every statement whose next statement is not a separate group must end in a semicolon.*
- *A Java program containing the above framework must be saved using the filename* `Name.java`*, where* `Name` *(including correct upper and lower cases) is the word that follows the keywords* `public class` *and the file extension is* `.java`*.*

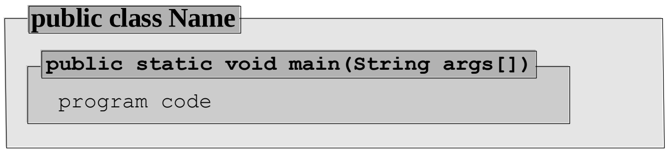


Figure 1.01: Representation of a basic Java program saved as `Name.java`

In other words, to create a Java program you first create a text file containing the lines

```
public class Name
{  public static void main(String args[])
   {  ... more lines ...
   }
}
```

Software Engineering Tip: While it is possible to write a program in one long line, a *good* Java program should be indented to clarify which line belong to which group. We use the following convention:[4]

- A block starts with the open curly bracket, followed by Java code.
- A block ends with a single closing curly bracket on a line by itself unless the brackets enclose a single statement.
- Lines that are part of the same block have the same level of indentation.
- Blocks within blocks are indented at increasing levels of indentation.

The file containing our code is called the source code file.

Source Code

A Java source code file is a text file that contains programming code written according to the Java language specifications, resembling a mixture of mathematical language and English. A computer cannot execute source code, but humans can read and understand it.

Java source code files should be saved as `Name.java`*, where* `Name` *is the name that appears in the first line of the program:* `public class Name`*. That* `Name` *is referred to as the name of the class, or program. By convention its first letter is capitalized.*

[4] Different authors prefer different styles of indentation. Some authors, for example, use a style similar to:
```
public static void main(String args[]) {
... }
```

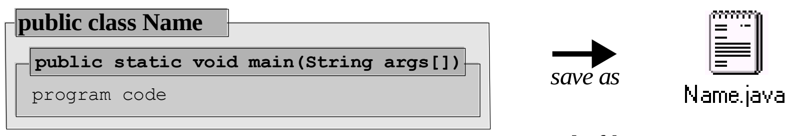


Figure 1.02: Saving a Java source code file

Here is an example of a Java source code file. We will later explain what the various lines mean; for now it is simply a text file that looks as shown.

Example 1.01: The first source code file

Create a source code file containing the necessary Java code to get the computer to write `"Hi - this is my first program"` on the screen.

Our first Java program looks as follows:

```
public class Test
{  public static void main(String args[])
   {  System.out.println("Hi - this is my first program");
   }
}
```

This program, or class, is called `Test` and must be saved under the file name `Test.java`.

■

Compiling a Java Program or Class

A source code file, which is more or less readable in plain English, needs to be transformed into another format before the computer can act upon it. That translation process is called *compiling* and is accomplished using the Java compiler `javac` from the Java Developer's Kit (JDK).

Compiling

Compiling is the process of transforming the source code file into a format that the computer can understand and process. The resulting file is called the byte-code, or class, file. The name of the class file is the same as the name of the program plus the extension `.class`. *The program* `javac` *from the Java Developer's Kit is used to transform a source code file into a class file.*

Under Windows and Unix, execute the compiler from the command line by typing `javac FileName.java`.[5] *On a Macintosh, double-click the* `javac` *compiler icon and select the source code file.*

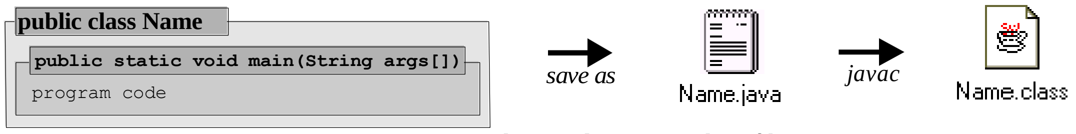


Figure 1.03: Compiling and creating class file

[5] See appendix B for more convenient ways of compiling source code.

Example 1.02: Compiling a source code file

Use `javac`, the Java compiler, to transform the source code file from example 1.01 into the corresponding class file.

The Java compiler is itself a computer program with a minimal interface. Under Windows and Unix, you execute it from the command line by typing:

```
javac Test.java
```

Using a Macintosh, you double-click on the Java compiler icon, then select the file `Test.java`.

The Java compiler checks your source code for compliance with the Java language specifications and if there are no errors it produces a file named

`Test.class`[6]

If the compiler detects errors, it displays corresponding error messages including the line number where an error occurred and a brief explanation. You need to return to your source code file, fix the errors, save the updated file, and compile it again.

■

Before executing a Java class file we take a look at what happens when the Java compiler encounters an error.

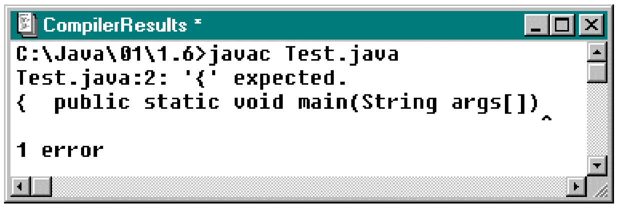
Example 1.03: Compiling source code with errors

Create a Java source code file that contains incorrect code, then compile it. Notice the error message. Fix your code until it compiles without any error messages.

We create a file similar to the above, but containing some errors:

```
public class Test
{  public static void main(String args[])
      system.out.println("Hi - this is my first program");
   }
}
```

The code contains two errors but when we compile it, the compiler finds only one, a missing beginning bracket in line 2, as shown in figure 1.04.



```
C:\Java\01\1.6>javac Test.java
Test.java:2: '{' expected.
{  public static void main(String args[])
                                         ^

1 error
```

Figure 4: Compiling `Test.java` *containing errors*

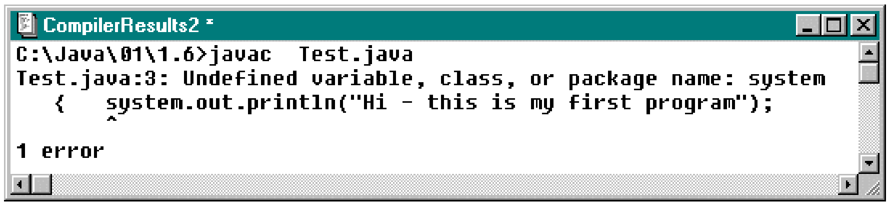
We add that bracket according to our basic framework for Java programs:

```
public class Test
{  public static void main(String args[])
```

[6] The Java compiler does not create an object file, which is linked with other object files to form an executable program, as other compiler do. It creates a byte-code file that can be *interpreted* by the Java Virtual Machine.

```
        {  system.out.println("Hi - this is my first program");
        }
    }
```

After saving and compiling the modified source code we expect no further problems, but the compiler informs us that the word `system` in line 3 is incorrect (see figure 1.05).



```
C:\Java\01\1.6>javac  Test.java
Test.java:3: Undefined variable, class, or package name: system
   {   system.out.println("Hi - this is my first program");
       ^

1 error
```

Figure 1.05: Compiling with lexical error

Indeed, we should have typed `System.out.println(...)` instead of `system.out.println(...)`, because Java is case sensitive. After fixing that error the compiler creates the class file as before.

■

Software Engineering Tip: In case of an error, the `javac` compiler shows the line number and position of where *it thinks* the error occurred in your source code.

- If the compiler points out an error, then there is an error *at or before* the indicated position.
- If the compiler reports a certain number of errors, than this is *the least* amount of errors.
- If one error is fixed, other errors may automatically disappear or new ones may appear.

Fix your source code a few errors at a time. Recompile often to see if the number of errors and the error messages change until no errors are reported. If you can not find an error at the position indicated by the compiler, look at the code *before* that position.

Executing a Java Program or Class

The Java compiler does not produce an executable file, so Java programs can not execute under the operating system of your machine. Instead they execute inside a *Java Virtual Machine*, which is invoked using the `java` program of the JDK.

Executing a Class File

To execute a Java program the Java Developer's Kit provides a program called `java`*. When executing that program with your class file as parameter the following happens:*

- *the Java Virtual Machine (JVM) is created inside your computer*
- *the JVM locates and reads your class files*
- *the JVM inspects your class file for any security violations*
- *the JVM executes, or interprets, your class file according to its instructions if possible*

Under Windows and Unix, execute a program by typing at the command prompt `java Name`*, where* `Name` *is the name of the program (no extension). On a Macintosh, double-click the* `java` *icon and select the appropriate class file.*[7]

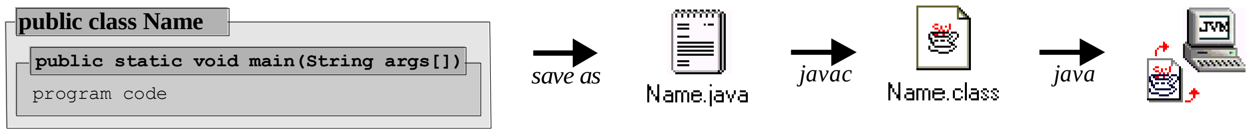


Figure 1.06: Executing a class file

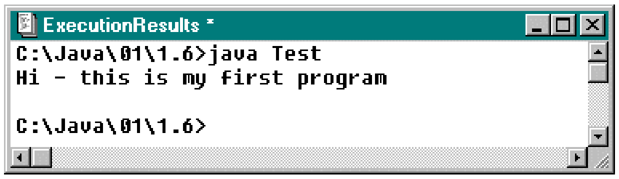
Example 1.04: Executing a class file

In example 1.02 we created a source code file `Test.java` and compiled it into the class file `Test.class`. Use the `java` program from the JDK to execute that program.

To execute the above program, we type (on a Macintosh, choose the appropriate icon):

```
java Test
```

i.e. the command `java`, followed by the name of a class file without the `.class` extension. The output of our program contains no surprises and is shown in figure 1.07.



```
C:\Java\01\1.6>java Test
Hi - this is my first program

C:\Java\01\1.6>
```

Figure 1.07: Our first functioning Java program

■

A good question at this point is which line in a Java program executes *first*.

Default Program Entry Point

The default program entry point is that part of a class (or program) where execution begins. For every Java class (or program), the standard program entry point consists of the line:

```
public static void main(String args[])
```

If that line is not present in your source code, the JVM can not execute your program and displays an error message.

Example 1.05: Executing class file without **`main`**

Write a small Java program similar to `Test.java` program in example 1.02, but misspell the word `main` slightly. Does the program compile? Does the program execute?

Here is a sample program, which is the same as before but instead of `main` we type `Maine`:

```
public class Test2
{   public static void Maine(String args[])
```

[7] See appendix B for more convenient ways of executing class files.

```
      {   System.out.println("Hi - this is my first program");
      }
   }
```

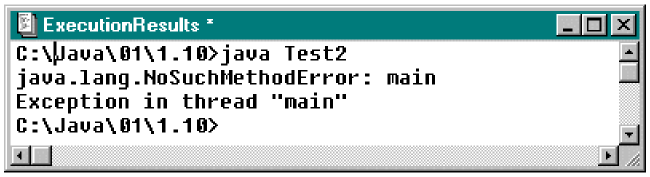
After saving this file as `Test2.java` we can compile it as follows:

```
javac Test2.java
```

The program compiles without errors but if we try to execute it by typing

```
java Test2
```

we see the (run-time) error message shown in figure 1.08.



```
C:\Java\01\1.10>java Test2
java.lang.NoSuchMethodError: main
Exception in thread "main"
C:\Java\01\1.10>
```

Figure 1.08: Compling `Test2.java` *with incorrect* `public static void main` *line*

■

This means that while the class file contains no compiler errors, it is missing the default program entry point `public static void main` and the Java Virtual Machine can not start the program.

The Java Virtual Machine (JVM)

At this point, we need to explain what the Java Virtual Machine is and how it relates to the operating system and to Java class files.

Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is a platform-independent engine used to run Java applets and applications. The JVM knows nothing of the Java programming language, but it does understand the particular file format of the platform and implementation independent class file produced by a Java compiler. Therefore, class files produced by a Java compiler on one system can execute without change on any system that can invoke a Java Virtual Machine.[8]

When invoked with a particular class file, the JVM loads the file, goes through a verification process to ensure system security, and executes the instructions in that class file.

The JVM, in other words, forms a layer between the operating system and the Java program that is trying to execute. That explains how *one* Java program can run without change on a variety of systems: it can not! A Java program runs on only *one* system, namely the Java Virtual Machine. That virtual system, in turn, runs on a variety of operating systems and is programmed quite

[8] In general, the Java Virtual Machine is an abstractly specified class file interpreter that can be realized by different software makers. The JVM that comes with the JDK was created by SUN Microsystems, but any other JVM is also able to run the same class files. Different JVM's can vary in efficiency, but they all must run the same class files.

differently for various systems. To the Java programmer, it provides a unified interface to the actual system calls of the operating system.[9]

You can include graphics, graphical user interface elements, multimedia, and networking operations in a Java program and the JVM will negotiate the necessary details between the class file(s) and the underlying operating system. The JVM produces exactly the same results – in theory – regardless of the underlying operating system. In the Basic (or C, or C++) programming language, for example, you can create code that specifies to multiply two integers 1000 and 2000 and store the result as another integer. That code works fine on some systems, but can produce negative numbers on others.[10] In Java, this can not happen: either the code fails on all platforms, or it works on all platforms.

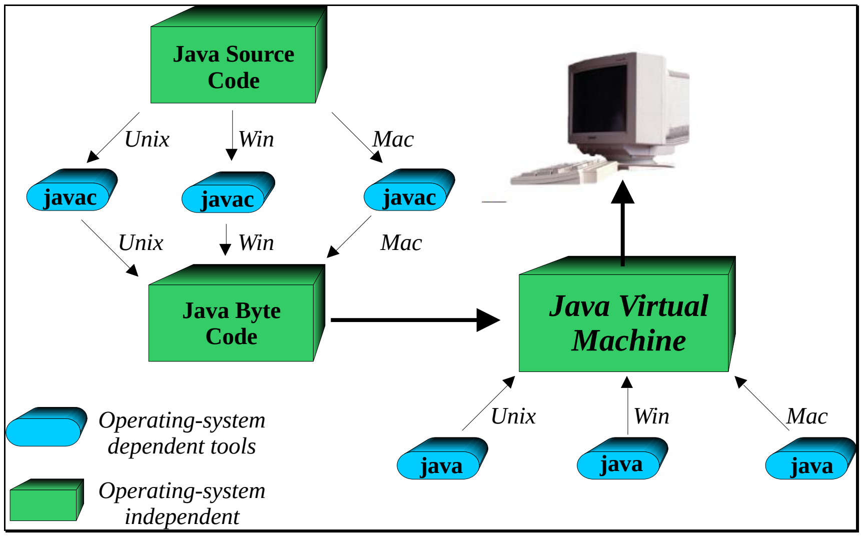


Figure 1.09: Illustrating the machine dependent/independent parts of Java programs

Because the JVM is in effect its own computer, it can shield the actual computer it is running on from potentially harmful effects of a Java program. This is especially important because Java programs known as *applets* can *automatically* start executing on your machine when you are surfing the web if the appropriate feature of your web browser is enabled. If these programs were allowed to meddle with your system, you could accidentally execute a program that would proceed to erase your entire disk. That, of course, would prompt people to disable Java on their web browser, which in turn would be bad news for anyone who supports the Java concept.

Fortunately, Java provides advanced security mechanisms to stop unwanted programs doing any damage on your system and since Java programs do not run directly on your operating system, security can be tightly enforced. There have been occasional rumors about mischievous Java programs distributed through the web, but so far there have been no reports of any actual damage from such programs. SUN Microsystems and others are constantly on the lookout to find any potential security holes and to plug them as soon as they are discovered.

[9] This is somewhat similar to old Basic programs: a simple Basic program can run on virtually any system that has a Basic interpreter installed since the interpreter mediates between the program trying to run and the operating system.
[10] Programming languages have a *largest possible* integer whose value can differ on different systems. A C++ program executing on a machine with a largest integer bigger than 2,000,000 produces the correct result, but on a system where the largest integer is, say, 32,767 it fails. The JVM has the *same* largest integer on every platform.

Security

The JVM enforces tight security and can effectively shield the operating system from unwanted effects of a Java program. Before the virtual machine executes a class file, it undergoes a multi-pass verification process to ensure system security, including verification of the correct class file format and a data flow analysis.

In addition, the JVM imposes run-time security restrictions that depend on whether it executes an applet (strict security) or an application (relaxed security).[11]

To summarize, Java class files are not stand-alone executable programs that run on the operating system, but instead they are interpreted by the JVM which in turn runs on the operating system. To create a class file, a Java source code file is compiled using the Java compiler `javac`. To execute the file, the JVM needs to be invoked with the class file as input. The JVM looks for the default program entry `point public static void main` and begins execution at that point if the class file is of valid format.

