Java programming for C/C++ developers

Skill Level: Introductory

Scott Stricker (sstricke@us.ibm.com)
Developer
IBM

28 May 2002

This tutorial uses working code examples to introduce the Java language to C and C++ programmers.

Section 1. Getting started

What is this tutorial about?

This tutorial introduces the Java programming language to C and C++ developers. Because you already know how to program in C/C++, we'll approach many Java programming concepts by comparison. You will learn a great deal about Java programming by learning how the Java language is similar to, and different from, C and C++. Overall, the purpose of this tutorial is to teach you the fundamentals of the Java language and get you programming quickly.

The creators of the Java programming language borrowed much of its syntax from C and C++. Because of this, many experienced C/C++ programmers are immediately familiar with many aspects of Java code, even if they've never programmed in the language before. This is important because developers with a C/C++ background are able to learn how to program in Java more quickly than beginning programmers or developers coming from other languages.

To make the most of your advantage as a C/C++ programmer, it is important to keep in mind that the differences between the languages are usually more significant than the similarities; failure to recognize this can result in incorrect code. First of all, C/C++ programmers have to be cautious when using the features of the Java language that behave differently from their C/C++ counterparts, such as boolean expressions and default parameter *passing by reference* instead of *passing by value*. Second, C/C++ programmers have to learn how to get along without many C/C++ language features on which they have previously relied, such as pointers, global variables, and the preprocessor.

Should I take this tutorial?

This tutorial is geared toward C and C++ programmers. If you already know C or C++ and want to learn how to program in the Java language, this tutorial is for you. If you don't know C or C++, but still want to learn the Java programming language, you may want to check the listings in Resources for a tutorial that is better suited to your background.

As far as this tutorial goes, it doesn't particularly matter if you're a C programmer or a C++ programmer; we'll discuss the differences between the C and C++ languages as they come up. Despite what many people think, C and C++ really are different languages. Many C programmers have never programmed in C++, and are completely unfamiliar with object-oriented programming. Likewise, many C++ programmers have learned object-oriented programming using C++ and are not proficient in a purely C, procedural programming environment.

What do I need to take this tutorial?

In this tutorial, we'll be compiling and running Java programs, so you'll need a Java development environment of some kind. There are several integrated development environments (IDEs) on the market. These are intended to help you develop Java programs quickly and easily. An even better tool for beginners is the Java Software Development Kit (SDK), which is a collection of very simple command-line tools for programming on the Java platform.

While you will almost certainly migrate to some other, more advanced Java programming environment, there are several good reasons to start out using the SDK. First, the SDK provides a standard implementation of the tools we'll need to compile and run Java programs. Second, as newer versions of the Java platform are released, Sun's SDK is usually the first and only up-to-date implementation available. Third, the SDK tools are simple, low-level command-line tools; using them provides you with a better understanding of how the Java platform actually works. Last, and perhaps most importantly, the SDK is available for free from Sun's Java Web site.

The SDK doesn't come with a text editor, so you'll need one of those as well. Any text editor will do, so long as it can save files in plain ASCII format. For example, on Windows, you can use Notepad or DOS Edit; on UNIX you can use `emacs` or `vi`; and on the Macintosh you can use SimpleText.

You may also want to download the entire example source now, to make it easier to follow along with the exercises in this tutorial. See Resources for links to download a Java SDK, a text editor, and the original source code for this tutorial.

Note: Since the purpose of this tutorial is to teach you the Java programming language, you will not need to compile any C or C++ programs to follow along with the exercises here. We will make use of C/C++ code from time to time, and you may find it instructive to compare C/C++ and Java code by running programs, but doing so is not a required exercise for this tutorial.

Historical background

The C programming language was developed in the early 1970s. C was originally designed to facilitate the writing of operating systems (OSs) and OS utility programs. At first, it was almost exclusively associated with UNIX. Later, C became a more generally used application development language across multiple platforms. In the middle of the 1980s, C became an official ANSII standard.

The C++ programming language was developed in the early 1980s. C++ was designed to add object-oriented programming techniques to the C language. Although C++ originally tended to be associated with systems programming, it has evolved into a mature programming language that is well-suited for a wide variety of application programming. In the early 1990s, C++ became an official ANSII and ISO standard.

The Java programming language and platform was developed in the early 1990s. The Java platform was originally designed to be used in consumer electronic devices (television sets, handheld devices, toasters, and the like), so the language had to be small, highly-portable, and efficient. Although the language never really caught on in digital devices, these same features made it ideally suited for the Internet. The Java language secured its place as an Internet technology after the Netscape Navigator and Internet Explorer Web browsers began to support Java applets. Since then, the Java language has continued to evolve and mature into a platform for enterprise application development.

Section 2. Setting up

Introducing the SDK

The Java Software Development Kit (SDK) is a group of command-line tools and packages that you will need to write and run Java programs. The most important of these tools are the Java compiler (`javac.exe`), which you use to compile Java programs, and the Java interpreter (`java.exe`), which you use to run Java programs. The SDK also includes the base classes (called the Java platform), which will provide you with the basic functionality and APIs you'll need start writing applications.

Sun has released an SDK for every one of its five major releases of the Java platform. Although I recommend that you get the latest version of the SDK (Java 1.4) for this tutorial, we'll briefly review all the versions.

- **Java 1.0** was the first public release of the Java platform. Because many Web browsers still use this version, many Java applets are still written to be compliant with Java 1.0.
- **Java 1.1** represented a vast improvement in the Java platform. Java 1.1 was the first Java platform stable enough to develop robust Java applications.
- **Java 1.2** was such a leap forward for the Java platform that it was officially dubbed Java 2. This version of the Java platform is specifically well-suited for enterprise application development.
- **Java 1.3** includes support for the Java Naming and Directory Interface (JNDI), Java Sound, and support for RMI over IIOP. It also includes a new, high performance just-in-time (JIT) compiler.
- **Java 1.4**, also known as Merlin, is the latest release of the Java platform. Java 1.4 includes support for XML processing, the Java Cryptography Extension (JCE), the Java Secure Socket Extension (JSSE), and the Java Authentication and Authorization Service (JAAS).

Note that Sun's development kit was called the Java Development Kit (JDK) prior to the release of Java 2. Thereafter, the development kit was officially renamed the Software Development Kit (SDK).

You can download the Java SDK of your choice for free from Sun's Java Web site (see Resources).

Installing the SDK

Once you download the SDK, you'll need to install it on your machine. The installation is pretty simple -- it should run just like most standard installation programs. If you're given the option between a *typical* or *custom* install, you should choose the typical install. You should only choose the custom install if you know exactly what you do and do not want to load on your machine.

You can download the API documentation for the Java platform separately, as a compressed file. This is a collection of HTML documents that you can navigate in a standard Web browser. Since the API documentation is an essential reference that you'll probably use a lot in the future, you may want to go ahead and get it now.

When you are installing, you'll usually be given the option of installing the source code for the standard Java platform classes. If you have sufficient memory on your machine, I recommend that you take this option. These files will give you a chance to look at the implementation of the classes that make up the Java language and standard APIs. These classes are particularly well designed and implemented, and you can learn a lot from studying this code.