1.2. Data Types, Assignments, and Arithmetic

Java programs can store data in basic data types or in more complex types called *classes*. We can also define new types with any degree of complexity, which we explain in chapter 3. For now, we focus on the basic Java data types and how they are represented inside the computer.

Basic Data Types and Representation

Primitive Java Data Types

Java supports the following primitive, or basic, data types:

• `int`, `long`, *or* `short`	*to represent integer numbers*
• `double` *or* `float`	*to represent decimal numbers*
• `char`	*to represent character values*
• `boolean`	*to represent logical values*
• `void`	*to represent "no type"*

Each numeric type has a largest and smallest possible value, as indicated in table 1.10.[12]

Most programs use `int` for integers and `double` for decimal numbers, while `long`, `short`, and `float` are needed only in special situations.

Type	Range	
`double`	largest positive/negative value:	±1.7976931348623157E308
	smallest non-zero value:	±4.9E-324

[11] See chapter 9.5 for additional information about the restrictions imposed by the Java security manager.

[12] Java also supports a basic type `byte`, which we do not need in this text, and another very useful type `String`, introduced in section 1.4.

	significant digits:	16 digits after decimal point
`float`	largest positive/negative value:	±3.4028235E38
	smallest non-zero value:	±1.4E-45
	significant digits:	8 digits after decimal point
`int`	largest value	2147483647
	smallest value:	-2147483648
`short`	largest value	32767
	smallest value:	-32768
`long`	largest value	9223372036854775807
	smallest value:	-9223372036854775808

Table 1.10: Ranges for valid decimal types

Each type can contain values called literals or unnamed constants in a particular format.

Literals

Literals are constant values for the basic data types. Java supports the following literals:

- `int, short:` *digits only, with possible leading plus (+) or minus (-) sign*
- `long:` *like* `int` *literals, but must end with an "*`L`*"*
- `double:` *digits including possible periodic point or leading plus (+) or minus (-) sign, or numbers in scientific notation* `#.###############E±###`*,*[13] *where each* `#` *represents a digit*
- `float:` *like* `double` *literals, but must end with an "*`F`*"*
- `char:` *Single Unicode characters enclosed in single quotes, including the special control sequences described in table 1.11*[14]
- `boolean:` `true` *or* `false`

In addition, Java has an object literal called `null` *for object references.*

Character literals include the following special characters called control sequences:

Control Sequence	Meaning	Control Sequence	Meaning
\n	new line	\t	tab character
\b	backspace	\r	return
\f	form feed	\\	backslash
\'	single quote	\"	double quote

Table 1.11: Common character control sequences

The ranges for the numeric types are the same, regardless of the underlying operating system (after all, programs run under the JVM, not the native operating system). In languages such as C or C++ an integer sometimes has a range similar to a Java `short`, and sometimes that of a Java `int`, depending on the underlying operating system, which can cause different results if the same program runs on different systems.

While the JVM is a virtual machine, it is nonetheless running on concrete computer hardware. At the base of that hardware are *transistors*, which are essentially lots of on/off switches. Therefore everything that a computer does must eventually be represented by sequences of on/off switches,

[13] For example, the `double` number `1.23456E002 = 1.234562 = 123.456`

[14] Unicode characters support characters in multiple languages and are defined according to their "Unicode Attribute table" (see `http://www.unicode.org/`). Every character on a standard US keyboard is a valid Unicode character.

which raises the question how sequences of on/off switches can store `int`, `double`, `char`, and `boolean` values. That is easiest to understand for `boolean` values:

- Reserve one switch per `boolean` value to store and interpret "off" as `false`, "on" as `true`.

To determine how to store an `int` such as 274 using a sequence of on/off switches you may recall from you math classes that every positive integer can be uniquely represented as a sequence of 1's and 0's.

Binary Representation

Take any positive integer `x`*. Then* `x` *can be represented as a finite sum of powers of* `2` *in the form* $x = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_1 2^1 + a_0 2^0$*, where each* a_j *is either* `0` *or* `1` *and* `n` *is a suitable integer. The sequence* $a_n, a_{n-1}, a_{n-2}, \ldots, a_2, a_1, a_0$ *is called the binary representation of the integer* `x`*, i.e. a representation that requires only* `0`*'s or* `1`*'s. We sometimes write such a binary representation as* $(a_n\ a_{n-1}\ \ldots\ a_2\ a_1\ a_0)_2$ *to indicate that the number in parenthesis is in binary form.*[15]

To see how this definition works, we convert some numbers from their standard base-10 representation into binary form.

Example 1.06: Converting integers to binary form

Convert the numbers 44 and 63 to their binary representation, and the binary numbers $(100010010)_2$ and $(1101011)_2$ into standard form.

To find the binary representation of the number `44` we write down a few powers of 2 in a table:

64 = 2^6	**32** = 2^5	**16** = 2^4	**8** = 2^3	**4** = 2^2	**2** = 2^1	**1** = 2^0

The highest power of 2 that fits into 44 is 32, with 12 remaining, so we enter a 1 into the "32" slot:

64 = 2^6	**32** = 2^5	**16** = 2^4	**8** = 2^3	**4** = 2^2	**2** = 2^1	**1** = 2^0
	1					

Since the remainder is 12, the next power of two that fits that remainder is 8, with a new remainder of 4. We therefore enter a 1 into the "8" slot:

64 = 2^6	**32** = 2^5	**16** = 2^4	**8** = 2^3	**4** = 2^2	**2** = 2^1	**1** = 2^0
	1		1			

The last remainder 4 is itself a power of 2, so that we enter a 1 into the "4" slot. The other powers of 2 are not used, so we enter 0 into their slots:

64 = 2^6	**32** = 2^5	**16** = 2^4	**8** = 2^3	**4** = 2^2	**2** = 2^1	**1** = 2^0
0	1	0	1	1	0	0

We have arrived at the binary representation $(101100)_2$ of 44 (dropping the leading 0), which can easily be confirmed: $1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 44$.

Similarly, the binary representation of 63 is $(111111)_2$:

[15] Our "usual" numbers can be uniquely represented as a finite sum of powers of 10. For example, the number 12,345 can be written as $1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0$ (which equals $(11000000111001)_2$).

64 = 2^6	**32** = 2^5	**16** = 2^4	**8** = 2^3	**4** = 2^2	**2** = 2^1	**1** = 2^0
0	1	1	1	1	1	1

The converse is also easy: to find out which integers are represented by $(100010010)_2$ and $(1101011)_2$ we enter them into appropriate tables, starting from the *right*.

256= 2^8	**128** = 2^7	**64** = 2^6	**32** = 2^5	**16** = 2^4	**8** = 2^3	**4** = 2^2	**2** = 2^1	**1** = 2^0
1	0	0	0	1	0	0	1	0
0	0	1	1	0	1	0	1	1

Then we add those powers of 2 with a non-zero entry: $(100010010)_2 = 2 + 16 + 256 = 274$ and $(1101011)_2 = 1 + 2 + 8 + 32 + 64 = 107$.

■

Since every positive integer can be converted to binary form, we combine a fixed number of on/off switches, interpret "off" as 0 and "on" as 1 and the sequence of switches can store an integer up to a maximum size.

Bits and Bytes

Computer memory is measured in bits and bytes. One bit is a unit that can store either a 0 or a 1. A group of 8 bits is called one byte.

Example 1.07: Number of bytes to store `boolean`*,* `int` *and* `long`

How many bits or bytes are necessary to store a `boolean`, `int`, and `long` value. How many `int` and `long` values could your computer theoretically hold in memory?

A `boolean` value has two states so it can be represented by one bit.

Java `int` values range from –2,147,483,648 to 2,147,483,647 which means that Java must be able to handle

$$2,147,483,647 + 2,147,483,648 + 1 = 4,294,967,296$$

different integers (including 0). That number is divisible by 2 and 4294967296 / 2 = 2147483648 is even again. If we keep dividing by 2 we find that

$$4,294,967,296 = 2^{32}$$

That implies that we need 32 on/off switches to represent 32 possible 0's or 1's to store 2^{32} `int` values. Therefore, to store any given `int` Java requires 32 bits, or 4 bytes. The number 1, for example, is stored as $(00000000000000000000000000000001)_2$, where all of the zeros are necessary so that the largest possible `int` can also be stored, including its sign.

Java `long` values can range from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 so there are

$$9,223,372,036,854,775,807 + 9,223,372,036,854,775,808 + 1 = 18,446,744,073,709,551,616$$

different `long` values (including 0). Again this number is a power of 2 and

$$18,446,744,073,709,551,616 = 2^{64}$$

Therefore we need 64 on/off switches to represent 64 possible 0's or 1's to store 2^{64} long values, or equivalently we need 64 bits, or 8 bytes, to store any given long value.

To find out how many `int` and `long` values a computer can store theoretically, we need to know the size of the computer's memory. The personal computer used to write this text has 94MB RAM, which is approximately 94 * 1,000,000 bytes. If 4 bytes are used to store one `int`, that computer can store

$$94 * 1,000,000 / 4 \approx 23,000,000$$

`int` values. Similarly, it could store approximately

$$94,000,000 / 8 \approx 12,000,000$$

`long` values. The actual amount of numbers to store is much lower, because the operating system and the JVM already occupy a substantial amount of memory.[16]

■

Now we understand how the computer stores `boolean`, `int`, and `long` values. Characters are stored by associating a unique integer with each character, so that characters, too, can be stored as a sequence of 0's and 1's. Decimal values are more complicated. Suffice it to say that they can be written in scientific notation and the base and exponent of that representation can be converted separately to 0's and 1's. An important consequence is that decimal values such as `double` are not stored exactly. Therefore what looks like 2.0 on a computer screen is only an approximation of that number.

Software Engineering Tip: The basic data types available in Java have advantages and disadvantages.

- Boolean and character values are cheap to store (each requires only a few bits), but their usefulness is limited because you can not do computations with them.
- Integer values have limited range but integer computations take place quickly.[17] In addition, integer values are always exact values.
- Double values have a large range but computations using double values are slower than integer computations. In addition, double values are not stored exactly.[18]

You should always pick the best possible data type for the problem you are solving.

Declaring Variables and Storing Values

To use the basic data types to store information, we must define variables that have one of these types:

Declaration of Variables

To declare a variable that can store data of a specific type, the syntax:

[16] Modern operating systems can swap unused portions of memory to disk, but an executing program together with the data it manipulates is usually kept in active memory.
[17] To multiply, for example, $(10101)_2$ by 2 amounts to appending a 0 at the end: $2 * (10101)_2 = (101010)_2$.
[18] The values `10.0 / 3.0` and `1.0 / 3.0 * 10.0` are mathematically equal, but not equal as `double` values.

```
type varName [, varName2,..., varNameN];
```

is used, where `type` *is one of the basic data types,* `varName` *is the name of the variable, and* `varName2, ..., varNameN` *are optional additional variables of that type. Variables can be declared virtually anywhere in a Java program.*

Variables must have a name and there are a few rules to follow when choosing variable names:

Valid Names for Variables

A variable name must start with a letter, a dollar sign '`$`*', or the underscore character '*`_`*', followed by any character or number. It can not contain spaces. The reserved keywords listed in table 1.12 can not be used for variable names. Variable names are case-sensitive.*

Java Reserved Keywords					
abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	false	final	finally
float	for	goto	if	implements	import
instanceof	int	interface	long	native	new
null	package	private	protected	public	return
short	static	super	switch	synchronized	this
throw	throws	transient	true	try	void
volatile	while				

Table 1.12: Reserved keywords in Java

Example 1.08: Declaring variables

Declare one variable each of type `int`, `double`, `char`, and two variables of type `boolean`.

This is an easy example. We declare our variables as follows:

```
int anInteger;
double aDouble;
char aChar;
boolean aBoolean, anotherBoolean;
```

■

Assigning a Value to a Variable

To assign a value to a declared variable, the assignment operator "=" is used:

```
varName = [varName2 = ... ] expression;
```

Assignments are made by first evaluating the expression on right, then assigning the resulting value to the variable on the left. Numeric values of a type with smaller range are compatible with numeric variables of a type with larger range (compare table 13). Variables can be declared and assigned a value in one expression using the syntax:

```
type varName = expression [, varname2 = expression2, ...];
```

The assignment operator looks like the mathematical equal sign, but it is different. For example, as a mathematical expression

$$x = 2x + 1$$

is an equation which can be solved for *x*. As Java code, the same expression

```
x = 2*x + 1;
```

means to first evaluate the right side `2*x + 1` by taking the current value of `x`, multiplying it by `2`, and adding `1`. Then the resulting value is stored in `x` (so that now `x` has a new value).

Value and variable types must be compatible with each other, as shown in table 1.13.

Value Type	Compatible Variable Type
`double`	`double`
`int`	`int`, `double`
`char`	`char`, `int`, `double`
`boolean`	`boolean`

Table 1.13: Value types compatible to variable types

Example 1.09: Declaring variables and assigning values

Declare an `int`, three `double`, one `char`, and one `boolean` variable. Assign to them some suitable values.

There are two possible solutions. Variables can be declared first and a value can be assigned to them at a later time:

```
int anInteger;
double number1, number2, number3;
anInteger = 10;
number1 = number2 = 20.0;
number3 = 30;

char cc;
cc = 'B';
boolean okay;
okay = true;
```

Alternatively, variables can be declared and initialized in one statement:

```
int anInteger = 10;
double number1 = 20.0, number2 = 20.0, number3 = 30;
char cc = 'B';
boolean okay = true;
```

■

Software Engineering Tip: Variables serve a purpose and the name of a variable should reflect that purpose to improve the readability of your program. Avoid one-letter variable names[19]. Do not reuse a variable whose name does not reflect its purpose.

Whenever possible, assign an initial value to every variable at the time it is declared. If a variable is declared without assigning a value to it, all basic types except `boolean` are automatically set to `0`, `boolean` is set to `false`, and all other types are set to `null`.**[20]**

[19] If a variable serves a minor role in a code segment (such as a counter in a loop) it can be declared using a one-letter variable name.

Declare variables as close as possible to the code where they are used. Do not declare all variables at once at the beginning of a program (or anywhere else).

Example 1.10: Using appropriate variable names

The code segment below computes the perimeter of a rectangle and the area of a triangle. Rewrite that segment using more appropriate variable names and compare the readability of both segments:

```
double x, y, z;
x = 10.0;
y = 20.0;
z = 2*(x + y);
w = 0.5 * x * y;
```

This code computes the perimeter `z` of a rectangle with width `x` and length `y` and the area `w` of a triangle with base `x` and height `y`, so the variable names should reflect that. In addition, variables should be assigned a value when they are declared, so the code segment should be rewritten as follows:

```
double width = 10.0, height = 20.0;
double perimeterOfRectangle = 2*(width + height);
double base = width;
double areaOfTriangle = 0.5 * base * height;
```

It is immediately clear that the formulas used are correct. Choosing appropriate variable names clarifies the code significantly and makes it easy to locate potential problems.

■

Arithmetic with Numeric Types

Example 1.10 uses variables and literals to perform computations so we should explain the arithmetic operations that Java supports:

Basic Arithmetic for Numeric Types

Java support the basic arithmetic operators `+` *(addition)*, `-` *(subtraction),* `*` *(multiplication),* `/` *(division), and* `%` *(remainder after integer division) for numeric variables and literals. The order of precedence is the standard one from algebra and can be changed using parenthesis.*

Each operator has a left and right argument and the type of the result of the computation is determined according to the rules outlined in table 1.14:

Left Argument	Right Argument	Result
int	int	int
int	double	double
double	int	double
double	double	double

Table 1.14: Resulting types of the basic arithmetic operations[21]

[20] The compiler may display an error message if it encounters variables that are not explicitly initialized.
[21] Similar rules apply to the other basic types `long`, `short`, and `float`.

Software Engineering Tip: Do not forget that *dividing two integers will result in an integer*. If an integer division does not work without a remainder, then integer division will result in the integer part of the answer, ignoring the remainder. This is often the root of hard-to-find programming errors, because the compiler will compile such lines fine, they will compute without error, but the answer may not be what is intended.

Example 1.11: Basic computations

Suppose we have variables declared and initialized as follows[22]:

```
int i1 = 10, i2 = 3;
double x1 = 8.0, x2 = 5.0;
```

What are the resulting values and types of the computations:

```
i1 * i2;
x1 / x2;
i1 / i2;
i1 - x1;
i1 % i2;
i1 % x1;
```

Based on table 14 we get the following answers:

```
i1 * i2 = 30,   an integer
x1 / x2 = 1.6,  a double
i1 / i2 = 3,    an integer (different from mathematical division)
i1 - x1 = 2.0,  a double
i1 % i2 = 1,    because i1 / i2 = 3 with remainder 1;
i1 % x1 = 2.0,  because i1 / x1 = 1.0 with remainder 2.0;
```

■

Java can, of course, perform more elaborate numerical arithmetic. To use the standard mathematical functions such as `sin` and `cos`, we must preface them with the word `Math` and a period, for reasons that will become clear in chapter 3.

Advanced Arithmetic for Numeric Types

Java provides the mathematical functions specified in tables 1.15 to 1.18. To use a mathematical function, it must be prefaced by the word `Math`*, followed by a dot, and the function's name. Each function returns the type* `double` *unless otherwise indicated.*

Tables 15 to19 show the mathematical functions available in Java.

Trig functions and their inverses	
`cos(number)`	cosine of `number` (input in radians)
`sin(number)`	sine of `number` (input in radians)
`tan(number)`	tangent of `number` (input in radians)
`acos(number)`	arc (inverse) cosine of `number`, between 0.0 and Pi
`asin(number)`	arc (inverse) sine of `number`, between -Pi/2 and Pi/2
`atan(number)`	arc (inverse) tangent of `number`, between -Pi/2 and Pi/2
`atan2(a, b)`	angle between x-axis and the vector (b, a), between –Pi and Pi.

[22] Since these variables do not have an inherent meaning, it is okay to use simple variable names

Table 1.15: Trigonometric functions and their inverses

Exponential and its inverse	
`exp(number)`	e = 2.718... raised to the power of `number`
`log(number)`	Natural logarithm (base e) of `number`

Table 1.16: Exponential and logarithm functions

Rounding and Approximation functions	
`ceil(number)`	Smallest (closest to negative infinity) value that is not less than `number` and is equal to a mathematical integer
`floor(number)`	Largest (closest to positive infinity) value that is not greater than `number` and is equal to a mathematical integer
`round(number)`	The value of `number` rounded to the nearest integer value. Returns `int` if `number` is `float`, and `long` if `number` is `double`

Table 1.17: Rounding and approximation functions

Other functions	
`abs(number)`	Absolute value of `number`; returns same type as `number`
`max(a, b)`	Greater value of `a` or `b`; returns same type as input values
`min(a, b)`	Smaller value of `a` or `b`; returns same type as input values
`pow(base, exp)`	The number `base` raised to the power of `exp`
`sqrt(number)`	Square root of `number`
`random()`	A random number between 0.0 and 1.0

Table 1.18: Miscellaneous functions

In addition, Java provides the following values as named constants[23]:

Mathematical Constants	
`Math.E`	The value that is closer than any other to e, the base of the natural logarithms.
`Math.PI`	The value that is closer than any other to π, the ratio of the circumference of a circle to its diameter.
`Double.POSITIVE_INFINITY`	A representation of positive infinity of type double
`Double.NEGATIVE_INFINITY`	A representation of negative infinity of type double
`Double.NaN`	A representation of "Not-a-Number"

Table 1.19: Mathematical (numerical) constants

Here is an example using some of the above functions and constants.

Example 1.12: Advanced computations

Write some Java code segments to compute:

1. The circumference and area of a circle with radius `3.0`.
2. The length of the hypotenuse of a right triangle given its base and height.
3. The angle in degrees, given that `rAngle` is the angle in radians.
4. The angle in radians, given that `dAngle` is the angle in degrees.
5. What is `exp(4000.0)`? What is `10.0/0.0`? What is `-10.0/0.0`? What is `0.0/0.0`?

1. The circumference of a circle with radius r is $2\pi r$ and its area is πr^2. In Java, we compute these values as follows:

```
    double radius = 3.0;
```

[23] Section 1.5 explains how to define named constants.

```
        double circumf = 2 * Math.PI * radius;
        double area = Math.PI * radius * radius;
```

We can also use the mathematical function `pow` to compute the square of the `radius`, as in:

```
        double area = Math.PI * Math.pow(radius, 2.0);
```

2. The length of the hypotenuse of a right triangle given its base and height is the square root of the sum of the squares of the base and of the height, or in a mathematical formula:

$\sqrt{x^2 + y^2}$, where *x* is the base and *y* is the height

In Java, we rewrite this as follows (we of course avoid using `x` and `y` as variable names):

```
        double hyp = Math.sqrt(base*base + height*height);
```

assuming that `base` and `height` have already been declared and initialized, or as:

```
        double hyp = Math.sqrt(Math.pow(base, 2.0) + Math.pow(height, 2.0));
```

3. Recall from trigonometry that angles can be measured in degrees (from `0` to `360`) or in radians (from `0` to 2π), where an angle of `360` degrees corresponds to an angle of 2π in radians. A simple related rates problem allows us to convert angles from one measurement to the other:

$\frac{rAngle}{2\pi} = \frac{dAngle}{360}$, where *rAngle* is in radians and *dAngle* is in degrees

Solving this for *dAngle* we get the mathematical conversion formula to convert an angle in radians to an angle in degrees as $dAngle = \frac{rAngle \cdot 360}{2\pi}$. Our first attempt to convert this to Java code might be:

```
        double dAngle = rAngle * 360.0 / 2 * Math.PI;
```

But that is *not correct* because division and multiplication have equal precedence. Java therefore performs this computation from left to right, taking the value of `rAngle`, multiplying by `360`, dividing by `2`, and then *multiplying* the result by π instead of dividing by it. That is not what we want so we need to use parenthesis:

```
        double dAngle = rAngle * 360.0 / (2*Math.PI);
```

4. Similarly, to convert angles the from degrees to radians we use the Java code:

```
        double rAngle = dAngle * 2 * Math.PI / 360.0;
```
[24]

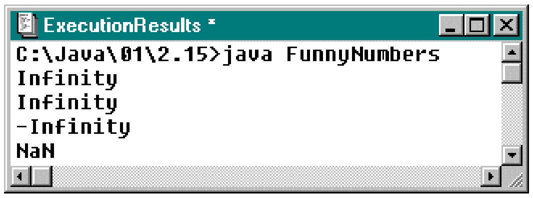
5. The value of `exp(4000.0)` must be a very large number, possibly larger than the largest `double`. Similarly, `10.0/0.0` is mathematically illegal because we can not divide by 0. It is not clear what Java makes of these expressions. but we can embed the code into a complete program, using the statement `System.out.println` to display the values on the screen.

```
        public class FunnyNumbers
        {  public static void main(String args[])
           {  double largeNum = Math.exp(4000.0);
              double posDivZero = 10.0 / 0.0;
              double negDivZero = -10.0 / 0.0;
```

[24] Both conversion formulas can be reduced but you should keep the formulas that are easiest to understand.

```
            double zeroDivZero = 0.0/0.0;
            System.out.println(largeNum);
            System.out.println(posDivZero);
            System.out.println(negDivZero);
            System.out.println(zeroDivZero);
        }
    }
```

When you type this file, save it under the name `FunnyNumbers.java`, compile it with the `javac` compiler, and run it by typing `java FunnyNumbers`, it will produce the results shown in figure 1.20.



```
C:\Java\01\2.15>java FunnyNumbers
Infinity
Infinity
-Infinity
NaN
```

Figure 1.20: Execution results for `FunnyNumbers`

This means that Java can, to some extend, deal with numbers that other run-time systems have problems with. It would be interesting to determine whether other operations can be performed with these values, i.e. can you add / subtract / multiply / divide the values `infinity` and `NaN`? We will leave this discussion as an exercise.

■

Shortcuts and Type-Casting

Certain types of computations occur so frequently that Java has created shortcut notations for them:

Increment and Decrement Operators

Java provides the increment operator `++` *and the decrement operator* `--`. *Both can be used in front of (prefix) or after a variable (postfix) of type* `int`, `double`, *or* `char`.

- *If used in front of a variable (prefix notation), first the value of the variable is incremented (or decremented), then the new value is used in the expression.*
- *If used after a variable (postfix notation), first the old value is used in an expression, then it is incremented (or decremented).*

Example 1.13: Prefix and postfix increment and decrement

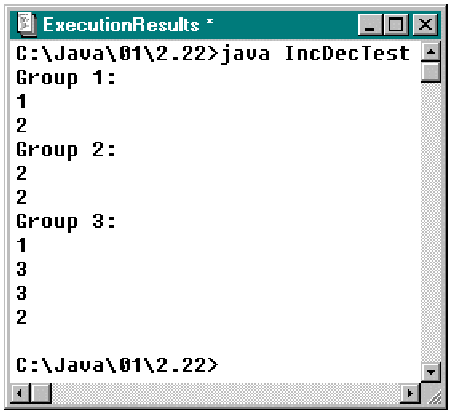
What is the output of the program below,:

```
public class IncDecTest
{  public static void main(String args[])
   {  int j, k;

      System.out.println("Group 1:");
      k = 1; j = k++;
      System.out.println(j);
      System.out.println(k);

      System.out.println("Group 2:");
      k = 1; j = ++k;
```

```
            System.out.println(j);
            System.out.println(k);

            System.out.println("Group 3:");
            k = 1;
            System.out.println(k++);
            System.out.println(++k);
            System.out.println(k--);
            System.out.println(k);
        }
    }
```

The code declares two integer variables `j` and `k` and is divided into three groups.

- The first group sets `k` to `1`, computes `j = k++` by first assigning the value of `k` (which is `1`) to `j`, then incrementing `k` by `1`. Therefore `j = 1 and k = 2`.
- The second group resets `k` to `1`, then `computes j = ++k` by first incrementing `k` by one, then assigning the result to `j`. Therefore `j` and `k` are equal to `2`.
- The third group resets `k` to `1`. The current value of `k` is displayed, then incremented by one. Then it increments `k` before using it and displays the new value (now `3`). Finally, that value is displayed (again `3`), then decremented by one, and in the last line the final value of `k` is displayed as `2`.



```
C:\Java\01\2.22>java IncDecTest
Group 1:
1
2
Group 2:
2
2
Group 3:
1
3
3
2

C:\Java\01\2.22>
```

Figure 1.21: Testing increment/decrement

■

Software Engineering Tip: The statement `k++` computes faster than the equivalent statement `k = k + 1`, but overusing the increment (decrement) operators results in code that is difficult to understand. Use increment (decrement) operators only in postfix notation, and do not use increment (decrement) and assignment operators in one statement.

In addition to the increment and decrement operators, Java provides the following shortcuts:

Computational Shortcut Notation

The following computational shortcuts can be used in Java:

```
aNumber += anotherNumber; <=> aNumber = aNumber + anotherNumber;
aNumber -= anotherNumber; <=> aNumber = aNumber - anotherNumber;
aNumber *= anotherNumber; <=> aNumber = aNumber * anotherNumber;
aNumber /= anotherNumber; <=> aNumber = aNumber / anotherNumber;
aNumber %= anotherNumber; <=> aNumber = aNumber % anotherNumber;
```

Example 1.14: Shortcut evaluation

What is the value of the variables after the following code executes:

```
int k = 10;
```

```
double x = 10.0, y = 20.0;

k /= 4;
x += 10.0;
y *= y;
```

The initial values of `k` and `x` are `10`, `y` has the value of `20`.

- The expression `k /= 4` is equivalent to `k = k / 4`, so it divides `k` by `4` and assigns the result to `k` again. Therefore, `k` is `2` (because of integer division).
- The expression `x += 10.0` adds `10.0` to the original value of `x`, then assigns that new value to `x`. Hence, `x` is `20`.
- The statement `y *= y` means `y = y * y`, so that `y` is assigned the square of its original value. Hence, y is `400.0`.

■

Software Engineering Tip: The `+=` and `-=` shortcuts are very common. Use them from the beginning to get acquainted to them.

Java also provides easy utility operations to convert data from one type into another one:

Simple Typecasting or Conversion

Typecasting is the process of changing the type of a variable or value. In general you can typecast `typeA` *into* `typeB` *if* `typeA` *is a special case of* `typeB` *using the syntax:*

```
(typeB)expressionOfTypeA;
```

Numeric types of larger range can be typecast into types with smaller ranges, but you will loose accuracy. Numeric values of smaller range are automatically typecast into types with larger range if necessary.

Example 1.15: Typecasting to `double`

Define two integers `int1` and `int2` and a double `aDouble`, then write some code segment that can divide `int1` by `int2` and store the mathematically correct result in `aDouble`.

We have already seen that if `int1 / int2` results in another integer. Storing that in a `double`:

```
double aDouble = int1 / int2;
```

does not work, because evaluation occurs first on the right side, producing an integer, then the result is assigned to `aDouble` and automatically typecast to a `double`. We must make sure that we force the integers to become doubles *before* we divide them:

```
double aDouble = ((double) int1) / ( (double) int2);
```

This first typecasts (converts) the integers to doubles, then performs a division with doubles, and finally stores the resulting `double` value in our variable `aDouble`.[25]

■

[25] In this case it is sufficient to typecast only one of the integers to a `double`. Alternatively, we could have multiplied one of the integers by a `double` before dividing: `(1.0 * int1) / int2`

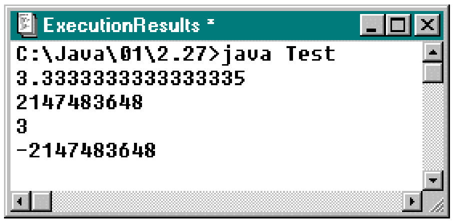
Using typecasting can have strange consequences.

Example 1.16: Typecasting large range to small range types

Below is a simple program that uses typecasting to convert large-range types to small-range types:

```
public class Test
{  public static void main(String args[])
   {  double aDouble = 10.0 / 3.0;
      long   aLong = 2147483647L + 1L;
      int    intFromDouble = (int)aDouble;
      int    intFromLong = (int)aLong;
      System.out.println(aDouble);
      System.out.println(aLong);
      System.out.println(intFromDouble);
      System.out.println(intFromLong);
   }
}
```

If you compile and execute this program, you will see the output below. Explain.



```
C:\Java\01\2.27>java Test
3.3333333333333335
2147483648
3
-2147483648
```

Figure 1.22: Unexpected type-casting results

The first two numbers are the computed values for `aDouble` and `aLong`. Because `double` values are not exact, the last digit is not mathematically correct. When we typecast `aDouble` into the integer `intFromDouble`, the decimals are cut off and the value of `intFromDouble` is `3`. As far as the value of `aLong` is concerned, its mathematical value is `2147483648`. That is within the range of a `long` but when type-cast into an `int` it is one more than the largest possible `int`. As a result, the value "wraps around" to the other side of the range of an `int`, and the value of `intFromLong` becomes `-2147483648`, which is the *smallest* possible `int` value and not at all what we expected.

■

Software Engineering Tip: Use explicit typecasting even if implicit (automatic) typecasting would work. In other words, use statements such as `double x = (double)(i * j)` instead of `double x = i * j`, assuming that `i` and `j` are integer types.

Avoid typecasting from larger-range numeric types into smaller-range types if possible, or use the mathematical functions `ceil`, `floor`, and `round` (see table 17) to clarify the potential loss of accuracy. In other words, use statements such as `int i = (int)Math.round(x)` instead of `int i = (int)x`, assuming that `x` is a `double` type.[26]

[26] If `long l = 2147483647L + 10L` then `(int)l` results in `-2147483639` while `(int)Math.round(l)` evaluates to `2147483647`. Both are mathematically incorrect, but the second version does at least not produce a negative number.

Logic and Comparison

Java can also perform computations with the literals `true` and `false` and with `boolean` variables and expressions using the operators `and`, `or`, and `not`.

Boolean Operators

To combine `boolean` *values and variables, Java provides the following operators:*

`&&` (*logical "and"),* `||` (*logical "or"),* `!` (*logical "not")*

Multiple logical statements connect by `||` *and* `&&` *are evaluated left to right unless parenthesis are used. Evaluation stops as soon as the resulting* `boolean` *value can be determined.*

The result of using `and` and `or` can be determined from table 1.23.

Operation	Expression A	Expression B	Result
A && B	A = true	B = true	true
	A = true	B = false	false
	A = false	B = true	false
	A = false	B = false	false
A || B	A = true	B = true	true
	A = true	B = false	true
	A = false	B = true	true
	A = false	B = false	false

Table 1.23: Truth table for `and` *and* `or` *operators*

Most of the time these operators are used to combine tests or comparisons so we need to know how to compare the basic data types.