After the SDK is installed, you may need to configure it to work on your system. How you configure the SDK will depend on your operating system and the SDK version you're using. Complete installation and configuration instructions will be provided when you download the SDK.

Section 3. Working with the SDK

Your first Java program

Before we begin talking about the structure and syntax of the Java language, let's just work with the SDK a little bit. We'll start by using the SDK's command-line tools to compile and run a Java program. Because the syntax of the Java programming language is very similar to C and C++, you should be able to follow most of the code in this non-trivial example.

In the next section, you will find the source code for a Java class called `Factorial`. The `Factorial` class computes the factorial of an integer. As you may recall, the factorial of a number n is the product of all integers from 1 to n. So, for example, the factorial of the number *5* is *5 x 4 x 3 x 2 x 1 = 120*.

In this exercise, you will pass in a value as a command-line argument and the `Factorial` class will attempt to compute the factorial of that number. As in C, command-line arguments are passed into Java applications as strings, so the `Factorial` class will attempt to transform the string argument into a valid integer. If you pass in non-digit characters, `Factorial` will generate an exception.

Factorial.java source

Java source files use the `java` extension, and every Java source code file must have the exact same name as the class that is defined inside of it. Since our first class is called `Factorial`, we need to save it in a file called `Factorial.java`.

Open your text editor and enter the source below exactly as you see it. When you are done, save it in a file called `Factorial.java`. You may save it in any appropriate directory on your machine. You'll need to go to this directory to use the command-line tools, so make sure it is convenient for you.

```
public class Factorial {
   public static void main(String[] args) {
      if(args.length != 0) {
         int num = Integer.parseInt(args[0]);
         System.out.println(factorial(num));
      }
   }

   private static int factorial(int fact) {
      int result = fact;
      if (fact == 0)
         return result;
      else {
         while (fact != 1)
            result *= --fact;
      }
      return result;
   }
}
```

Compiling the program

Once you've saved `Factorial.java` on your machine, you are ready to compile the program. The Java compiler that comes with the SDK is a command-line application called `javac.exe`. To compile a Java source code file, simply pass the name of the `.java` file to the `javac.exe` program. To compile your `Factorial` program, open a command-line prompt and change your directory to the location where you saved the `Factorial.java` file. Then type this command:

```
javac Factorial.java
```

Just like a C or C++ compiler, the Java compiler may generate any number of errors. Naturally, you'll need to correct all the errors before the Java compiler will successfully compile the `Factorial` program. Once the Java compiler is able to successfully compile, it will generate a class file called `Factorial.class`. This represents the executable that we'll run in the Java interpreter.

There are several options you can use with the `javac` compiler. Type `javac -help` at the command line to see a usage message and a list of valid options.

Running the program

The Java interpreter that comes with the SDK is a command-line application called `java.exe`. To run a Java bytecode executable, you simply pass the name of the Java program to the Java interpreter. Be sure that you *do not* specify the `.class` extension when using the Java interpreter. This program expects only class files, so it will produce an error if you explicitly write the `.class` extension.

To run your `Factorial` program, open a command-line prompt and change your directory to the location where you compiled the `Factorial.java` file. That's where your bytecode executable file, `Factorial.class`, should be. Then type this command:

```
java Factorial 5
```

The Java interpreter will try to execute the `main()` method of the `Factorial` program. A Java method is basically the same thing as a C/C++ function. The argument that we specified on the command line is `5`, and the Java interpreter will pass this argument into the `main()` method as a parameter -- specifically an array of `String` objects.

The Java interpreter may report a run-time error, which will usually terminate program execution. As in C and C++, Java run-time errors are more difficult to debug than compile-time errors. Since Java programs are executed in a virtual machine (that is, the Java interpreter) run-time errors can be handled in a graceful way. Whereas C and C++ programs may simply crash, the Java interpreter will at least report the run-time error that caused program execution to halt.

There are several options that you can use with the Java interpreter. Type `java -help` at the command line to see a usage message and list of valid options.

What you've learned about the SDK

We'll be compiling and running more Java applications in this tutorial, so let's go over the process once again. You may need to refer back to this section later.

1. Write your Java source code program in a text editor and save it with a `.java` extension. Make sure that your text editor saves the file in plain ASCII format, and make sure that it supports long file names. You can't save a Java program as a `.jav` file -- the extension has to be `.java`.

2. Compile your program from a command-line prompt, using the `javac` compiler that comes with the SDK. For example, for a source code file named `Sample.java`, you would type `javac Sample.java`. If all goes well, a Java class file will be produced. In our example, this file would be called `Sample.class`. Remember to *always* specify the `.java` extension when compiling a Java program.

3. Run your program from a command-line prompt, using the `java` interpreter that comes with the SDK. For example, to run the `Sample` program from the previous step, you would type `java Sample`. To specify command-line arguments to a Java program, simply type them after the program name, separated by spaces. Remember to *never* specify the `.class` extension when running a Java program.

4. Errors can occur when compiling or running a Java program. As you know, run-time errors are more difficult to debug than compile-time errors. When you are new to a language, however, compile-time error messages can seem very cryptic. Correcting compile-time errors can be very instructive, but if you can't get any of the examples in this tutorial to work, try using the example code (in Resources) instead.

Section 4. Introducing the Java language

Overview

Now that you have a basic idea of what Java code looks like and how to compile and run it on your test machine, we can begin to talk more in depth about the structure and syntax of the Java programming language.

In this section, we'll learn about the Java programming environment and the Java primitive data types. Because you are familiar with programming in C/C++, and because the Java language has much in common with these languages, we'll learn by comparison. In the sections that follow, we'll discuss the fundamental components of the Java platform, describing each one in terms of its relation to or difference from a similar component underlying the C/C++ programming framework.

C and C++ execution environments

C and C++ are high-level programming languages; their purpose is to make it easier for human beings to develop computer programs. Computers cannot understand high-level languages -- they can only understand low-level machine languages. A machine language consist of a sequence of binary instructions that can be directly executed on a computer's processor. For this reason, programs written in high-level languages must be translated into machine language programs, which are called *executables*, before they can be run on a computer.

Two methods are available for translating a high-level programming language into machine language executables: compilation and interpretation. *Compilation* involves translating an entire high-level program into a whole machine language program, which can then be executed in its entirety. *Interpretation* involves translating a high-level program into machine instructions line-by-line; one line is translated and executed before the next line is reached. Compilation and interpretation are logically equivalent, but compiled programs tend to execute faster than interpreted programs.

C and C++ programs are compiled into machine language executables by a program called a *compiler*. C compilers and C++ compilers are different. A C compiler can compile C source code files but not C++ source code files. Since C is retained as a subset of C++, a C++ compiler can compile both C and C++ programs.

Different computers use different machine languages. An executable that runs on one machine will not run on another machine that uses a different machine language. In order to run on different computers, a C or C++ source code file must be recompiled on different compilers; one for each type of machine, or *platform*, on which the executable will be run.

The Java execution environment

Like C and C++ programs, Java programs are compiled. Unlike C and C++ programs, Java programs are not compiled down to a platform-specific machine language. Instead, Java programs are compiled down to a platform-independent language called *bytecode*. Bytecode is similar to machine language, but bytecode is not designed to run on any real, physical computer. Instead, bytecode is designed to be run by a program, called a *Java virtual machine* (JVM), which simulates a real machine.

Simply put, the JVM is an interpreter that translates Java bytecode into real machine language instructions that are executed on the underlying, physical machine. More specifically, the term *Java virtual machine* is used generically to refer to any program that executes Java class files. The Java interpreter program, `java.exe`, is a specific JVM implementation. The Java Runtime Environment (JRE) is another example of a JVM implementation.

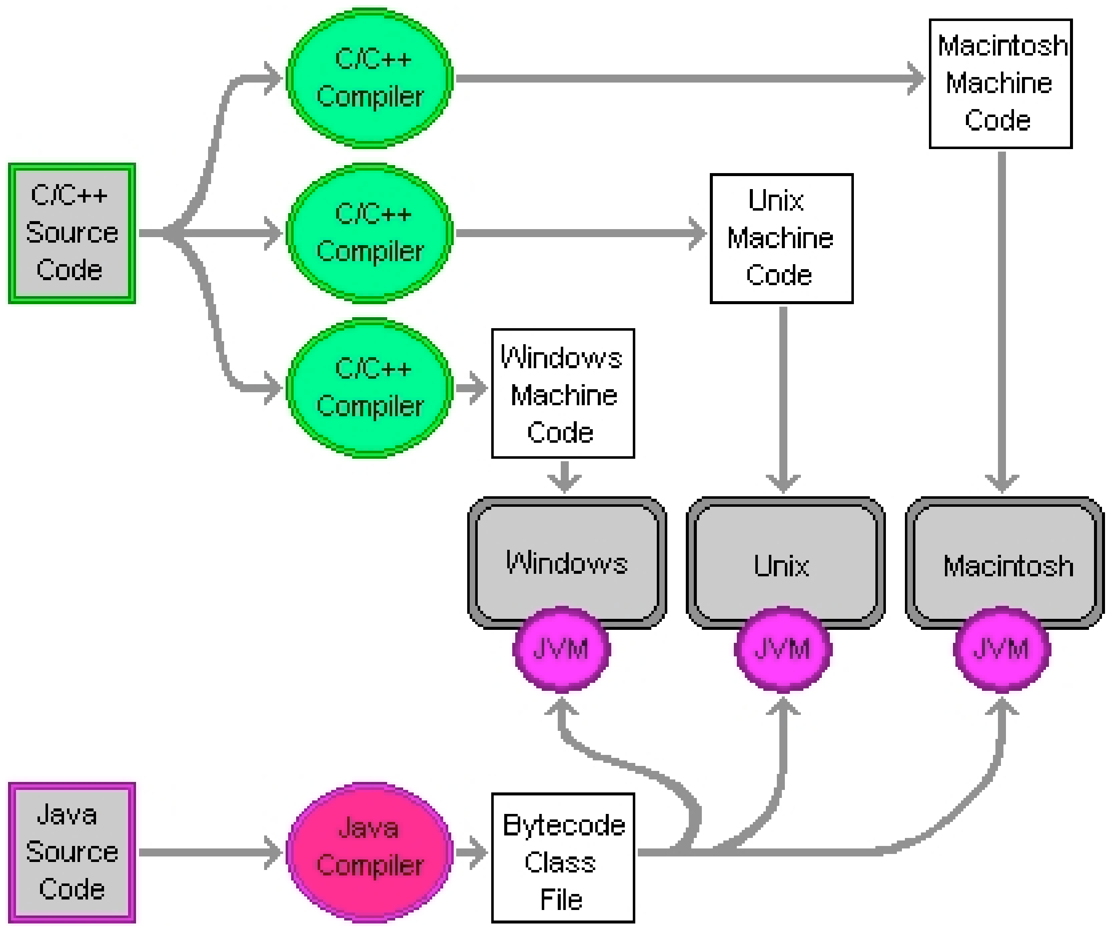
The Java platform uses the virtual machine layer to ensure that programs written in the Java language are platform independent. Once a Java program is compiled down to bytecode, it can be run on any system that has a JVM. Whereas a C or C++ program must be recompiled for each platform on which it is to be executed, a Java program needs to be compiled only once; it can then run on any machine that has a JVM installed.

Comparison of execution environments

Let's say you're going to write an application and you want it to run on three platforms: Windows, UNIX, and Macintosh. If you're writing the code in C or C++, you're going to have to compile your code three times, using the correct compiler for each of the three platforms. Of course, this is assuming that you can use the same code base without modification for each compiler. Very often, you'll need to modify your code to get it to compile, because you will probably use different APIs for each platform. In other words, you might have to write three different versions of your source code, one for each target platform.

Now, let's say you choose to write your application using the Java platform. In this case, all you have to do is write the Java code and compile it once. Since each of the three target platforms has a JVM implementation, you can run the same compiled bytecode on all of them, without modification. It doesn't matter if you compile the Java code on a Windows machine, a UNIX machine, or a Macintosh; it will run on any platform with a JVM.

The figure below illustrates how a program is compiled and executed in both the Java programming environment and a C/C++ environment:



Primitive data types

Many of the primitive data types defined by the Java programming language have names that are the same or similar to the fundamental data types defined in C. Despite the similarity in their names, all of the Java primitives are different from their C/C++ equivalents.

In general, C and C++ do not define strict sizes for their fundamental types. In other words, most of the aspects of the C/C++ fundamental types are defined by the compiler, so different compilers often represent fundamental types in different ways. You are forced to deal with this issue if you want your C/C++ code to be portable.

The Java programming language, on the other hand, guarantees the size, range, and behavior of its primitive types. No matter what Java compiler you use, the primitive data types will always be represented in the exact same way. This effectively isolates you from the details of the compiler's implementation.

The char types

```
// C char examples
char letterJ = 'J';
char letterA = 'A';
char letterV = '\126';
char digit0  = '\x030k';
char digit1  = '1';
char digit2  = '2';


// Java char examples
char letterJ = 'J';
char letterA = 'A';
char letterV = '\u0056';
char digit0  = '\u0030';
char digit1  = '1';
char digit2  = '2';
```

Both the C and Java languages have a `char` type, which is used to represent characters. Although these types have the same names and uses, they are represented in completely different formats.

The C `char` type is used to hold a character. The compiler determines the character set and the size of the values that are stored in the `char` type. Typically, the C `char` type represents an ASCII character, and uses 8 bits (including the sign bit), yielding a range of values from -128 through 127. Because there is no standard, however, many problems can arise from using the C `char` type, which causes many portability issues.

The Java `char` type is also used to store a character, but in this case the language strictly defines the character set and size of the `char` type. A Java `char` represents a character from the Unicode character set, and is stored in 16 bits (with no sign bit), yielding a range of 0 through 65,536. As in C, character literals are written within single quotes.

The Unicode character set is capable of representing most of the characters used in the world's major written languages. You can encode a Unicode character in an ASCII file using the `\u` escape sequence, followed by a hexadecimal number representing a Unicode character. See Resources to learn more about the Unicode specification.