Comparison Operators and Tests

To compare two numerical or character expressions, tests of the form `(expression operator expression)` *are used, where* `operator` *is one of the following:*

- `>` *tests whether the left side is larger than the right side*
- `>=` *tests whether the left side is larger than or equal to the right side*
- `<` *tests whether the left side is less than then right side*
- `<=` *tests whether the left side is less than or equal to the right side*
- `==` *tests whether the left side is equal to the right side*
- `!=` *tests whether the left side is not equal to the right side*

The result of a test is the `boolean` *value* `true` *or* `false`*. Tests can be combined with the operators* `&&` *(and),* `||` *(or), and* `!` *(not) and prioritized using parenthesis.*[27]

These operators are used in testing the relationship between two expressions, as in the following example:

[27] Characters are compared according to the order in the Unicode table (which is the same as alphabetical order for English letters, with capital letters before lowercase ones).

Example 1.17: Simple tests

Suppose we have defined `int x = 10`, `double y = 11`, and `double z = 12.0`. Which of the following tests return `true`?

```
(x < z)
(y >= z)
((x+1) == y)
((x / 3) == (z / 4))
```

The first test asks the question: is `10 (= x)` less than `12 (= z)` and therefore evaluates to `true`. The second test, similarly, evaluates to `false`. For the third test, first `(x + 1)` is evaluated to `11`, then checked for equality against the value of `y`. That comparison returns `true` even though one value is of type `int`, the other of type `double`. Finally, the last line contains `(x / 3)` which – because it is integer division – evaluates to `3`. Since that's the same as on the right side `(12 / 4 = 3)`, the test evaluates to `true`.

■

Software Engineering Tip: It is a common mistake to use the assignment operator `"="` to try to compare two expressions for equality instead of the double-equal operator `"=="`. The compiler will flag this as an error and will not compile the code.[28] Remember that `"="` looks like a *mathematical equal* symbol but stands for the assignment operator, whereas `"=="` is used exclusively to check expressions for equality.

Using the double-equal operator to check `double` values for equality can pronounce a mathematical equality as `false` because `double` values are inexact. For example, if `double x = 10.0/3.0` and `double y = 1.0/3.0*10.0`, then `(x == y)` evaluates to `false`! Avoid using double-equal comparisons for `double` values, use suitable inequalities instead, if possible.

The next example illustrates that evaluation of a `boolean` expression stops as soon as the final value can be determined.

Example 1.18: Tests with short-circuit* `boolean` *evaluation

Does the following code always work or will it generate a "run-time error" such as "division by zero" if `x` is equal to zero?

```
boolean okay = ( (x != 0) && (Math.abs(1/x) < 5));
```

If `x` is not zero, then `(x != 0)` evaluates to `true`. Since the connection is via `&&` (and) Java needs to evaluate the second part before being able to decide on the value of the entire expression.

If `x` is equal to zero then the `(x != 0)` results in `false`. Because of the `&&` (and) the entire expression must then evaluate to `false` regardless of the remaining part and Java does *not* perform the second test. No division by zero occurs.

■

Java also provides a comparison operator to check whether a variable is of a certain (non-basic) type.

[28] In C++, unfortunately, an expression such as `(x = y)` compiles but does *not* check `x` and `y` for equality.

The `instanceof` Operator

The `instanceof` *operator checks whether a reference variable is of a particular type. If so, it returns* `true`*, otherwise* `false`*. It is used as follows:*

```
(varName instanceof classType)
```

Both `varName` *and* `classType` *must be class (reference) types, not primitive types.*[29]

1.3. Basic Program Control

Until now our code samples have been sequential: the JVM starts executing a program at its default program entry point, then works through one line after the other until it reaches the last line in that group (and therefore in the program). Of course that's not all there is and Java provides several ways to control which lines should execute, under what conditions, and how often.

Basic program control falls into two groups: flow control and loop control. Flow control lets you include or exclude lines or blocks of code based on certain conditions while loop control governs how often blocks of code execute.

Block of Code

A block of code in Java consists either of a single Java statement or of a collection of statements grouped together using the grouping symbols `{` *and* `}`*. Lines belonging to the same block should be properly indented in your source code.*

Conditional Execution: The `if` and `switch` Statements

The basic flow control statement that Java provides is the `if` statement, which has three different forms. Here is the easiest incarnation:

The Simple `if` Statement

A simple `if` *statement specifies the condition under which a certain block of code executes, using the syntax*

```
if (condition)
   blockOfCode;
```

where `condition` *is an expression of* `boolean` *type, such as a test. If* `condition` *evaluates to* `true` *the* `blockOfCode` *executes, otherwise execution will skip that block. In either case, execution resumes immediately after the* `blockOfCode`*.*

[29] This operator will be useful only after introducing classes in chapter 3. It has no significance for the basic data types mentioned in this chapter and is listed here for completeness only.

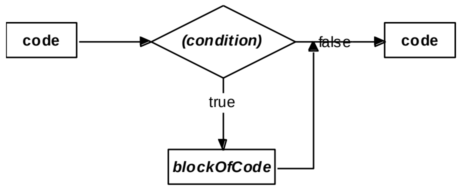


Figure 1.24: Representation of simple `if` statemet

We already know how to create logical tests resulting in a `boolean` value so here are some examples:

Example 1.19: Simple `if` statement

Assume that a `double` variable `x` has been declared and initialized to an unknown value. Write some code that will print the words "`x is positive`" on the screen if `x` is positive.

The logical test is clearly `(x > 0)` so the corresponding `if` statement looks like:

```
if (x > 0)
   System.out.println("x is positive");
```

There is only one line following the `if` statement so that no grouping brackets are necessary.

■

Example 1.20: Invalid simple `if` statement

Consider the following program code (where `x` has previously been declared as a `double` variable of unknown value):

```
if (x == 0.0)
   System.out.println("x is zero");
   System.out.println("Can not divide by x");
```

What is displayed if `x` is equal to `0.0`? How about if `x = 10.0`? Is this code what was (most likely) intended? If not, fix the code accordingly.

If `x` is equal to `0.0`, the test evaluates to `true` and the segment displays the following:

```
x is zero
Can not divide by x
```

If `x` is equal to `10.0`, the test for the `if` statement fails, skipping its code block. Since we are not using grouping parenthesis only the single line immediately following the `if` statement is considered part of the skipped block. Therefore, the screen shows:

```
Can not divide by x
```

But that is not correct, and probably not what the programmer intended. The fact that the last line is indented and *appears* to be part of the `if` statement is irrelevant, only grouping symbols can group statements together. The code, as the programmer probably meant it to write, looks as follows:

```
if (x == 0.0)
{  System.out.println("x is zero");
   System.out.println("Can not divide by x");
}
```

■

Another form of the `if` statement is used to make decisions in an "either-or" situation:

The `if-else` Statement with Alternative

An `if-else` *statement with alternative lets you use a* `boolean` *condition, usually a test, to determine which of two blocks of code executes, using the syntax:*

```
if (condition)
   blockOfCode1;
else
   blockOfCode2;
```

If `condition` *returns* `true`*, the* `blockOfCode1` *executes, otherwise* `blockOfCode2` *executes. After either of the two blocks has executed, the lines immediately after the* `else` *block continue to execute.*

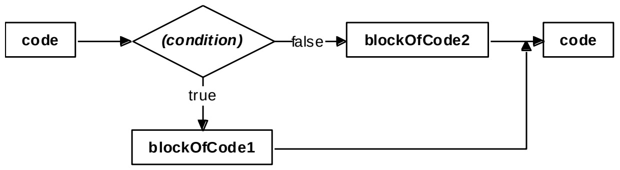


Figure 1.25: Representation of `if-else` *statement*

Example 1.21: Simple `if-else` *statement*

If `x` is a `double` whose value is not known, write some code that will print out "`x is positive`" or "`x is not positive`", depending on the value of `x`.

We have to decide between two alternatives so we use an `if-else` statement. Our test is `(x > 0)`, as before, and the code looks like this:

```
if (x > 0)
   System.out.println("x is positive");
else
   System.out.println("x is not positive ");
```

■

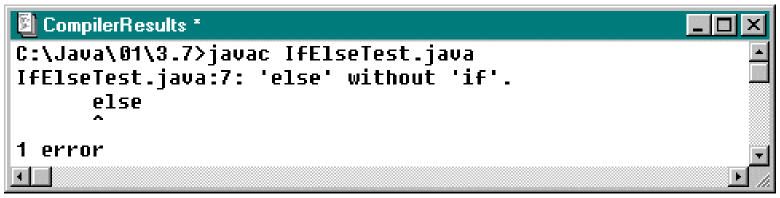
Example 1.22: Invalid `if-else` *statement*

Consider the following lines of code, where `x` and `y` have previously been declared as `double`:

```
if ((x*x - 10) >= 0.0)
   y = Math.sqrt(x*x - 10);
   System.out.println("y = " + y);
else
   y = 0;
   System.out.println("Can not take square root.");
```

What is the output if this code executes, assuming that `x = 2.0`?

This is a trick question, because a program containing that code does not execute. In fact, it does not even compile:



```
C:\Java\01\3.7>javac IfElseTest.java
IfElseTest.java:7: 'else' without 'if'.
      else
      ^

1 error
```

Figure 1.26: `if` *without* `else` *error message*

The compiler is complaining that the `else` is not part of an `if`, even though our code seems to have a matching one. The mistake is that the code did not use any grouping brackets and the compiler interprets it as a simple `if` statement instead of an `if-else` statement. The one-line code block `y = Math.sqrt(x*x - 10)` is part of the `if` statement and execution will resume at the line `System.out.println("y = " + y)`. Then the compiler encounters an `else` statement that does not belong to an `if`. To correct the code, we need to insert grouping brackets as follows:

```
if ( (x*x - 10) >= 0.0)
{   y = Math.sqrt(x*x - 10);
    System.out.println("y = " + y);
}
else
{   y = 0;
    System.out.println("Can not take square root.");
}
```

■

There is a final version of the `if` statement that allows choosing between multiple alternatives:

The Nested `if-else-if` Statement

A nested `if-else-if` *statement lets you choose which of* `N` *blocks of code plus one optional block of code executes, based on* `boolean` *conditions or tests. The syntax is:*

```
if (condition1)
   codeBlock1;
else if (condition2)
   codeBlock2;
...
else if (conditionN)
   codeBlockN;
[ else
   optionalBlock;]
```

If `condition1` *evaluates to* `true` *execute* `codeBlock1`*, otherwise check* `condition2`*. If* `condition2` *evaluates to* `true` *execute* `codeBlock2`*, otherwise check* `condition3`*. Continue in this fashion. If* `conditionN` *evaluates to* `true`*, execute* `codeBlockN`*, otherwise execute* `optionalBlock`*, if defined, or continue with regular execution.*

The k*th block of code executes if all conditions prior to the* k*th one evaluate to* `false` *and the* k*th one is* `true`*. The* `optionalBlock` *of code executes if all conditions evaluate to* `false`*.*

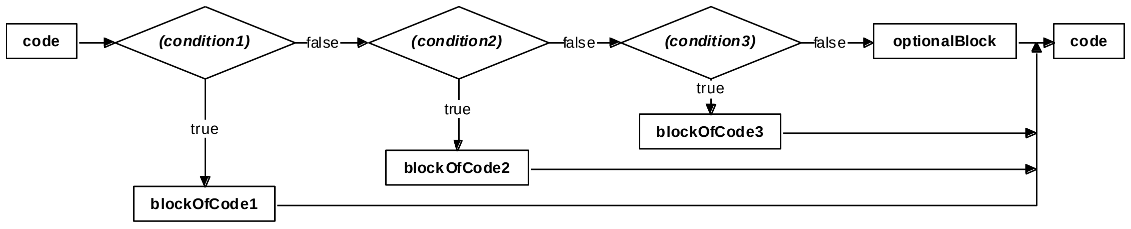


Figure 1.27: Nested `if-else-if` *statement with 3 code blocks and an optional block*

Example 1.23: Converting numeric scores to letter grades

Suppose a `double` variable `score` contains a number between `0` and `100`, reflecting a student's performance on a test. Based on that `score` write a code segment that displays the corresponding letter grade.

This is a situation where we have to choose among multiple alternatives. If the score is 90 or better, the letter grade is A, if the score is between 80 and 90, the letter grade is B, and so on. The code reflecting this situation could look as follows:

```
if (score >= 90)
   System.out.println("Letter grade: A");
else if ((score < 90) && (score >= 80.0))
   System.out.println("Letter grade: B");
else if ((score < 80) && (score >= 70.0))
   System.out.println("Letter grade: C");
else if ((score < 70) && (score >= 60.0))
   System.out.println("Letter grade D");
else if (score < 60)
   System.out.println("Letter grade: F");
```

This code is technically correct, but it contains redundant tests. Our definition states that the only way the *second* block of a nested `if-else` statement can execute is if the *first* condition evaluates to `false` and the second to `true`. Therefore the second condition does not need to test for `(score < 90)` because if the second condition is reached, the first one must have failed so that we know that at that point `score` must be less than 90. The same is true for the remaining tests so that the improved version of the code looks as follows:

```
if (score >= 90)
   System.out.println("Letter grade: A");
else if (score >= 80.0)
   System.out.println("Letter grade: B");
else if (score >= 70.0)
   System.out.println("Letter grade: C");
else if (score >= 60.0)
   System.out.println("Letter grade D");
else
   System.out.println("Letter grade: F");
```

■

Software Engineering Tip: It is easy to decide which form of an if statement applies:

- If something needs to happen or not, use a simple `if` statement
- If a decision between 2 alternatives is to be made, use the `if-else` statement
- If you need to decide between 3 or more alternatives, use the nested `if-else-if` statement

If you are using a nested `if-else-if` statement and the condition includes the `&&` (and) operator, you may be performing redundant tests and you should try to rephrase your statement, if possible.

Java provides an additional flow control statement called `switch`. That statement has a more complicated syntax and is one of the few language elements in Java that has not been properly modernized. Here is the definition and syntax:

The `switch` Statement

The `switch` *statement lets you choose between multiple alternatives using the syntax:*

```
switch (expression)
{  case value1: alternative1;
                break;
   case value2: alternative2;
                break;
   ...
   case valueN: alternativeN;
                break;
   [ default:   defaultAlternative;
                break; ]
 }
```

where `expression` *must be of type* `int` *or* `char`*. If* `expression` *equals* `value1`, `alternative1` *executes, if* `expression` *equals* `value2`, `alternative2` *executes, and so on. If no value matches* `expression`*, the* `defaultAlternative` *executes. Multiple distinct values can be handled using:*

```
case value1: case value2: ... case valueM: statement(s);
                                           break;
```

Example 1.24: Converting letter grades to numeric scores

Assuming that `grade` is a character variable containing a letter grade (A through F), create a code segment using a `switch` statement that displays the numeric range for that `grade`.

The expression that determines which of several alternatives should be chosen is of type `char` so a `switch` statement applies. We handle upper- and lowercase letters in one alternative using the code:

```
switch (grade)
{ case 'A': case 'a': System.out.println("90% or better");
                       break;
  case 'B': case 'b': System.out.println("between 80 and 89.99%");
                      break;
  case 'C': case 'c': System.out.println("between 70 and 79.99%");
                      break;
  case 'D': case 'd':
  case 'F': case 'f': System.out.println("less than 70%");
                      break;
  default:            System.out.println("invalid grade");
}
```

This `switch` statement will display, for example, "`90% or better`" if grade is either `'A'` or `'a'`.

■

Software Engineering Tip: A `switch` statement has a complicated syntax and is limited to handle distinct alternatives of `int` or `char` expressions. Since a nested `if-else-if` statement can always replace a `switch` statement, you should avoid using `switch` altogether.[30].

Example 1.25: Replacing `if-else-if` code by `switch`

Can a `switch` statement be used to replace the `if-else-if` code in example 1.23?

Recall the code for example 1.23, where `score` was of type `double`:

```
if (score >= 90)
   System.out.println("Letter grade: A");
else if (score >= 80.0)
   System.out.println("Letter grade: B");
else if (score >= 70.0)
   System.out.println("Letter grade: C");
else if (score >= 60.0)
   System.out.println("Letter grade D");
else
   System.out.println("Letter grade: F");
```

This code makes a decision based on the value of `score`. It can not be converted to a `switch` statement because `score` is not of type `int` or `char`. Even if `score` was of type `int`, a `switch` statement does not work, since it can only handle finitely many distinct possibilities, not ranges of numbers.

■

Example 1.26: Replacing `switch` by `if-else-if code`

Can a suitable `if-else-if` statement be used to replace the `switch` code in example 1.24? If so, provide that replacement code.

The following `if-else-if` statement has the same functionality as the `switch` statement in example 1.24:

```
if ((grade == 'A') || (grade == 'a'))
   System.out.println("90% or better");
else if ((grade == 'B') || (grade == 'b'))
   System.out.println("between 80 and 89.99%");
else if ((grade == 'C') || (grade == 'c'))
   System.out.println("between 70 and 79.99%");
else if ((grade == 'D') || (grade == 'd') ||
         (grade == 'F') || (grade == 'f'))
   System.out.println("less than 70%");
else
   System.out.println("invalid grade");
```

We are using the operator `||` with an `if-else-if` statement but no redundant tests are performed.

■

Loop Control: `for`, `while`, and `do`

[30] A `switch` statement can always be replaced by corresponding `if-else-if` blocks, but the converse is not true. Not every `if-else-if` statement can be converted to a `switch` statement.

The final basic programming structure we need is a *loop*. So far we can decide which code should execute based on the outcome of various tests, but we cannot execute code repeatedly without retype it. A simple program to display the numbers from 1 to 1000 would be a major exercise in typing at this stage.

Loop

A loop is a language structure that causes a block of code to repeat until certain conditions are met. A valid loop has three elements:

- *The test:* *a condition that determines whether a loop should continue or end.*
- *The initialization:* *the variable(s) used in the test must be initialized before the loop begins.*
- *The modification:* *the variable(s) used in the test should be modified inside the loop.*

Java offers three types of loop constructions. The simplest is a `for` loop:

The `for` Loop

A `for` *loop combines initialization, testing, and modification in one line, separated by semicolons. It applies if the number of times the loop should execute is known and has the syntax:*

```
for (initialization; test; modification)
   codeBlock;
```

where `initialization` *initializes a variable called the loop counter,* `test` *involves the loop counter in a* `boolean` *condition, and* `modification` *modifies the value of the loop counter.*

When the `for` *statement is first encountered,* `initialization` *is performed, then* `test` *evaluates. If* `test` *evaluates to* `true` *the* `codeBlock` *executes. At the end of* `codeBlock` *the* `modification` *is performed and* `test` *is evaluated again. The loop continues in this fashion until* `test` *is* `false`.[31]

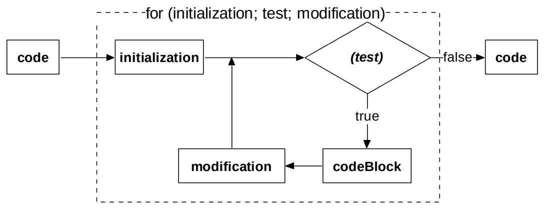


Figure 1.28: Representation of a `for` *loop*

The simplest type of `for` loop counts the number of times that `codeBlock` should execute:

Example 1.27: Simple `for` loop

Write a program that produces the statement "I like Java" `20` times (without, of course, writing that line `20` times "by hand").

We know we want our loop to execute `20` times, so one possible implementation is:

```
for (int i = 0; i < 20; i++)
```

[31] Initialization, test, and modification could be blank, but semicolons must be present. Initialization and modification could involve more than one variable (confusing): `for (int i=0, j= 0; (i*j<10); i++,j++) ...`

```
        System.out.println("I like Java");
```

We declare the variable `i` as an integer and initialize it to `0`. The test checks if `i` is less than `20`, in which case the loop continues. The modification part increments `i` by one each time the block executes so that the loop stops if `i` is incremented past `19`. Therefore the statement "I like Java" displays exactly `20` times, with `i` running from `0` to `19`.

Other `for` loops would work just as well:

```
for (int i = 1; i <= 20; i++)               for (int i = 200; i < 220; i++)
   System.out.println("I like Java");          System.out.println("I like Java");
```

For a complete program we embed one of these loops in the standard framework for Java programs:

```
public class SimpleCountingLoop
{  public static void main(String args[])
   {  for (int i = 0; i < 20; i++)
         System.out.println("I like Java");
   }
}
```

■

Software Engineering Tip: Do not modify the loop counter of a `for` loop *inside* the loop. It should only be changed in the `modification` part of the `for` statement. While the compiler does not have a problem with that, it makes your code difficult to understand.

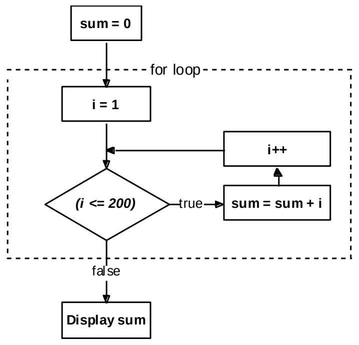
Loops can of course perform some useful action:

Example 1.28: Finding running total

Write a complete program to find the sum of the first `200` positive integers.

The program clearly involves a loop. We use a `for` loop with a loop counter `i` whose value runs from `1` to `200`. Before the loop starts, we declare a variable `sum` and initialize it to `0`. Each time the value of `i` changes in the loop, we add it to the running `sum` using the expression `sum = sum + i`, or more appropriately the equivalent shortcut notation `sum += i` (first the right side evaluates, then the result is stored back in `sum)`.

```
public class SumOfIntegers
{  public static void main(String args[])
   {  int sum = 0;
      for (int i = 1; i <= 200; i++)
         sum += i;
      System.out.println(sum);
   }
}
```



This program produces the number `20100` as the answer but we need to make sure that the answer is correct. We certainly do not want to repeat the calculation by hand but we can modify the program to add only the first 10 numbers. We can check that sum manually and if the program produces the correct answer we could assume it works correctly for larger sums.

That is an approach that is taken often to verify a program.[32] A program is executed for a few simple cases where the correct answer is known or can be computed manually. If the program produces the right answers in these test cases it is assumed that the answer for more complicated cases is correct as well.

In our case we actually *can* add the first 200 numbers by hand. We write all 200 numbers down twice (at least the first and last few numbers), once in increasing and again in decreasing order:

```
   1    + 2    + 3    + 4   + ...   + 197   + 198   + 199   + 200     = sum
 200  + 199  + 198  + 197   + ...     + 4     + 3     + 2     + 1     = sum
 201  + 201  + 201  + 201   + ...   + 201   + 201   + 201   + 201   = 2*sum
```

The numbers in each column add up to 201, and there are 200 columns. Therefore the third row adds up to 200 * 201, which in turn is twice the desired sum. Therefore:

2 * sum = 200 * 201, or equivalently, sum = 200 * 201 / 2 = 100 * 201 = 20100

We have indeed verified the answer that our program produced. Using the same technique it is easy to show that in general:[33]

The sum of the integers from 1 to N is equal to $\frac{N(N+1)}{2}$

■

Software Engineering Tip: The problem of finding a running total is very frequent in writing computer programs. The technique is almost always the same as the one used in example 1.28: initialize a variable such as sum outside the loop to 0 and employ a shortcut notation similar to sum += var inside the loop.

Example 1.29: Finding running totals with different step sizes

Write some code segments to find the sum of all positive even integers less than 500 and the sum of all positive integers divisible by three that are less than 500.

Each code segment involves a for loop and computes a running total. We could use the %, or "modulo" operator to determine if a number is even or not:

- If (i % 2) is 0, then i must be even, otherwise it must be odd.

Our first attempt to find the sum of the even numbers might be:

```
int evenSum = 0;
for (int i = 1; i < 500; i++)
{  if ( (i % 2) == 0)
      evenSum += i;
```

[32] Compare section 2.4.
[33] This formula is attributed to the mathematician F. E. Gauss. While Gauss was in grade school, a teacher asked his students to add the first 100 integers, expecting to keep the students busy for some time. Instead, Gauss was reported to come up with the above trick and produced the answer in a few seconds, much to the teacher's surprise.

```
    }
```

But odd numbers are not added to `evenSum` and the loop runs twice as long as necessary. In a `for` loop we can increment the loop counter variable any way we choose so we can rewrite this code as follows:

```
int evenSum = 0;
for (int i = 2; i < 500; i+=2)
   evenSum += i;
```

To find the sum of multiples of three that are less than 500, we could choose yet another alternative:

```
int threeSum = 0;
for (int i = 1; i < 500/3; i++)
   threeSum += (3*i);
```

But now we tried to be too smart and our code will produce a wrong answer. `500 / 3 = 166` because of integer division so that the last value added to `threeSum` is `165 * 3  = 495`. But the last number divisible by three that is less then `500` is `498` and that number will be missing from the sum. We could us a similar approach as for the even numbers to get it right, or simply replace the strict inequality < by <= in the test of the for loop (details left as an exercise).

■

Here is one last example that illustrate that a `for` loop may sometimes not execute at all, and sometimes a little too often:

Example 1.30: Over- or underperforming loops

How often do the code blocks inside each of the following for loops execute?

```
int x = 15;
for (int i = 1; ((i + x) < (x - i)); i++)
   System.out.println(i);

for (int i = 1; (i != 10); i+=4)
   System.out.println(i)
```

We trace the values of the variables and expressions manually to determine the answer:

Statement	Values
`int x = 15`	`x = 15, i` is unknown
(*initialization*) `int i = 1`	`x = 15, i = 1`
(*test*) `(i + x) < (x - i)`	`x = 15, i = 1, (16 < 14) = false`

Since the test immediately evaluates to `false` and is performed *before* the code block in the `for` loop executes, the `System.out.println` statement never executes.

Statement	Values
(*initialization*) `int i = 1`	`i = 1`
(*test*) `(i != 10)`	`i = 1, (1 != 10) = true`
`System.out.println(i)`	displays `1`
(*modifier*) `i += 4`	`i = 5`
(*test*) `(i != 10)`	`i = 5, (5 != 10) = true`
`System.out.println(i)`	displays `5`
(*modifier*) `i += 4`	`i = 9`

(*test*) `(i != 10)`	`i = 9, (9 != 10) = true`
`System.out.println(i)`	displays `9`
(*modifier*) `i += 4`	`i = 13`
(*test*) `(i != 10)`	`i = 13, (13 != 10) = true`
`System.out.println(i)`	displays `13`
etc.	etc.

The value of `i` is *never* equal to `10` so theoretically the loop never stops. On a computer, loops such as this will eventually terminate because the system is running out of resources (or someone turns the off switch). Such loops are called *infinite loops* and they are a common programming error.

■

Software Engineering Tip: Loops are frequently programmed incorrectly:

- The condition that should end the loop is never met, resulting in an "infinite" loop.
- The loop exceeds the number of times it is supposed to execute by one or more.
- The loop falls short by one or more in the number of times it is supposed to execute.
- The loop never starts executing, due to incorrect or missing initialization.

Your should always double-check whether a loop executes the intended number of times by checking the values of all variables manually for easy sample situations. Use inequalities instead of equalities in the test condition of a loop, especially if `double` values are involved.

The second and most flexible loop structure that Java offers is called a `while` loop.

The `while` Loop

A `while` *loop usually places any initialization before the actual loop and the modification statement inside the loop. The test together with the keyword* `while` *is the essential and mandatory part of the loop, using the syntax:*

```
while (test)
   codeBlock;
```

The `test` *is performed each time before* `codeBlock` *executes and must evaluate to* `true` *before* `codeBlock` *executes.*

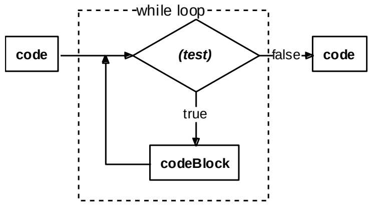


Figure 1.29: Representation of a `while` *loop*

Example 1.31: Replacing `for` *loop with* `while`

In example 1.29 we created a loop that found the sum of all positive even integers less than 500. Rewrite that code, using a `while` loop instead of a `for` loop.

Recall the prior code:

```
int evenSum = 0;
for (int i = 2; i < 500; i+=2)
   evenSum += i;
```

For a `while` loop we move the initialization `int i = 2` in front of the loop and the modification `i+=2` inside the loop. The `test` becomes part of the `while` statement:

```
int evenSum = 0;
int i = 2;
while (i < 500)
{  evenSum += i;
   i += 2;
}
```

■

Software Engineering Tip: A `while` loop can always replace a `for` loop so you could, in principle, use only `while` loops.[34] But a `for` loop is less complicated and more compact, therefore less error prone, so you should use a `for` loop whenever possible.

Here is a loop that is difficult to reproduce using a `for` statement:

Example 1.32: Testing for prime number

If `x` is an unknown, positive integer, write some code that tests if `x` is a prime number.

Recall that a number is prime if the only integers that divide it without remainder are `1` and itself. The easiest prime numbers are `1`, `2`, `3`, `5`, and `7`, but not `9`. To check whether an unknown number `x` is prime, we divide `x` by every integer between `1` and `x`. If the division yields a zero remainder, the number can not be prime. Here is our code segment (assuming that `x` has previously been declared and initialized):

```
int i = 2;
while ((i < x) && ( (x % i) != 0))
   i++;[35]
if (i >= x)
   System.out.println("The number is prime");
else
   System.out.println("The number is not prime");
```

In other words, we start with `i = 2` and continue our loop as long as `i < x` and the remainder after dividing `x` by `i` is not `0`. Inside our loop we increment `i` by one. The loop exits for two reasons:

`(i >= x)` or `(x % i) == 0`

Since `i` is incremented by one inside the loop, it will end for sure because of the first condition. When the loop is over, we check which of the two conditions made it stop:

- If `(i >= x)` after the loop exits, we have *not* found a number `i` that is less than `x` and gave a zero remainder, so that `x` must be prime.

[34] A `for` loop in Java can replace a `while` loop because `initialization` and `modification` can be empty.

[35] This loop converted to a `for` loop could look as follows:

```
int i = 2;
for ( ; ((i < x) && ( (x % i) != 0)); i++);
```

- If `(i < x)` after the loop exits, `x` must have been evenly divisible by some `i` that is less than `x` so that `x` can not be prime.

■

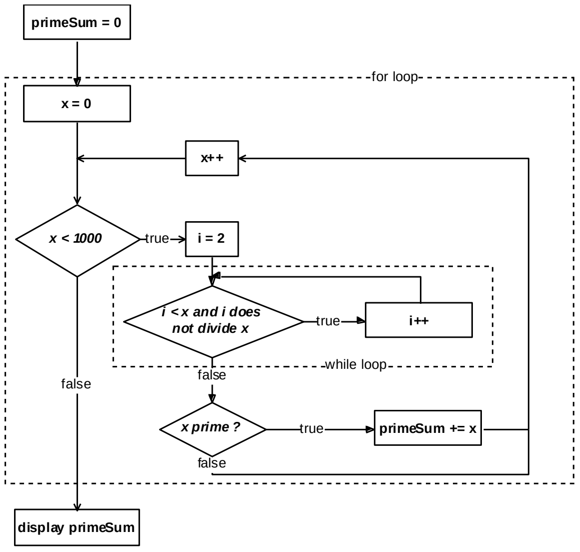
We can extend this example to create a program using both a `for` and a `while` loop:

Example 1.33: Sum of primes

Find the sum of all prime numbers less than `10000`.

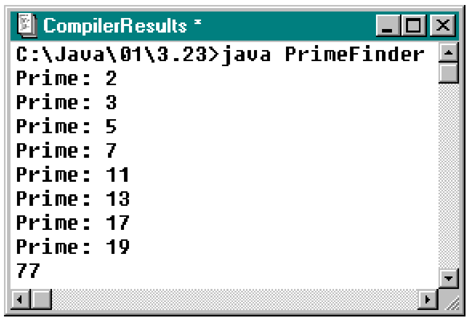
We have already seen the code for finding a running total (example 1.28) and for determining whether a number is prime (example 1.32). The code that solves this example combines both strategies:

```
int primeSum = 0;
for (int x = 0; x < 10000; x++)
{  int i = 2;

   while ((i < x) &&
          ((x % i) != 0))
      i++;

   if (i == x)
      primeSum += x;
}
System.out.println(primeSum);
```



This code is inefficient, but it works – almost. To see the error we insert a `System.out.println` statement that shows the prime numbers found in addition to adding them to the running total. We also verify the code by finding the sum of all primes less than `20`, which we can check with a calculator. A complete test program looks as follows:

```
public class PrimeFinder
{  public static void main(String args[])
   {  int primeSum = 0;
      for (int x = 0; x < 20; x++)
      {  int i = 2;
         while ((i < x ) && ((x % i) != 0))
            i++;
         if (i == x)
         {  System.out.println("Prime: "+x);
            primeSum += x;
         }
      }
      System.out.println(primeSum);
   }
}
```



```
C:\Java\01\3.23>java PrimeFinder
Prime: 2
Prime: 3
Prime: 5
Prime: 7
Prime: 11
Prime: 13
Prime: 17
Prime: 19
77
```

Figure 1.30: Finding the sum of primes

When we run this program (as shown in figure 1.30) we see the problem: the first prime number `1` is not added to the total. We leave it as an exercise to fix the mistake.

■

Java offers a third loop control structure, the `do` loop.