Character escape sequences

As in C, Java characters and strings can use escape sequences to represent special characters. Most of these escape sequences are the same as those defined in C, but not all C escape sequences are valid Java escape sequences. The following table shows the valid Java escape sequences that are inherited from C:

Escape sequence	Character value
`\b`	Backspace
`\t`	Horizontal tab
`\v`	Vertical tab
`\n`	New line
`\b`	Backspace
`\f`	FormFeed
`\r`	Carriage return
`\"`	Double quote
`\'`	Single quote
`\\`	Backslash

In addition to the sequences above, C and C++ also define the alert (`\a`), question mark (`?`), and hexadecimal and octal number escapes (`\x`*hhh* and `\`*ooo* respectively). These are not valid Java escape sequences.

Java and C integer types

```
// C integers
long  int val_1 = 250000;
long      val_2 = 0x36B;
int       val_3 = -1800;
short int val_4 = 017;
short     val_5 = -25;


// Java integers
long  val_1 = 250000;
long  val_2 = 0176;
int   val_3 = 0x3F;
short val_4 = -93;
short val_5 = 25;
byte  val_6 = 120;
byte  val_7 = -34;
```

Both C and Java define an integer type named `int`. In addition, the Java language also defines the `long` and `short` types, which originate from the type modifiers of the same names in C and C++. The Java integer type `byte` has no counterpart in C. You can write Java integer values as octal or hexadecimal numbers using the same format as C/C++. Octals start with a `0` and hexadecimals start with a `0x`.

The `int` type in C is used to store a signed whole number value. The exact size of the `int` type is specified by the compiler, but in general the `int` type is 16 bits (including a sign bit), yielding a range of -32,768 through 32,767.

The Java `int` type is also used to store a signed whole number. The Java language strictly defines the size of the `int` type as 32 bits (including a sign bit), yielding a range of -2,147,483,648 through 2,147,483,647.

The C and C++ languages define a set of type modifiers, which have an effect on the way the compiler stores an `int` value. The `long` modifier usually forces the compiler to use 32 bits to represent an `int`, and the `short` modifier usually forces the compiler to use 16 bits to represent an `int`. C also defines the `signed` and `unsigned` modifiers, which have no counterpart in the Java language; Java integers are always signed values.

The Java `long` type is used to store a signed whole number using 64 bits (including a sign bit), yielding a range of -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807. The Java `short` type is used to store a signed whole number using 16 bits (including a sign bit), yielding a range of -32,768 through 32,767. The Java language also defines the `byte` type, which is used to store a signed whole number using 8 bits (including a sign bit), yielding a range of -128 through 127.

The floating-point types

```
// C floating-points
float  val_1 = 0.25f;
float  val_2 = 12.4901f;
float  val_3 = 25.138e-10f;
double val_4 = 0.123456789;
double val_5 = 1.9876540e5;
double val_6 = 0.000001234;


//Java floating-points
float  val_1 = 0.25f;
float  val_2 = 12.4901f;
float  val_3 = 25.138e-10f;
```

```
double val_4 = 0.123456789;
double val_5 = 1.9876540e5;
double val_6 = 0.000001234;
```

Both C and Java define floating-point types named `float` and `double`, which represent single- and double-precision floating-point values, respectively. Java floating-point types are, however, represented in a completely different format from their C/C++ counterparts. As in C, Java floating-point values can be represented with an exponential portion.

The `float` type in C is used to store a signed floating-point number value. The exact size of the `float` type is specified by the compiler, but in general the `int` type is 32 bits (including a sign bit), yielding a range of approximately -34.4E-38 through 3.4E+38. The `float` type can generally be used safely to store values of six to seven digits of precision.

The Java `float` is also used to store signed floating-point number values, but the language specifies that 32 bits (including a sign bit) are *always* used to store IEEE 754 floating-point values. Floating-point literals are assumed to be of type `double`, so if you specify a `float` literal you need to append the letter `f` or `F`.

The `double` type in C is used to store a signed floating-point number value. The exact size of the `double` type is specified by the compiler, but in general the `double` type is 64 bits (including a sign bit), yielding a range of approximately -1.7E-308 through 1.7E-308. The `double` type can generally be used safely to store values of 14 to 15 digits of precision.

The Java `double` is also used to store signed floating-point number values, but the language specifies that 64 bits (including a sign bit) are *always* used to store IEEE 754 floating-point values. Floating-point literals are always assumed to be of type `double`, but you can append the letter `d` or `D` if you wish.

The boolean types

```
// C boolean
bool on  = true;
bool off = false;
bool yes = 0;
bool no  = 1;


// Java boolean
boolean on  = true;
boolean off = false;
```

```
// This is illegal
boolean yes = 1;
// This is illegal
boolean no  = 0;
```

Both the C++ and Java languages have *boolean* types, which are called `bool` and `boolean`, respectively. Although these types have similar names and uses, they are represented in completely different formats.

Later implementations of C++ include a new `bool` type, whose values are represented by the new keywords `true` and `false`. In actuality, the `bool` type is represented as an `int`, and `true` and `false` correspond to 1 and 0 respectively. You can use `int` values and `bool` values interchangeably; 0 is converted to `false`, and all other number values are converted to `true`.

The Java language defines the `boolean` type, whose values are represented by the `true` and `false` literals, which are the only valid values of the Java `boolean` type. Unlike the C++ `bool` type, the `boolean` type cannot be converted to or from the `int` type. In fact, the only valid conversion for a `boolean` value is to or from another `boolean` value.

In Java programs, you *cannot* use `int` type values or expressions in place of `boolean` type values or expressions. For example, if you use an `int` in an `if` statement, which evaluates a `boolean` expression, the Java compiler will generate an error. This is a major change from C and C++, both of which use `int` values in logical expressions.

Operators

You will probably be happy to learn that the Java language defines the same arithmetic and logical operators that C and C++ define, and their behaviors are very nearly always comparable. The only notable exception is that the Java + operator is overloaded so that it can concatenate `String` objects. If you use the + operator with a `String` and another operand that is not a `String`, the other operand is converted into a `String`. For example

```
"To be, " + "or not to be."  // results in "To be, or not to be."
1 + "2" + 1                  // results in the new String "121"
```

Java operators will return different results than their C/C++ equivalents in a number of cases. For example, the Java divide operator (`/`) will generate an exception if you try to divide an integer by zero. Also, the Java language defines positive and negative zeros, positive and negative infinities, and not-a-number values (`java.lang.Float.NaN` and `java.lang.Double.NAN`). No floating-point operations produce an exception -- one of these values is returned instead.

Positive zero and negative zero compare equal, but other operations distinguish between positive and negative zeros. So, `1.0/0.0` results in positive infinity and `1.0/-0.0` results in negative infinity. But the expression `0.0==-0.0` is true and the expression `0.0>-0.0` is false. Because `NaN` is unordered, all of the comparison operators will return `false` if either operand is `NaN`, except for `!=`, which will always return true if either operand is `NaN`.

Note: In C++, you can overload operators when you define a class. For example, if you write a class to represent a matrix, you can overload the + operator so that it can perform matrix addition correctly on two matrix objects. The Java language does not allow programmers to overload any operators.

C/C++ functions versus Java methods

```
// C/C++ functions
// A Java method
void funct(void) {
  //implementation
  }
void funct() {
  //implementation
  }
void funct()
  {
  //implementation
  }
```

C and C++ allow you to define functions. In C++, you can define functions as members of a class. In Java terminology, functions are called *methods*. Methods can *only* be declared as members of a class; you can't define a method outside of a Java class.