The `do` Loop

A `do` loop usually places any initialization before the actual loop and the modification statement inside the loop. The keywords `do` at the beginning of the `codeBlock` and `while` at the end are the essential and only mandatory parts, using the syntax:

```
do
{  codeBlock;
}
while (test);
```

The `test` is performed each time after `codeBlock` executed. If the `test` evaluates to `false` the `codeBlock` stops executing. In a `do` loop `codeBlock` executes at least once.

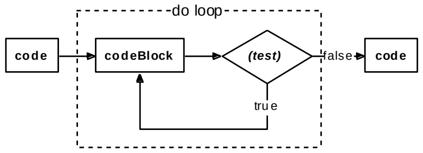


Figure 1.31: Representation of a `do` loop

A `do` loop is useful if the modification needs to occur *before* the test, because testing takes place at the end. But since it is easy to replace a `do` loop by a functionally equivalent `while` or `for` loop, we will not need `do` loops in this text.

Software Engineering Tip: We know three loop control structures, `for`, `while`, and `do`.

- The `for` loop is easiest to use. Use it if the number of times the loop should execute is known or can be computed easily. It should be your preferred looping structure.
- A `while` loop is the most general loop and can replace all other loops. Use it if a `for` loop is not applicable.
- A `do` loop is useful only if modification needs to occur before the loop test condition. A `do` loop always executes at least once.

Java provides two additional flow control statements called `break` and `continue`.

`break` and `continue`

A `break` statement can appear inside a `switch`, `for`, `while`, or `do` construction. It causes execution to jump to the first line immediately after the structure containing the `break`.

A `continue` statement can appear inside `for`, `while`, or `do` loops. It causes execution to skip over the remaining code in the code block and to jump directly to the loop test.

It is always possible to restructure code that involves the `break` and `continue` statements into equivalent code without those statements. We will therefore not need either statement in this text.

Software Engineering Tip: Avoid using `break` and `continue`. They introduce unnecessary jumps in the flow of your code and make it difficult to understand. Instead of using `break` or `continue`, restructure your code by using conditional execution, `boolean` variables, or methods utilizing a `return` statement.[36]

1.4. Strings

We have introduced the basic data types available in Java and can create programs that perform numeric computations and handle single characters. Programs often need to process text as well, but none of our current data types is suitable for text. To accommodate textual information, Java provides two types called `String` and `StringBuffer`. They are not primitive data types but strings are so useful that we introduce them as early as possible. You may want to revisit this section after learning about classes in chapter 3.

What is a `String`

First we need to define what a `String` is. Our first attempt is to say:

A String is a list of characters

But a list has a specific length whereas we want to consider, for example, "Java" and "Java by Definition" to be valid strings even though they have different lengths. We could say:

A String is a list of characters that can vary in length as needed

This is certainly a better definition but it does not cover a `String`'s complexity. A `String` is not only a list of characters whose length is the same as the number of characters it stores, but every variable of type `String` includes a variety of useful operations that can be used to manipulate itself. In chapter 3 we will see that this is a common phenomenon: a `String` is our first example of a *class* and classes always include operations to manipulate their own data.

`String`

A `String` is an immutable list of characters whose length is the same as the number of characters it stores such that every variable of type `String` contains a variety of useful operations. There are two ways to initialize a `String`:

```
String s1 = "list of characters"
String s2 = new String("list of characters");
```

Variables of type `String` can be concatenated, or appended to each other, using the concatenation operator `+`.[37] The operations that can be performed on `Strings` are described in table 34.

[36] See chapter 2 for the definition of a method and the `return` keyword.
[37] The concatenation operator `+` looks like the plus symbol +. The context decides how it is interpreted: if used with numeric types it represents addition, if used with `String` types it represents concatenation.

The characters in a `String` *are indexed starting at* `0`*, which implies that the last character in a* `String` *of length* `N` *has the index* `N-1`*. A variable of type* `String` *that is declared but not initialized is set to the object literal* `null`.

In other words, a `String` is different from the previous basic types for two reasons:

- A `String` can be of any length as needed, resulting in a flexible need for memory (whereas the basic types have fixed memory requirements).
- A `String` contains its own operations (whereas the standard numeric operations and comparison operators are separate operators that apply to the basic types but are not part of them).

Example 1.34: Initializing `String` *variables*

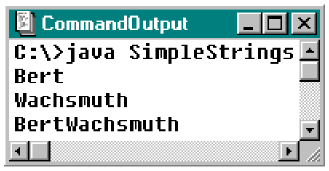
Create three variables of type `String` and initialize two of them to your first and last name, respectively. Then assign the full name to the third variable. Display all values on the screen in a complete program.

To initialize the first two strings we use the first variation of the above definition. To initialize the third, we concatenate the first two variables:

```
String first = "Bert";
String last = "Wachsmuth";
String name = first + last;
```

We embed this code segment into a complete program, using `System.out.println` to display the values on the screen. Figure 1.32 shows the result of executing the program.

```
public class SimpleStrings
{  public static void main(String args[])
   {  String first = "Bert";
      String last = "Wachsmuth";
      String name = first + last;
      System.out.println(first);
      System.out.println(last);
      System.out.println(name);
   }
}
```



```
C:\>java SimpleStrings
Bert
Wachsmuth
BertWachsmuth
```

Figure 1.32: Executing `SimpleStrings`

The variable `name` does not contain a space between first and last name so should concatenate the two strings `first` and `last` with a space in the middle:

```
String name = first + " " + last;
```

■

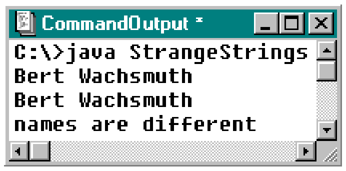
This example seems to imply that strings can be handled like the other basic types. But that is not true as the following example shows.

Example 1.35: Invalid String comparison with double-equal

Concatenate two strings to form a full name, based on the first and last name, and store it in a `String`. Then create another `String` that is initialized to the same full name in one line. Compare the two strings for equality and display the result of that comparison.

This might seem like a strange request. Both strings are supposed to contain the same characters in the same order, so comparing them for equality should result in `true`. But if we complete the example, the answer is surprising, as shown in figure 1.33.

```
public class StrangeStrings
{  public static void main(String args[])
   {  String name1 = "Bert Wachsmuth";
      String first = "Bert";
      String last  = "Wachsmuth";
      String name2 = first + " " + last;

      System.out.println(name1);
      System.out.println(name2);
      if (name1 == name2)
         System.out.println("names are the same");
      else
         System.out.println("names are different");
   }
}
```



```
C:\>java StrangeStrings
Bert Wachsmuth
Bert Wachsmuth
names are different
```

Figure 1.33: `StrangStrings` *Output*

■

Of course the characters in the strings are not different, but the double-equal operator "==" does not check strings for equality. Instead it checks whether two strings are located at the same memory location, which is usually of no interest to us.

Software Engineering Tip: Strings are *not* basic types and can not be compared using the comparison operators ==, <, >, <=, >=, or `!=`. To compare strings use the built-in `String` operations `equals` or `compareTo` described in table 34. Unfortunately the compiler does not complain if you *do* use the double-equal operator == to compare strings so this mistake is hard to catch.

`String` Operations

Built-in String Operations

Every variable of type `String` *contains a variety of built-in operations as shown in table 34. To access them, use the* `dot` *operator (described in detail in chapter three) as follows:*

```
s.operation(inputValues);
```

where `s` *is a variable of type* `String`, `operation` *is one of the operations described in table 34, and* `inputValues` *is a comma-separated list of variables or values of the type(s) required by the respective* `operation`, *or none if the* `operation` *does not require any parameters.*

Some of the operations that are allowed for a variable of type String are described as follows:

Operation	Description
`int` **`length`**`()`	Returns the length of the string.
`char` **`charAt`**`(int index)`	Returns the character at the specified index.
`boolean` **`equals`**`(String s)`	Returns `true` if the string represents the same

	sequence of characters as `s`.
`boolean **equalsIgnoreCase**(String s)`	Same as `equals` but ignores the case of the strings.
`int **compareTo**(String s)`	Returns `0` if the string is equal to `s`, less than `0` if it is lexicographically less than `s`, otherwise greater than `0`.
`boolean **startsWith**(String prefix)`	Tests if the string starts with the given `prefix`.
`boolean **endsWith**(String suffix)`	Tests if the string ends with the given `suffix`.
`int **indexOf**(String str)`	Returns the position where `str` first occurs in the string, or –1 if `str` is not contained in the string.
`String **substring**(int begin, int end)`	Returns a new string containing the characters from positions `begin` to `end-1`.
`String **replace**(char old, char new)`	Returns a new string where all characters `old` are replaced with `new`.
`String **toLowerCase**()`	Returns string converted to lowercase.
`String **toUpperCase**()`	Returns string converted to uppercase.
`String **trim**()`	Returns string without white space on both ends.

Table 1.34: Selected operations of the `String` *type*

Each operation acts on the string variable that invokes it and returns the data type indicated by the first word in the definition of the operation but leaves the original string unmodified. Additional operations and further information about the `String` class is available in the Java documentation in the `java.lang` package.

Example 1.36: String comparison with `equal`

In example 1.35 we used the double-equal operator to compare two strings with surprising results. Redo that example using the `equals` operation.

Instead of using the double-equal operator == as we have done in example 1.35:

```
if (name1 == name2)
   System.out.println("names are the same");
else
   System.out.println("names are different");
```

we use the equals operation as follows:

```
if (name1.equals(name2))
   System.out.println("names are the same");
else
   System.out.println("names are different");
```

The test `name1.equals(name2)` returns `true` if `name1` and `name2` contain the same characters in the same order. If you modify the `StrangeStrings` program from example 1.35 accordingly it reports that the strings are the same.

The test `name2.equals(name1)` is equivalent to `name1.equals(name2)`, and if upper and lowercase did not matter to use we could have used `name1.equalsIgnoreCase(name2)` or equivalently `name2.equalsIgnoreCase(name1)`. The tests

`(name1.compareTo(name2) == 0)` or `(name2.compareTo(name1) == 0)`

are also equivalent to `name1.equals(name2)`.

■

Example 1.37: Reversing a `String`

If `s` is an unknown string, write a program that will display the string in reverse order of characters, as well as in all upper case characters.

The operations that will be useful for this example are:

- `length()` to determine the length of the `String`
- `charAt(i)` to extract the `i`-th character from the `String`
- `toUpperCase()` to convert a `String` to uppercase characters

We use a loop that extracts characters from s in reverse order and appends them to a new `String` variable, similar to finding a running total in example 1.28. When the loop is done we convert the new `String` to uppercase characters. Remember that the last character in a `String s` has the index `s.length() - 1`.

```
String newString = "";
for (int i = s.length()-1; i >= 0; i--)
    newString += s.charAt(i);
newString = newString.toUpperCase();
```

■

Here is one more example with strings:

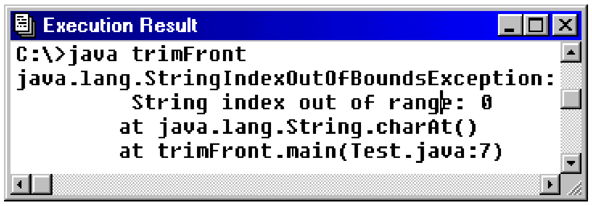
Example 1.38: Removing leading blanks

The `trim` operation of a string removes all trailing and leading blanks, if any, from a string. Write some code that removes only the *leading* blanks from an unknown string `s`.

We can not use a `for` loop because we do not know how many leading blanks the string `s` contains. Therefore we use a `while` loop to check each character in the string, starting at the beginning. If it is a space we remove it, otherwise our loop is done. The `String` operations that will be useful are `charAt(i)` to extract the `i`-th character from the string and `substring(start, end)` to extract characters from the string. Our first attempt looks like this:

```
while (s.charAt(0) == ' ')
   s = s.substring(1, s.length());
```

If `s = "   Java by Definition"` the code removes all leading blanks from s by setting it eventually to `"Java by Definition"`. But in the extreme situation of `s = "      "`, a string consisting entirely of blanks, our code generates a runtime error as shown in figure 1.35.



```
C:\>java trimFront
java.lang.StringIndexOutOfBoundsException:
         String index out of range: 0
        at java.lang.String.charAt()
        at trimFront.main(Test.java:7)
```

Figure 1.35: Accessing character at an invalid position in String

The error occurs because all characters were removed from the string `s` so that we can no longer ask for the character at the `0`th position. The string `s = ""` does not have a character at that position.

We guard against that possibility by adding an additional test to our `while` loop as follows:

```
while ((s.length() > 0) && (s.charAt(0) == ' '))
```

```
        s = s.substring(1, s.length());
```

Note that if the first condition evaluates to `false` Java does not need to check the second one before evaluating the entire expression to `false`. No runtime error will result in this improved code.

■

`StringBuffer` and Its Operations

We did not elaborate on the word "immutable" in the definition of a `String`. It means that a `String`, once initialized, can not change. The operations of a `String` variable provide information *about* the string or return a *new* string as a modified version of the original, but they leave the string itself unchanged. If we need to modify the characters contained in a `String`, another type must be used.

`StringBuffer`

A `StringBuffer` is a mutable list of characters whose length and composition is variable. A variable of type `StringBuffer` is initialized using the syntax:

```
   StringBuffer sb = new StringBuffer("list of characters");
```

Variables of type `StringBuffer` can use the `dot` operator to access the operations listed in table 1.36, similar to variables of type `String`.[38]

None of the standard numerical operators are valid for variables of `StringBuffer` type including the concatenation operator `+` that works fine for `String` types. In fact, any operation using the concatenation operator `+` for strings is internally converted to an operation using `StringBuffers` and their `append` and `toString` operations. For example, the third line of the code segment

```
String first = "Bert";
String last = "Wachsmuth";
String name = first + " " + last;
```

is internally and automatically converted[39] to operations similar to the following:

```
StringBuffer tempBuffer = new StringBuffer(first);
tempBuffer.append(last);
String name = tempBuffer.toString();
```

Operation	Description
append`(type t)`	Appends `t`, where `t` can be any `type`.
insert`(int pos, type t)`	Inserts `t` at position `pos`, where `t` can be any `type`.
delete`(int start, int end)`	Deletes characters from position `start` to `end-1`.
deleteCharAt`(int pos)`	Removes character at position `pos`.
setCharAt`(int index, char ch)`	Sets character at position `pos` to `ch`.
replace`(int start, int end, String str)`	Replaces characters from `start` to `end` by `str`.

[38] Additional operations and further information is available in the Java documentation about the `java.lang.StringBuffer` class.

[39] The concatenation operator does not change a `String`. For example, in `String s = "Pizza"; s += s` a new `StringBuffer` is initialized, `"Pizza"` is appended twice, and a *new* `String` with value `"PizzaPizza"` is returned. The old `String` with value `"Pizza"` is no longer used.

`int length()`	Returns the number of characters currently stored.
`String toString()`	Returns the `StringBuffer` as `String`.

Table 1.36: Selected operations of the `Stringbuffer` *type*

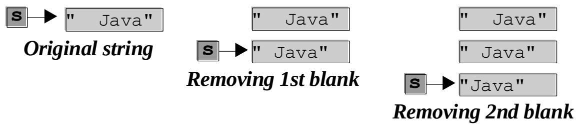
Example 1.39: Removing leading blanks using `StringBuffer`

In example 1.38 we wrote a code segment to remove leading blanks from a string. Was that code appropriate? Rewrite it if necessary.

Recall the code segment from example 1.38:

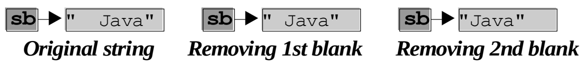
```
while ((s.length() > 0) && (s.charAt(0) == ' '))
   s = s.substring(1, s.length());
```

That code is inappropriate. Every time a leading blank needs to be removed a *new* string is created by extracting characters from `s` and reassigning them to `s`.



Previous versions of `s` are unused, wasting computer memory and time. Since our intention is to modify the `String s` if necessary we should use a `StringBuffer` instead of a `String`:

```
StringBuffer sb = new StringBuffer(s);
while ((sb.length() > 0) && (sb.charAt(0) == ' '))
   sb.removeCharAt(0);
s = sb.toString();
```



This code is more efficient and has the same functionality so it is preferred over the original code.

■

Software Engineering Tip: A `String` contains characters that can not change while a `StringBuffer` is flexible.

- If you need to add or remove characters from a string, use a `StringBuffer`, not a `String`.
- If you need text that remains more or less constant, use a `String`.

As a rule of thumb, if a `String` variable is on the left side of an assignment (other than initialization), a `StringBuffer` would be the better data type.

The difference in efficiency of using `StringBuffer` over `String` becomes apparent only if a large string or a large number of strings are manipulated. In this text we are rarely confronted with that situation so we will use, for convenience, the concatenation operator to add strings even though a `StringBuffer` with its `append` operation would be more appropriate.

1.5. Output and Constants

At this point we know how to store and manipulate data and how to control the basic flow of a program. This section describes how to produce formatted output on the screen and how to create and use named constants. It includes a recipe for formatting decimal numbers so that we can create programs that produce reasonably nice looking output.

Simple Screen Output

We have already seen how to produce output that appears on the screen using `System.out.println`. Here is a description of that mechanism in some more detail.[40]

Output using `System.out` and `System.err`

Java can display the values of variables and constants on the system console (usually the display from where the program started) using the keywords `System.out`*, the dot operator, and either*

- `println(expression)` *to display the values of the expression and an additional* `newline` *character, leaving the cursor at the beginning of the next line, or*
- `print(expression)` *to display the values of the expression, leaving the cursor immediately after the last displayed character*[41]

where `expression` *can contain constants, literals, and variables or all types, combined with the concatenation operator* `+`*. The control sequences* `\t` *and* `\n` *provide limited formatting options.*

Java also offers the commands `System.err.print` *and* `System.err.println` *to produce output on a device to display error messages, which is usually the system console as well.*

Example 1.40: Output with simple formatting

Create a complete program with one variable from each of the basic types. Initialize them and display their values together with appropriate text. Also, display the values of e (Euler's number), π, $\sqrt{2}$, and $\cos(e+\pi)$.

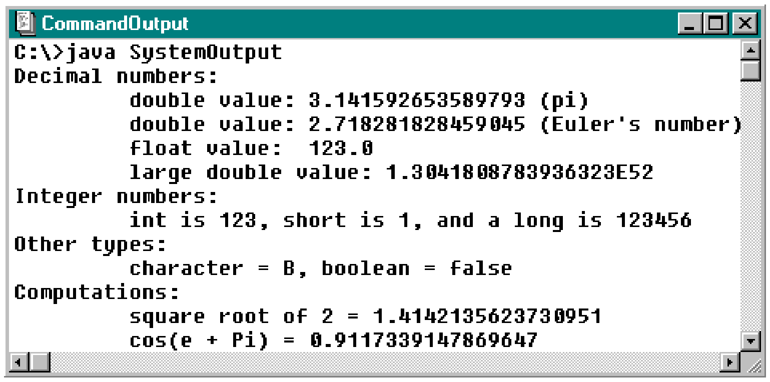
To create this program we follow our basic framework:

```
public class SystemOutput
{  public static void main(String args[])
   {  short aShort = 1;
      int anInt = 123;
      long aLong = 123456L;
      float aFloat = 123F;
      double pi = Math.PI, e = Math.E;
      char cc = 'B';
      boolean okay = false;
      System.out.println("Decimal numbers:");
      System.out.println("\t double value: " + pi + " (pi)");
      System.out.println("\t double value: " + e + " (Euler's number)");
      System.out.println("\t float value:  " + aFloat);
      System.out.println("\t large double value: " + Math.exp(120.0));
      System.out.println("Integer numbers:");
```

[40] A complete discussion of the `System.out` and `System.err` streams can be found in section 8.6.
[41] Output may not appear until you produce a `newline` character. To force output to appear in that case, use the command `System.out.flush()` after `System.out.print(expressionList)`.

```
            System.out.print("\t int is " + anInt);
            System.out.print(", short is " + aShort);
            System.out.println(", and a long is " + aLong);
            System.out.println("Other types:");
            System.out.println("\t character = " + cc + ", boolean = " + okay);
            System.out.println("Computations:");
            System.out.println("\t square root of 2 = " + Math.sqrt(2.0));
            System.out.println("\t cos(e + Pi) = " + Math.cos(e + pi));
            System.out.flush();
        }
    }
```

The program must be saved as `SystemOutput.java`, compiled via `javac SystemOutput.java`, and executed using `java SystemOutput`.



```
C:\>java SystemOutput
Decimal numbers:
        double value: 3.141592653589793 (pi)
        double value: 2.718281828459045 (Euler's number)
        float value:  123.0
        large double value: 1.3041808783936323E52
Integer numbers:
        int is 123, short is 1, and a long is 123456
Other types:
        character = B, boolean = false
Computations:
        square root of 2 = 1.4142135623730951
        cos(e + Pi) = 0.9117339147869647
```

Figure 1.37: Execution results for `SystemOutput.java`

The output shown in figure 1.37 shows in particular:

- `Double` values can have up to 15 digits after the period (see "Formatting Decimals" below).
- Large `double` values are displayed in scientific form.[42]
- The control sequence `\t` can be used to indent the output.
- `System.out.print` leaves the cursor immediately after the last character displayed.
- Spacing and other characters between expressions can be appended as `String` literals.
- `System.out.println(...)` can display the results of computations directly.

■

The ability to connect multiple expressions via the concatenation symbol is very convenient to quickly produce reasonably looking output, but it is not always correctly interpreted.

Example 1.41: Different interpretation of plus symbol

What is displayed when the following statements are executed:

```
int i = 10;
System.out.println("Adding one to i: " + i+1);
```

The `System.out.println` statement contains two `+` symbols with different interpretations:

- The first `+` is meant to concatenate the `String` and the result of the computation `i+1`.
- The second `+` is meant to add `1` to the value of `i`.

[42] A `double` is displayed with up to 15 digits after the period if $10^{-3} < |$`double`$| < 10^{7}$, otherwise in scientific notation.

But only the first interpretation of + is used and Java displays

```
Adding one to i: 101
```

instead of "`Adding one to i: 11`" because it appends the test, the current value of `i`, and the literal `1` and does not *add* `1` to `i`. To produce the intended output we need to use parenthesis to clarify the interpretation of the + symbols:

```
System.out.println("Value of i+1 is: " + (i+1));
```

■

Formatting Decimal Numbers

There are many occasions when you not only want to display the value of a number but also control its format. For example, you may want all decimals to have six digits after the period, or variables representing currency values should show two digits after the decimal point.[43]

Formatting Decimals

To display formatted decimal numbers without changing their value use the following steps:

1. *Before defining your class or program insert as the very first line of the source code:*
   `import java.text.DecimalFormat;` *<br>*
2. *In your code create a "number formatting tool" using the syntax:*
   `DecimalFormat formatToolName = new DecimalFormat("pattern");` *<br>*
3. *Display a formatted version of a* `float` *or* `double` *variable* `number` *using the syntax:*
   `System.out.println("Formatted: " + formatToolName.format(number));` *<br>*

The value of `pattern` *is a combination of the symbols* `"#"`, `"0"`, `","` *and* `"."`, *where* `","` *denotes a number group separator,* `"."` *denotes a decimal separator,* `"#"` *represents a digit except trailing zeros after a decimal separator, and* `"0"` *represents a digit.*

Table 1.38 shows some examples of formatting patterns and their meaning:

Pattern	Meaning of Pattern
`"#.#"`	digits before decimal point as needed, no group separator between them, and at most one digit after the decimal point
`"#,###.##"`	numbers before the decimal point as needed with groups of three digits separated by a comma and *at most* two digits after the decimal point
`"#,###.00"`	numbers before the decimal point as needed with groups of three digits separated by a comma and *exactly* two digits after the decimal point

Table 1.38: Examples of decimal number formatting patterns[44]

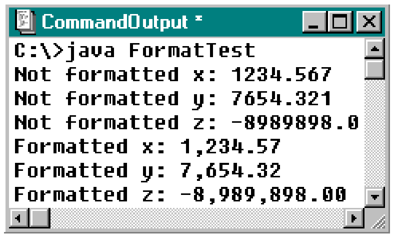
Example 1.42: Formatting decimal numbers

Create a program that contains three `double` numbers bigger than 1,000, two with 3 decimal numbers after the period and one without digits after the period. Display each number on the screen, first without formatting, then formatted as US Dollar values.

[43] The formatting mechanism explained here contains language constructs introduced in chapter 3 but since numeric formatting is simple and useful we introduce it here in a "cook-book" style.
[44] For additional formatting options check the Java API for the `java.text.DecimalFormat` class

Currency values in US Dollar need "," after each group of three digits before the decimal point and 2 digits after the decimal point. Based on table 38 we might try to use the pattern "#.##", but it would not insert grouping symbols after each group of three digits and it would display a *maximum* of 2 digits after the period, not *exactly* 2 digits. Instead the pattern "#,###.00" will achieve our goal, as shown in figure 1.39.[45] We name our formatting tool inUSDollars to signify its formatting pattern.

```
import java.text.DecimalFormat;

public class FormatTest
{  public static void main(String args[])
   {  DecimalFormat inUSDollars = new DecimalFormat("#,###.00");
      double x = 1234.567;
      double y = 7654.321;
      double z = -8989898;
      System.out.println("Not formatted x: " + x);
      System.out.println("Not formatted y: " + y);
      System.out.println("Not formatted z: " + z);
      System.out.println("Formatted x: " + inUSDollars.format(x));
      System.out.println("Formatted y: " + inUSDollars.format(y));
      System.out.println("Formatted z: " + inUSDollars.format(z));
   }
}
```



```
C:\>java FormatTest
Not formatted x: 1234.567
Not formatted y: 7654.321
Not formatted z: -8989898.0
Formatted x: 1,234.57
Formatted y: 7,654.32
Formatted z: -8,989,898.00
```

Figure 1.39: Output of FormatTest *program*

The numbers are formatted as specified and are properly rounded.[46]

■

Software Engineering Tip: One formatting tool can be used to format any number of double variables. It should be declared at the beginning of your program to make it easy to locate and change the formatting pattern.

If multiple formatting patterns are needed, define one formatting tool for each formatting option and declare all of them at the beginning of your program. Choose suitable names for the formatting tools that signify the formatting they apply (e.g. inPercent or withTwoDigits).

Named Constants

To conclude our discussion about the fundamentals of Java programming we explain how to define named constants, which are commonly used to create flexible programs and to provide mnemonic devices. For example, instead of the value 3.141592653589793 the constant Math.PI is easier to

[45] A more appropriate pattern is "$#,###.00" to automatically precede the numeric value by the dollar symbol $.
[46] You can display the unformatted version of the numbers again to verify that the actual values remain unchanged.

remember, and instead of creating a for loop from `0` to `10` it is more convenient to define a constant such as `MAX_LOOP` that equals `10` so that the loop goes from `0` to `MAX_LOOP`.

Named Constants via `final`

To prohibit any modification of a variable its type can be prefaced by the keyword `final`*. Such a variable is called named constant and must be initialized when it is declared. Its value can not change after it has been initialized.* `Final` *variable names usually consist of uppercase characters and are declared at the beginning of a program.*

We have already seen the named constants `Math.PI` and `Math.E` for the values of and e. This definition explains how to define our own constants.

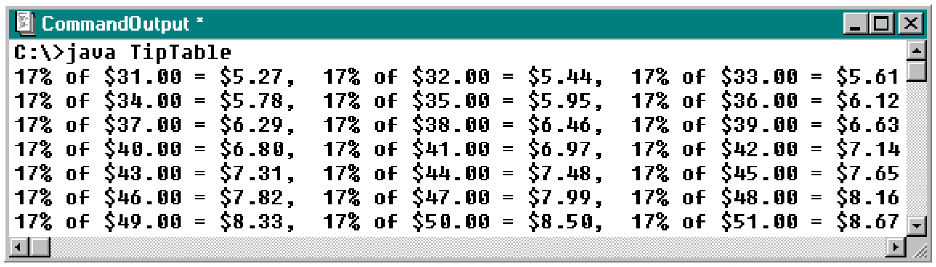
Example 1.43: A Tip Table program

Write a program that creates a "tip table" to find the correct tip for the amount of a check at a restaurant.

Tips are usually about 15% of the amount of the check. Our tip table should contain possible amounts for restaurant checks together with the corresponding tip. People often have their own preferred tip rate and visit restaurants with different price ranges, so we make our program flexible by defining the tip rate and the smallest and largest amount for the check as constants. To ensure that our table will look nice we use decimal formatting and output options as explained previously.

```
import java.text.DecimalFormat;

public class TipTable
{  public static void main(String args[])
   {  final double TIP_RATE = 17;
      final double MINIMUM_BILL = 31.0;
      final double MAXIMUM_BILL = 51.0;
      final double STEP    = 1.0;
      DecimalFormat inUSDollars = new DecimalFormat("#,###.00");
      DecimalFormat inPercent   = new DecimalFormat("#,###");

      for (double check = MINIMUM_BILL; check <= MAXIMUM_BILL; check += STEP)
      {  double tip = check * TIP_RATE / 100;
         System.out.print(inPercent.format(TIP_RATE));
         System.out.print("% of $" + inUSDollars.format(check));
         System.out.print(" = $" + inUSDollars.format(tip));
         if ((check % 3) == 0)
            System.out.println("");
         else
            System.out.print(",\t");
      }
   }
}
```

Figure 1.40 shows the output for this program. Because we used named constants it can easily be adjusted by modifying the values at the top of our program without changing the code that does the actual computations.



```
C:\>java TipTable
17% of $31.00 = $5.27,  17% of $32.00 = $5.44,  17% of $33.00 = $5.61
17% of $34.00 = $5.78,  17% of $35.00 = $5.95,  17% of $36.00 = $6.12
17% of $37.00 = $6.29,  17% of $38.00 = $6.46,  17% of $39.00 = $6.63
17% of $40.00 = $6.80,  17% of $41.00 = $6.97,  17% of $42.00 = $7.14
17% of $43.00 = $7.31,  17% of $44.00 = $7.48,  17% of $45.00 = $7.65
17% of $46.00 = $7.82,  17% of $47.00 = $7.99,  17% of $48.00 = $8.16
17% of $49.00 = $8.33,  17% of $50.00 = $8.50,  17% of $51.00 = $8.67
```

Figure 1.40: Tip rates at 17% for restaurant checks from $31 to $51

■

Software Engineering Tip: Defining constants via the keyword `final` is common practice. A good Java program should never refer to an unnamed constant for anything significant. Always use constants to give a name to relevant values and choose the names for your constants so that they clearly signify their meaning.

Case Study: Java Primes and the Prime Number Theorem

In this optional section we create a program to find prime numbers and to verify a mathematical theorem called the Prime Number Theorem. The program pulls together everything we have learned so far into one project.

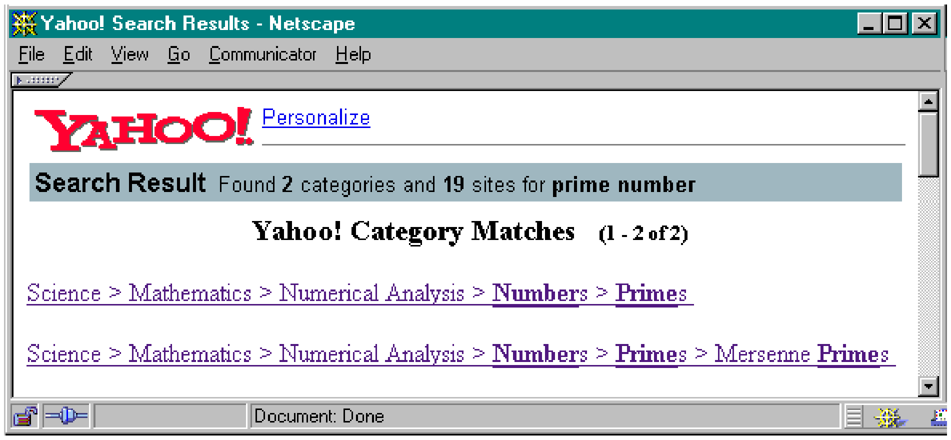
Example 1.44: The largest Java prime and the Prime Number Theorem

How many prime numbers are there, and what is the largest prime number?

Before we attempt to write a program we need to understand the problem. One fact is clear:

> ***Definition of Prime Number:*** *A positive integer is called prime if the only numbers that divide it without remainder are 1 and itself.*

For example, the numbers 1, 2, 3, 5, 7, 11, and 13 are prime but 9 and 15 are not. To find more information about primes, we search the World-Wide-Web using a search engine such as `www.yahoo.com` for the topic "`prime number`".