Both C and C++ allow you to use the `void` keyword as the return type of a function that does not return a value. The Java language uses the `void` keyword for the exact same purpose; when a Java method returns no value, its return type is declared to be `void`. In C and C++, you can define a function that takes no parameters in one of two ways: by using empty parentheses or by using the `void` keyword in between parentheses. The Java language does not allow you to use the `void` in this way; any Java method that accepts no parameters must be declared with empty parentheses.

C and C++ allow you to define functions that take a variable-length parameter list, by using the ellipses (`...`) notation. The Java language does not provide this facility, nor is there a replacement for this feature. In general, passing in an array parameter can serve a similar purpose.

Arrays

Java arrays are similar to C arrays, but there are important differences between them. Java arrays are objects, so they are declared using the `new` operator. Since Java arrays are objects, they have *attributes*, the most important of which is the `length` attribute, which you can use to determine the size of the array. Also, the bracket characters (`[ ]`) that are used to indicate arrays are bound to the array type, not the array name. Array literals look the same in both languages, as you can see below.

```
// C/C++ arrays                         // Java arrays
int scores[100];                        int[] scores = new int[100];
char grades[] = {'A', 'B', 'C'};        char[] grades = {'A', 'B', 'C'};
int table[2][2] = {{1, 2},{3, 4}};     int[][] table = {{1, 2},{3, 4}};
```

The Java interpreter guarantees that an array will not be accessed outside of its bounds. If you try to read an array index outside of the array's actual size, the Java interpreter will throw a `java.lang.ArrayIndexOutOfBounds` exception.

Strings

C uses a null-terminated sequence of `char` values to represent strings. Java strings are represented by objects of the `String` class. Both C and Java string literals are represented as a sequence of characters enclosed in double quotes.

There are two ways to make a `String` object: you can use a string literal, or you can use a constructor. `String` objects are immutable, which means that once a `String` is given an initial value it cannot be changed. In other words, if you want to change the value of a `String` reference, you need to assign the reference a new `String` object.

Since Java strings are objects, you interact with them through the interface defined by the `String` class. The `String` class has a rich interface, with quite a few useful methods. Some of the most commonly used methods are demonstrated in the code below. Consult the API documentation for a detailed description of the `String` class and all the methods it defines (see Resources).

```
/*
 *  The StringTest class simply demonstrates
 *  how Java Strings are created and how
 *  String methods can be used to create
 *  new String objects. Notice that when you
 *  call a String method like toUpperCase()
 *  the original String is not modified. To
 *  actually change the value of the original
 *  String, you have to assign the new
 *  String back to  *  the original reference.
 */
class StringTest {
    public static void main(String[] args) {
        String str1 = "Hi there";
        String str2 = new String("Hi there");

        System.out.println(str1 == str2);
        System.out.println(str1.equals(str2));

        System.out.println(str1.toUpperCase());
        System.out.println(str1.toLowerCase());
        System.out.println(str1.substring(1,4));
        System.out.println(str1.trim());

        System.out.println(str1.startsWith("Hi"));
        System.out.println(str1.endsWith("there"));
        System.out.println(str1.replace('i', 'o'));
    }
}
```

The main() method

Like C and C++, Java applications must define a `main()` method in order to be run. In Java code, the `main()` method must follow a strict naming convention. All `main()` methods must be declared as follows:

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

Note: Actually, you can reverse the `public` and `static` modifiers, and the `String` array can be named anything you like. Note, however, that the above format is conventional.

Passing arguments to the main() method

Command-line arguments are passed into a Java application as parameters to the `main()` method, just as they are in C. Instead of coming in as a `char` array like `argv`, however, arguments come into Java programs as an array of `String` objects. Because you can determine the length of a Java array, there is no need for an `int` parameter such as C's `argc`. Also unlike C, the first element in the array (at index 0) is the first argument, not the name of the program.

The C and Java `main()` methods below are similar. They both take the command-line arguments and print them back to the screen. Note the differences between them.

```
// A C main() function                     // A Java main() method
void main(int argc, char* argv[])          public static void main(String[] args) {
{                                          {
    int i;
    for(i=0; i<argc; i++)                   for(int i=0; i<args.length; i++)
        printf("%d: %s\n", i, argv[i]);           System.out.println(i +": " + args[i]);
}                                          }
```

Other differences

We've gone over the major syntactic differences between Java and C/C++. What remains is a handful of lesser differences, which are briefly outlined below.

- **Pointers**: The Java language does not include pointers. The reason for this is simple: pointers tend to cause confusion in the code, and they are a common source of bugs. Java references are pointers to Java objects, but you can't use Java references in the same way that you can use C pointers. Java references cannot be incremented or decremented, you can't convert references to or from primitive types, and there are no *address of* operators, such as `&`, or dereferencing operators, such as `->` or `*`.
- **Global variables**: Unlike C and C++, the Java language offers no way to declare global variables (or methods).
- **The preprocessor**: The Java platform does not have a preprocessor, nor does it have any preprocessor directives or macros.
- **goto**: Although the Java language reserves `goto` as a keyword, there is no `goto` statement like the one used in C.
- **struct**, **union**, **typedef**, **enum**: The Java language does not have the `struct` or `union` types, nor does it have the `typedef` or `enum` keywords.

- **Freely placed variables**: Java variables can be declared anywhere, as they are needed. Variables do not need to be grouped together at the top of a block, as they do in C.
- **Freely placed methods**: C requires that function declarations appear before they are invoked, but Java methods can be invoked before they have been declared.
- **Garbage collection**: The Java platform uses a garbage collector to automatically reclaim memory by recycling objects when they are no longer referenced. The `malloc()` and `free()` functions used by C aren't necessary in Java programming, and no similar methods exist for the Java language.

Section 5. Classes and objects

Introduction to object-oriented programming

The Java programming language is object oriented. Because it uses syntax that is generally very similar to C++, Java classes and objects are generally easy to learn if you come from a C++ background. If you come from a C programming background, however, you may know very little about object-oriented programming. In this section we'll start off with a brief, C-based introduction to object-oriented programming. Next, we'll build a simple bank account application, starting with a C implementation, migrating to a C++ implementation, and finishing the section with a Java implementation. Each implementation will build on the one before it, which should leave you with a good grasp of both the advantages and shortcomings of each language used.

Working with classes

You can think of a *class* as a data type that is defined by a programmer. Variable instances of a class are called *objects*. Like other variables, objects have a type, a set of attributes, and a set of operations. The *type* of an object is represented by the class from which the object was instantiated. The *attributes* of the object represent its value or state. The *operations* of an object are the set of possible functions that can be invoked on an object in order to change its state.

Consider C's `int` fundamental data type, which represents an integer. This type's name, of course, is `int`, and you use this type name to create variables that are instances of an integer. Every `int` variable has one attribute, which represents the integer number that the variable holds. Every `int` variable also has the same set of operations, which you can use to change the state (or value) of the variable. Some examples of the operations that can be performed on an `int` variable include: addition (`+`), subtraction (`-`), multiplication (`*`), division(`/`), and modulo (`%`).