Yahoo! Search Results - Netscape
File Edit View Go Communicator Help
YAHOO!
Personalize
Document: Done

Search Result Found 2 categories and 19 sites for prime number

Yahoo! Category Matches (1 - 2 of 2)

Science > Mathematics > Numerical Analysis > Numbers > Primes

Science > Mathematics > Numerical Analysis > Numbers > Primes > Mersenne Primes

Figure 1.41: Searching the web for "prime number"

We find many references (see figure 1.41), among them the site `www.utm.edu/research/primes/`. That site has a great collection of facts about prime numbers and their mathematical theory and answers our question right away:

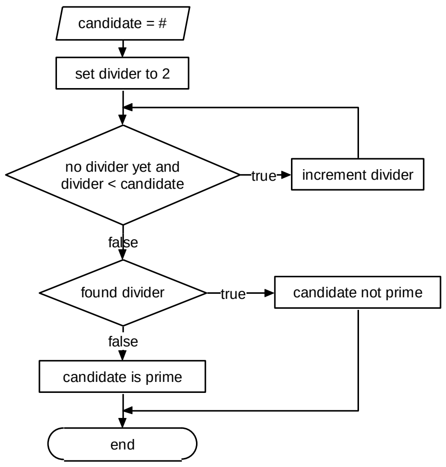
> ***Euclid's Theorem:*** *There is no largest prime number, which implies that there are infinitely many prime numbers.*[47]

This answer is somewhat unsatisfying so we decide to rephrase the questions slightly:

1. We know that there is no largest prime number, but what is the largest prime we can compute using a Java program?
2. We know that there are infinitely many prime numbers, but how many prime numbers are less than `N`, where `N` is any integer?

Question 1: To find the largest prime we can compute we need an algorithm to decide whether a given number is prime (compare example 1.32).

- Define a number `candidate`
- Divide `candidate` by all numbers less than `candidate` and check the remainder
- If a remainder equals zero, `candidate` can be divided without remainder and is not prime.
- If the remainders are never zero, `candidate` cannot be factored and must be prime.



To determine the remainder we can use the `%` operator, but we must make sure that all numbers computed are exact and not approximations. Therefore we can not use `double` types, even though they have the largest range (compare the *Software Engineering Tip* after "*Bits and Bytes*" in section 1.2). We must restrict our search for the largest Java prime to integers, which implies:

> *The largest prime we can compute must be smaller than the largest* `long` *value possible, i.e. it must be smaller than 9,223,372,036,854,775,807.*[48]

We have an algorithm and we know that all numbers involved must be of type `long` so we can write Java code to tell us whether a given number `primeCandidate` is prime.

```
long divider = 2;
boolean foundDivider = (primeCandidate % divider) == 0;
while ((!foundDivider) && (divider < primeCandidate))
{  foundDivider = (primeCandidate % divider) == 0;
   divider++;
```

[47] See `www.shu.edu/projects/reals/logic/proofs/euclidth.html` for a proof of Euclid's theorem.
[48] Java includes facilities to handle numbers of arbitrary size so that the restriction of looking for primes smaller than the largest `long` is imposed by our limited knowledge of Java programming, *not* by what is *really* possible in Java.

```
}
if (foundDivider)
   System.out.println(primeCandidate + " is not prime");
else
   System.out.println(primeCandidate + " is prime");
```

But if a number `n` divides `primeCandiate`, then `n * m = primeCandidate` for some integer `m` and at least one of the numbers `n` or `m` must be less than or equal to $\sqrt{\text{primeCandiate}}$. Therefore we only need to check for dividers that are no larger than $\sqrt{\text{primeCandiate}}$, so we can improve the above `while` loop:

```
... as before
while ((!foundDivider) && (divider <= Math.sqrt(toCheck)))
{  foundDivider = (toCheck % divider) == 0;
   divider++;
}
... as before
```

We are checking divisibility by 2, 3, 4, ... but if a number is not divisible by 2 it can not be divisible by any even number. Therefore it suffices to check divisibility by 2 and then only by odd numbers:

```
long divider = 2;
boolean foundDivider = (primeCandidate % divider) == 0;
divider = 3;
while ((!foundDivider) && (divider <= Math.sqrt(primeCandidate)))
{  foundDivider = (primeCandidate % divider) == 0;
   divider += 2;
}
... as before
```

Now we create a complete program to find the largest prime number that is less than or equal to the largest possible `long`. It starts by setting `primeCandidate` to the largest `long` possible, checks it for being prime, then decrements `primeCandidate` by one and checks again until a prime is found.

```
public class LargestPrime
{  public static void main(String args[])
   {  long primeCandidate = 9223372036854775807L;[49]
      boolean foundPrime = false;
      while (!foundPrime )
      {  long divider = 2;
         boolean foundDivider = (primeCandidate % divider) == 0;
         divider = 3;
         while ((!foundDivider) && (divider <= Math.sqrt(primeCandidate)))
         {  foundDivider = (primeCandidate % divider) == 0;
            divider += 2;
         }
         if (foundDivider)
            System.out.println(primeCandidate + " is not prime");
         else
            System.out.println(primeCandidate + " is prime");
         foundPrime = !foundDivider;
         primeCandidate--;
      }
   }
}
```

Figure 1.42 shows the output of our program, which takes quite some time to complete.[50] It finds:

[49] Note the `L` at the end of the digits, which indicates that this is a `long` literal (compare "*Literals*" in section 1.2).

The largest prime of type `long` is 9,223,372,036,854,775,783[51]

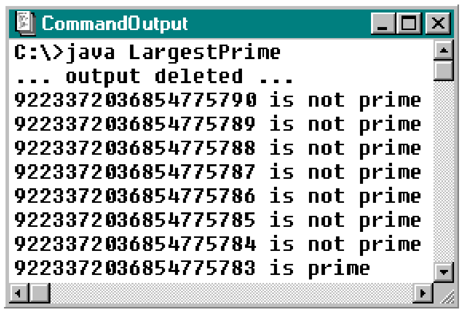

```
C:\>java LargestPrime
... output deleted ...
9223372036854775790 is not prime
9223372036854775789 is not prime
9223372036854775788 is not prime
9223372036854775787 is not prime
9223372036854775786 is not prime
9223372036854775785 is not prime
9223372036854775784 is not prime
9223372036854775783 is prime
```

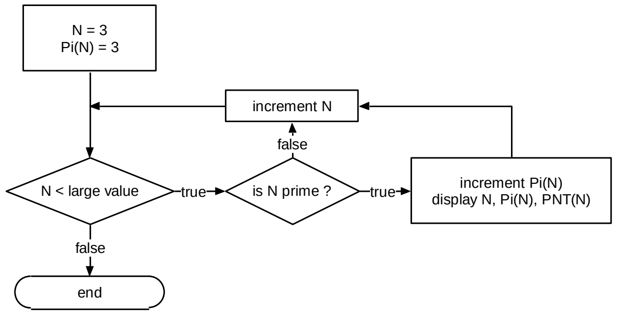
Figure 1.42: Finding the largest prime inside the range for the `long` type

Question 2: Now we solve the second question of counting the number of primes less than `N` for any given integer `N`. Reading through the information provided at `www.utm.edu/research/primes/` we find a theorem called the *Prime Number Theorem* that answers the question:

Prime Number Theorem: *There are approximately* $\frac{N}{\log(N)-1}$ *primes not exceeding N.*

We want to create a program to verify this theorem. If `PNT(N)` stands for the number hypothesized by the Prime Number Theorem and `Pi(N)` denotes the actual number of primes less than or equal to `N` we can use the following algorithm:

- Set `N` to `3` and `Pi(N)` to `3` (because there are 3 primes less than or equal to `3`).
- Check if `N` is prime.
- If so, increment `Pi(N)` by one and display `N`, `Pi(N)`, `PNT(N)`, and the difference |`Pi(N)` – `PNT(N)`| in percent.
- Increment `N` and repeat the process.
- Stop when `N` reaches the largest possible prime we can handle.



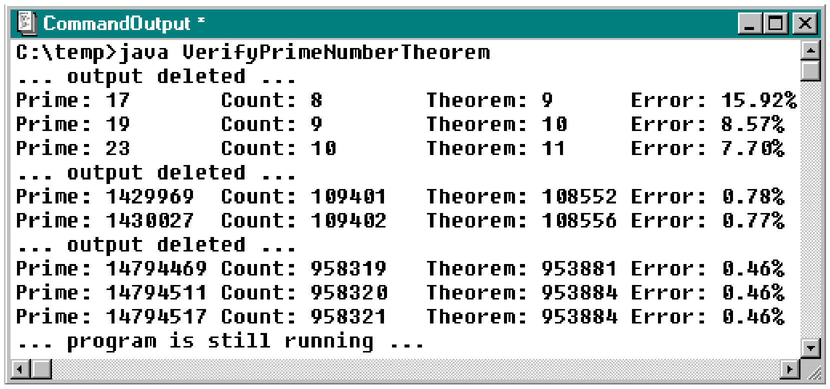
To make our program flexible we define constants for the starting values of `N` and `Pi(N)`. To format the error between `Pi(N)` and `PNT(N)` we use the decimal formatting tool described in section 1.5. We use the variable names `candidate` instead of `N`, `counter` instead of `Pi(N)`, and `pnt` instead of `PNT(N)` to make our program more readable.

```
import java.text.DecimalFormat;

public class VerifyPrimeNumberTheorem
{  public static void main(String args[])
   {  final long START_AT    = 3;
      final long STOP_AT     = 9223372036854775783L;
      final long START_COUNT = 2;
```

[50] It is easy to decide whether small numbers are prime, but finding large primes gets increasingly so complicated that encryption algorithms for commercial and military data encryption are based on that fact.
[51] François Edouard Anatole Lucas (1842 – 1891) showed in 1876 that 2^{127}-1 (which is a 39 digit number) is prime, which remains the largest prime number discovered *without the aid of a computer*.

```
            DecimalFormat inPercent = new DecimalFormat("#,##0.00");

            long counter = START_COUNT;
            for (long candidate = START_AT; candidate <= STOP_AT; candidate++)
            {  long divider = 2;
               boolean foundDivider = (candidate % divider) == 0;
               divider = 3;
               while ((!foundDivider) && (divider <= Math.sqrt(candidate)))
               {  foundDivider = (candidate % divider) == 0;
                  divider+=2;
               }
               if (!foundDivider)
               {   counter++;
                   double pnt = candidate/(Math.log(candidate)-1);
                   double error = Math.abs(counter - pnt) / counter * 100;
                   System.out.print("Prime: " + candidate);
                   System.out.print("\tCount: " + counter);
                   System.out.print("\tTheorem: " + (int)Math.round(pnt));
                   System.out.println("\tError: " + inPercent.format(error) + "%");
               }
            }
         }
      }
```

This program will take forever to finish, but we can stop it at any time,[52] adjust the constants according to the latest output of the program, and recompile and restart the program at a later time. Using that technique several runs of the program are shown in figure 1.43.



```
C:\temp>java VerifyPrimeNumberTheorem
... output deleted ...
Prime: 17       Count: 8        Theorem: 9      Error: 15.92%
Prime: 19       Count: 9        Theorem: 10     Error: 8.57%
Prime: 23       Count: 10       Theorem: 11     Error: 7.70%
... output deleted ...
Prime: 1429969  Count: 109401   Theorem: 108552 Error: 0.78%
Prime: 1430027  Count: 109402   Theorem: 108556 Error: 0.77%
... output deleted ...
Prime: 14794469 Count: 958319   Theorem: 953881 Error: 0.46%
Prime: 14794511 Count: 958320   Theorem: 953884 Error: 0.46%
Prime: 14794517 Count: 958321   Theorem: 953884 Error: 0.46%
... program is still running ...
```

Figure 1.43: Verifying prime number theorem

We can see, for example, that there are 10 prime numbers less than or equal to 23 (1, 2, 3, 5, 7, 11, 13, 17, 19, and 23), while the Prime Number Theorem states that there are approximately 11 (rounded to the nearest integer). After the program executes for quite some time, it states that 958,321 prime numbers are less than or equal to 14,794,517, while the Prime Number Theorem hypothesizes a count of 953,884, amounting to an error of 0.46%. Our program indicates that the Prime Number Theorem indeed predicts the number of primes not exceeding a given integer accurately, especially for large numbers.

■

If you are interested in prime numbers, check the site `www.mersenne.org`. It describes the "*Great Internet Mersenne Prime Search*" (GIMPS), which ties together thousands of small personal computers over the Internet to find *huge* prime numbers. GIMPS looks for *Mersenne Primes*, which

[52] To interrupt a Java program at any time, press `<Ctrl><C>`.

are prime numbers that can be expressed by the formula 2^p-1 for some integer p.[53] Throughout history, the largest known prime number has usually been a Mersenne prime. Over 8,000 people have contributed computer time to help discover world-record Mersenne primes. GIMPS has discovered four Mersenne primes since 1996, and on June 1st 1999, Nayan Hajratwala found the current world record prime, $2^{6972593}$-1.[54] You can join the search for the next world record prime by downloading free software provided at that site.

Chapter Summary

In this chapter we introduced the following concepts and terminology:

Java Programming Guidelines, Source Code
See section 1.1, example 1.01 (*The first source code file*)
Compiling
See section 1.1, examples 1.02 (*Compiling a source code file*) and 1.03 (*Compiling source code with errors*)
Executing a Class File
See section 1.1, example 1.04 (*Executing a class file*)
Default Program Entry Point
See section 1.1, example 1.05 (*Executing class file without* `main`)
Java Virtual Machine (JVM), Security
Primitive Java Data Types, Literals, Binary Representation
See section 1.2, example 1.06 (*Converting integers to binary form*)
Bits and Bytes
See section 1.2, example 1.07 (*Number of bytes to store* `boolean`, `int`, *and* `long`)
Declaration of Variables, Valid Names for Variables
See section 1.2, example 1.08: (*Declaring variables*)
Assigning a Value to a Variable
See section 1.2, examples 1.09 (*Declaring variables and assigning values*) and 10 (*Using appropriate variable names*)
Basic Arithmetic for Numeric Types
See section 1.2, example 1.11 (*Basic computations*)
Advanced Arithmetic for Numeric Types
See section 1.2, example 1.12 (*Advanced computations*)
Increment and Decrement Operators
See section 1.2, example 1.13: (*Prefix and postfix increment and decrement*)
Computational Shortcut Notation
See section 1.2, example 1.14 (*Shortcut evaluation*)
Simple Typecasting or Conversion
See section 1.2, example 1.15 (*Typecasting to* `double`) and 1.16 (*Typecasting large range to small range types*)
Boolean Operators, Comparison Operators and Tests
See section 1.2, example 1.17 (*Simple tests*) and 1.18: (*Tests with short-circuit* `boolean` *evaluation*)
The `instanceof` Operator
Block of Code, The Simple `if` Statement

[53] Prime numbers are not necessarily of the form 2^p-1. For example, the largest `long` is 2^{63}-1, which is not prime, and the number 9,223,372,036,854,775,807 we found earlier is prime but not a Mersenne prime.
[54] $2^{6972593}$-1 is a prime number with 2,098,960 digits (our largest prime has a meager 19 digits).

See section 1.3, examples 1.19 (*Simple* `if` *statement*) and 1.20 (*Invalid simple* `if` *statement*)

The `if-else` Statement with Alternative

See section 1.3, examples 1.21 (*Simple* `if-else` *statement*) and 1.22 (*Invalid* `if-else` *statement*)

The Nested `if-else-if` Statement

See section 1.3, example 1.23 (*Converting numeric scores to letter grades*)

The `switch` Statement

See section 1.3, examples 1.24 (*Converting letter grades to numeric scores*), 1.25 (Replacing `if-else-if` *code by* `switch`*),* and 1.26 (*Replacing* `switch` *by* `if-else-if` *code)*

Loop, The `for` Loop

See section 1.3, examples 1.27 (*Simple* `for` *loop*), 1.28 (*Finding running total*), 1.29 (*Finding running totals with different step sizes*), and 1.30 (*Over- or underperforming loops*)

The `while` Loop

See section 1.3, examples 1.31 (*Replacing* `for` *loop with* `while`*)*, 1.32 (*Testing for prime numbers*), and 1.33 (*Sum of primes*)

The `do` Loop, `break` and `continue`

`String`

See section 1.4, examples 1.34 (*Initializing* `String` *variables*) and 1.35 (*Invalid String comparison with double-equal*)

Built-in String Operations

See section 1.4, examples 1.36 (*String comparison with* `equal`*)*, 1.37 (*Reversing a* `String`), and 1.38 (*Removing leading blanks*)

`StringBuffer`

See section 1.4, example 1.39 (*Removing leading blanks using* `StringBuffer`)

Output using `System.out` and `System.err`

See section 1.5, examples 1.40 (*Output with simple formatting*) and 1.41 (*Different interpretation of plus symbol*)

Formatting Decimals

See section 1.5, example 1.42 (*Formatting decimal numbers*)

Named Constants via `final`

See section 1.5, example 1.43 (*A Tip Table program*)

Case Study: Java Primes and the Prime Number Theorem