An account struct

Now, let's examine a situation where you might want to develop your own type, which will represent a complex object that the C language doesn't support as a fundamental type. Suppose you are part of a team developing software for a financial institution, and your job is to develop code to represent a typical bank account. A bank has several accounts, but each account has the same basic set of attributes and operations. In particular, an account has a balance and an ID number. (Different types of accounts have different attributes and operations, but we'll keep this example both general and simple.)

Now, if you're programming in C, how do you go about creating a new `Account` module to represent a bank account in your program? You can represent an account's attributes by using a `float` variable for the balance, and an `int` variable for the ID number. But we need to group these variables together to create a single coding module or structure. The obvious way to do this is to use a `struct`, as shown below.

```
struct Account
{
   float balance;
   int id;
};
```

Account functions

The valid operations for an account are depositing an amount and withdrawing an amount. C provides a built-in way to define operations, namely *functions*. We can create two functions that implement these operations, called `deposit()` and `withdraw()`, as shown below. (Naturally, you can't withdraw an amount of money that is greater than the balance of the `Account`, so the withdrawal operation should not change the account balance at all if an attempt is made to overdraw the account.)

```
void deposit(struct Account *account, float amount)
{
    (*account).balance += amount;
}

void withdraw(struct Account *account, float amount)
{
    if((*account).balance >= amount)
        (*account).balance -= amount;
}
```

Grouping operations with data

Although you can use the C `struct` to group data, a `struct` can only have data members. Functions cannot be members of a `struct`. This leaves us with a problem, because we'd like to group our `Account` operations with our `Account struct`.

One solution is to use function pointers. We can include function pointers as members of the `Account struct`, and use these pointers to invoke the correct `Account` operations.

To use function pointers, we need to initialize them so that they point to the correct functions. One way to do this is to write a special initialization function, which will construct a new `Account struct`. Such a function is called a *constructor*. Besides initializing the function pointers, we can also use this constructor function to give the `Account` an initial state. In particular, we can pass in the account ID and an initial balance.

After initialization, the account ID will never change, and the balance will only be changed by the `deposit()` and `withdraw()` functions. Of course, the `id` and `balance` members can be accessed directly, so their values can be changed directly. Unfortunately, there is no way to limit the access to the members of a C `struct`.

In the next section, *Account.c* shows a C implementation of an `Account` struct.

Account.c

```
#include <stdio.h>

void depositAmount(struct Account*, float);
void withdrawAmount(struct Account*, float);
```

```
struct Account {
    float balance;
        int id;
    void (*deposit)(struct Account*, float);
    void (*withdraw)(struct Account*, float);
};

struct Account initAccount(int id, float new_balance) {
    struct Account account;
    account.balance = new_balance;
    account.id = id;
    account.deposit = &depositAmount;
    account.withdraw = &withdrawAmount;
    return account;
}

void depositAmount(struct Account *account, float amount) {
    (*account).balance += amount;
}

void withdrawAmount(struct Account *account, float amount) {
    if((*account).balance >= amount)
        (*account).balance -= amount;
}
```

Using the Account

Next, let's take a look at how we can use the `Account struct`. Here's an example of a `main()` function, which shows how an `Account` is created and used.

```
void main(int argc, char* argv[])
{
    struct Account my_account = initAccount(1002552, 5000.00);
    my_account.deposit(&my_account, 2000);
    printf("Balance: %f\n", my_account.balance);
    my_account.withdraw(&my_account, 3000);
    printf("Balance: %f\n", my_account.balance);
    my_account.withdraw(&my_account, 6000);
    printf("Balance: %f\n", my_account.balance);
}
```

First, we create an `Account` instance using the `initAccount()` function. Because we pass in `1002552` and `5000.00`, the `Account` will have an ID of 1002552 and an initial balance of $5000.00.

Next, we use the `deposit` function pointer to call the `depositAmount()` function. We pass in $2000.00, so when we print out the balance in the next line, it will be $7000.00. Next, we use the `withdraw` function pointer to call the `withdrawAmount()` function. We pass in $3000.00, so when we print out the balance in the next line, it will be $4000.00. Finally, we try to withdraw another $6000.00, but because this amount is larger than the account's total balance, the `withdrawAmount()` function does nothing.

Benefits of objects

A class is more than just a C `struct` with functions. There are a great many other features of classes that we can't easily simulate in C. Here are three of the primary benefits of using classes and objects:

- **Encapsulation or information hiding** refers to a way of treating an object like a "black box," which means that you can use an object without knowing (or caring) how it is implemented. Using objects through the interface defined by the methods (operators) defined in the class ensures you can change the class implementation without breaking any code that uses objects of that class.
- **Polymorphism** refers to the ability to associate different features to the same name, together with the ability to choose the right feature based on context. The most common example of polymorphism is method overloading, whereby you can define several methods that have the same name, as long as they take different parameters.
- **Inheritance** refers to reusing code by writing new classes that *extend* an existing class. For example, let's say you wanted to write a new class to represent a checking account. Since a checking account is a special kind of bank account, you could write the `CheckingAccount` class so that it extended (or subclassed) the `Account` class. Then, the `CheckingAccount` class would automatically get the state and all the operators (functions) of the `Account` class. You would only need to add the new state and operators specific to the `CheckingAccount` class. For example, you might add a `cashCheck()` function to perform the operation of cashing a check that was written for the checking account. If required, you could also change the inherited state or behavior of the subclass. For example, a user might be allowed to overdraw on his checking account, so you would need to override the `withdrawal` function.

Defining a C++ class

Now, let's code a real C++ class that represents an `Account`. Even if you're not an experienced C++ programmer, you can see that the `Account` class is very similar to our `Account struct`. The source code for `Account.cpp` will be shown in the next section.

Notice that a class has data and function members, and each member has either `public` or `private` *access*. As you might guess, `public` members can be accessed outside of the class, while `private` members can be accessed only inside of the class definition. Since the `balance` field is `private`, we need a special *getter* function, called `getBalance()`, which we can use to query the state of the balance. Notice, however, that you can't directly access or change the `balance` variable outside of the class. We can only change the `balance` using the `deposit()` and `withdrawal()` functions. If you try to access the balance member outside of this class using `account.balance`, the compiler will generate an error.

Also notice the special function called `Account`. This is a special constructor function, similar to our `initAccount()`, which we use to initialize new `Account` objects. Note that a constructor always has the same name as the class does, and that it *does not* declare a return type. We use the constructor to set the initial balance of the `Account`.

Finally, let's look at the `main()` function and see how we can use our `Account` class. First, we declare an `Account` object, called `account`. We implicitly call the constructor by following the `account` name with parentheses containing the initial balance. Now we can call the `deposit()` and `withdrawal()` functions to change the balance of our account. When we want to print the balance of the `account`, we use the `getBalance()` function. We can't get the balance directly, using `account.balance`, because it is a private member.

Account.cpp

```
class Account {
    private:
        double balance;

    public:
        Account(double start_balance) {
            balance = start_balance;
        };

        void deposit(double amnt) {
            balance += amnt;
        };

        void withdrawal (double amnt) {
            if (balance >= amnt)
            balance -= amnt;
        };

        double getBalance() {
            return balance;
        };
};
```

```
void main(void)
{
    Account account(5000.00);
    account.deposit(2000.00);
    printf("Balance: %f\n", account.getBalance());
    account.withdrawal(4000);
    printf("Balance: %f\n", account.getBalance());
}
```

Defining a Java class

We'll close this section by implementing the `Account` class in the Java language. The source for `Account.java` will be shown in the next section. As you will see, defining a Java class is very similar to defining a C++ class. The major differences are as follows:

- The `public` and `private` keywords are modifiers, not labels. Each member must have its own `public` or `private` modifier.
- You don't use semicolons (`;`) after the closing brackets in class and method definitions.
- The `main()` method is a member of the class -- all Java methods must be members of a class. Unlike the C++ example, you can access the private members of the Java `Account` class in the `main()` method, because it is a member of the class.
- You call the `Account` constructor using the `new` keyword, followed by the name of the class and the parameterized list of arguments for the constructor.

Account.java

```

class Account {
    private double balance;

    public Account(double balance) {
        this.balance = balance;
    }

    public void deposit(double amnt) {
        balance += amnt;
    }

    public void withdrawal (double amnt) {
        if (balance >= amnt)
            balance -= amnt;
    }

    public double getBalance() {
        return balance;
    }

    public static void main(String[] args)
    {
        Account account = new Account(5000.00);
        account.deposit(2000.00);
        System.out.println("Balance: " + account.getBalance());
        account.withdrawal(4000);
        System.out.println("Balance: " + account.getBalance());
    }
}
```

Section 6. Java classes in depth

Overview

Having discussed the general role of classes and objects in an object-oriented programming framework, we're ready to delve more deeply into the specifics of the Java platform's class structure and implementation.

In this section, we'll talk about the following:

- **Class members**: A class member is always either a *field* or a *method*. A field represents data, and a method represents operations. Classes can define any number of members.
- **Access modifiers**: Class members are declared with *access modifiers*, which specify the accessibility of the member outside of the class in which it is defined. For example, members that are declared `private` cannot be accessed at all, but `public` members are freely accessible.
- **Objects**: Classes are really just definitions. What we really use in code are class instances, and these are called *objects*. We'll learn how to create objects from classes.
- **Constructors**: A *constructor* is a special operator that is used to create objects. In general, a class is not of much use if you can't create objects of that class. Constructors are very important because they provide the ability to create new class instances.

- **The 'this' keyword**: Java objects implicitly reference themselves. It is important that you understand how to use the `this` keyword for this purpose.

Class members

A Java class is an independent module of code that defines attributes and operations in terms of *members*. Java classes have four kinds of members: fields, methods, inner classes, and inner interfaces. We'll talk here about the first two member types, leaving the discussion of inner classes and interfaces to the section on inheritance.

A *field* is a variable declared inside of a class. Java fields come in two varieties: instance variables and class variables. *Instance variables* are associated with each instance of a class, and each instance has its own copies of instance variables. *Class variables*, which are declared using the `static` keyword, are associated with the class as a whole, and the class shares a single class variable with all class instances. For example, the `balance` field in a `BankAccount` would be an instance field because each `BankAccount` instance has its own `balance`, which is independent of every other `Account` object's `balance`. On the other hand, the `interest` field would be declared as a class field because every `BankAccount` object uses the same interest rate, which is the same for every `BankAccount` object.

A *method* is a function declared inside of a class. Java methods also come in two varieties: instance methods and class methods. Every class instance gets its own copy of *instance methods*, but there is only one copy of a *class method*, which is shared among all class instances. Class methods are declared using the `static` keyword. Instance methods are used to operate on instance variables and class methods are used to operate on class variables. For example, a `deposit()` method in our `BankAccount` class would be an instance method because each `BankAccount` has its own `balance` field, which the `deposit()` method would change. The `setInterest()` method would be declared as a class method because every `BankAccount` shares a single `interest` field, which the `setInterest()` method would change.

s
One final word about class members. Instance members (that is, instance variables and instance methods) are accessed, using the dot (`.`) notation like that of a C `struct`, using an object name. Class members (that is, class variables and class methods) are accessed using a class name.

BankAccount.java

The program below, called `BankAccount`, has five members. Two members are fields: `balance`, which is an instance field, and `interest`, which is a class field. Three members are methods: `deposit()` and `withdraw()` are instance methods and `setInterest()` is a class method. Notice that you use an object name to access instance members, and the class name to access class members.

```
class BankAccount {
    float balance;          // an instance field
    static float interest;  // a class, or static, field

    // an instance method
    void deposit(float amount) {
        balance += amount;
    }

    // an instance method
    void withdraw(float amount) {
        balance -= amount;
    }

    // a class, or static, method
    static void setInterest(float interestRate) {
        interest = interestRate;
    }

    public static void main(String[] args) {

        // create a new account object
        BankAccount account = new BankAccount();
        // deposit $250.00 into the account
        account.deposit(250.00F);
        // set interest rate for all BankAccount objects
        BankAccount.setInterest(5.0F);
    }
}
```

Access modifiers

Like C++, the Java language allows you to set the visibility of a class member. Java members use the `public` modifier to indicate that a member is freely accessible, both inside of the class and outside. Java members use the `private` modifier to indicate that the member is only accessible inside of the class. Outside of the class, `private` members are inaccessible.

Let's consider our `BankAccount` class again. We want programmers using `BankAccount` objects to change the `balance` by using the `deposit()` and `withdraw()` methods. We'll declare these methods to be `public`, so they can be invoked in code outside of the `BankAccount` class. We do not, however, want programmers to change the `balance` field directly, so we'll make the `balance` field `private`.

You might be wondering what the default access level is. That is, what is the access level for a class member that is not declared using the `public` or `private` modifiers? You may suspect that the default access level is `public`, since `public` is the default access level in C++. In fact, the default access level is called *package* access, because only classes in the same package have access to these class members. The only way to declare a member with package access is to use no access-modifier keyword at all.

The Java language defines one further access level; this access level is borrowed from C++ and is called `protected`. The `protected` modifier is used when you want a member to be accessible in subclasses. We'll discuss `protected` class members when we talk about inheritance.

BankAccount.java with access modifiers

In the code listing below, you'll note that the `balance` field is declared using the `private` access modifier. We don't want any programmers accessing an account's `balance` field directly. Instead, we want them to change this field using the `deposit()` or `withdraw()` methods, which are declared `public`. The same is true of the `interest` field, which is `private`, and the `setInterest()` method, which is public.

```
class BankAccount {

    private float balance;
    private static float interest;

    public void deposit(float amount) {
        balance += amount;
    }

    public void withdraw(float amount) {
        balance -= amount;
    }

    public static void setInterest(float interestRate) {
        interest = interestRate;
    }

    public static void main(String[] args) {

        // create a new account object
        BankAccount account = new BankAccount();

        // deposit $250.00 into the account
        account.deposit(250.00F);
        // set interest rate for all BankAccount objects
        BankAccount.setInterest(5.0F);
    }
}
```

Creating objects

Looking at the `main()` method of the `BankAccount` class, you can see that we created a new `BankAccount` object, like this:

```
BankAccount account = new BankAccount()
```

First, we declare an object (that is, a variable) of type `BankAccount`. As you can probably guess, the `new` keyword is used to set aside enough memory to create a new object. The new object is actually created by this statement: `BankAccount()`. This statement looks like a method call, but we didn't declare a method with this name, so you may wonder what the statement is actually doing.

This is, in fact, a constructor call. A *constructor* is a special operator that is used to create objects. You can't create Java objects without a constructor, so if you write a class without a constructor, the compiler will create a default one. That's why we can call `BankAccount()`, even though we didn't explicitly write a constructor in the `BankAccount` class.

Strictly speaking, a constructor is not a kind of method because methods are class members and constructors aren't. Class members, like fields and methods, are inherited in subclasses -- constructors are never inherited.

Java versus C++ constructors

Java constructors are declared in much the same way as C++ constructors. Java constructors do not have return types; all constructors implicitly return a new object of the class in which it is defined. Every Java constructor must have the exact same name as the class in which it is declared. Otherwise, constructor declarations are pretty much the same as method declarations. In particular, constructors can take parameters, just like Java methods.

Let's write our own constructor for our `BankAccount` class. When we first create a `BankAccount` object, we'll want to initialize that `BankAccount` object by giving it an initial balance, so our constructor will take a `float` parameter. The body of the constructor will simply set the `balance` field of the newly created `BankAccount` object to the specified value.

You can create as many constructors as you want in a class. If you write a constructor, however, the Java compiler *will not* create a default constructor for you. So, once we write our `BankAccount(float initBalance)` constructor we can't create `BankAccount` objects using the `new BankAccount()` statement. We'll have to use a statement like `new BankAccount(1000.00F)`, which will create a `BankAccount` object with an initial balance of $1000.00.

BankAccount.java with a constructor

Below is the `BankAccount` class with a constructor added:

```
class BankAccount {

    private float balance;
    private static float interest;

    public BankAccount(float initBalance) {
        balance = initBalance;
    }

    public void deposit(float amount) {
        balance += amount;
    }

    public void withdraw(float amount) {
        balance -= amount;
    }

    public static void setInterest(float interestRate) {
        interest = interestRate;
    }

    public static void main(String[] args) {

        // create a new account object
        BankAccount account = new BankAccount(500.00F);

        // deposit $250.00 into the account
        account.deposit(250.00F);

        // set interest rate for all BankAccount objects
        BankAccount.setInterest(5.0F);
    }
}
```

The this keyword

The Java language uses the `this` keyword to reference the current object. You can use the `this` keyword to explicitly refer to fields, methods, and constructors in the current class.

A common use for the `this` keyword is to resolve variable scope issues. For example, our `BankAccount` class has a field called `balance`. Let's say we want to write a method called `setBalance(float balance)`, which will set the `balance` field of our object. The problem is that inside of the `setBalance(float balance)` field, when we refer to `balance`, we are actually referring to the `balance` parameter, not the `balance` field. We can explicitly refer to the field by using the `this` keyword.

```
public void setBalance(float balance) {
    this.balance = balance;
}
```

Another use for this

Another use of the `this` keyword is to call a constructor from within the body of another constructor. For example, say that we want to write a constructor for our `BankAccount` class that takes no parameters and sets the balance to $0.00. We could do this by calling the constructor that we already wrote. To call a constructor inside the body of another constructor, use the `this` keyword, followed by the parameters for the constructor you want to call. You can only use `this` to call constructors inside of another constructor body, and it must be the first statement in the constructor body.

```
public BankAccount() {
    this(0.0F);
}
```

What you've learned about Java classes

We'll close this section with a quick review of the important points we've covered, as follows:

- **Class members**: Java class members are *fields* and *methods*. Fields represent data, and methods represent operations. A `class` is a declaration of a *class* of objects, which are defined in terms of the `class`'s members.

- **Access modifiers**: You use *access modifiers* to limit the visibility of class members and constructors outside of the class in which they are defined. Most often, you'll encapsulate all the data in a class by declaring class fields `private`, and you'll define the interface of a class by writing `public` methods.
- **Constructors**: You define *constructors* as a way to let programmers create instances of your class. Generally, you'll define constructors that make it easy for a programmer to create an object that is initialized properly. Very often, you'll define several constructors that call other constructors using the `this` keyword.

Section 7. Inheritance

Overview

We'll close this tutorial with a discussion of inheritance. Inheritance is one of the most important benefits of object-oriented programming, and it is important that you understand it correctly in order to use it to the greatest effect.

Here's what we'll cover in this section:

- **The `extends` keyword**: Inheritance is defined when a class is declared. You use the `extends` keyword to specify the superclass of the class you're writing.
- **Constructors**: Constructors are not inherited in subclasses, but you will very often invoke the constructors of superclasses in your subclass's constructors.
- **Overloading/overriding**: *Overloading* refers to writing several methods with the same name but different parameters. *Overriding* refers to changing the implementation of a method that is inherited in a subclass.
- **The Object class**: All Java objects ultimately inherit from the `Object` class, which defines the basic functionality that every Java object is guaranteed to have.
- **Interfaces**: An *interface* is a description of behavior that provides no

implementation.

- **Inner classes and interfaces**: Sometimes you may need to write a class or interface that will only be used within another class or interface. The Java language allows you to declare and use *inner classes* and *inner interfaces* for this purpose.

Extending classes

In C++, a class can inherit from any number of classes, but Java classes can only *extend* one class. In other words, C++ allows multiple inheritance, but the Java language only allows single inheritance.

Basically, inheritance is a way to reuse code. When class A inherits from, or *extends*, another class B, class A automatically inherits all of the `public` and `protected` members of class B. If class A is in the same package as class B, class A will also inherit all of the members with default, or *package*, access. It is important to note, however, that subclasses never inherit the private members of the classes they extend.

Once you extend a class, you can add new fields and methods, which define the attributes and operations that make your new class distinct from the superclass. Also, you can *override* the operations of the superclass that must behave differently in the subclass.

You can explicitly extend a class when you define it. To extend a class, you simply follow the name of the class with the `extends` keyword and the name of the class you want to extend. If you do not explicitly extend a class, the Java compiler will automatically extend the class `Object`. In this way, all Java objects are ultimately subclasses of class `Object`.

Extension example

Let's suppose we want to create a new `CheckingAccount` class. A `CheckingAccount` is a special kind of `BankAccount`. In other words, a `CheckingAccount` has the same attributes and operations as a `BankAccount`. However, a `CheckingAccount` also has an added operation; namely cashing a check. So we'll define our `CheckingAccount` class so that it extends `BankAccount` and add a `cashCheck()` method, as shown below.

```
class CheckingAccount extends BankAccount {
    public void cashCheck(float amount) {
        withdraw(amount);
    }
}
```

Subclass constructors

Constructors are not really members of a class, and constructors are not inherited. This makes sense if you think about it. A `BankAccount` constructor creates `CheckingAccount` objects, so it can't be used in the `CheckingAccount` class to create `CheckingAccount` objects.

You can, however, use constructors from a superclass to initialize the parts of a subclass that are inherited. In other words, you'll often need to call superclass constructors in subclass constructors to partially initialize your subclass objects. You can do this by using the `super` keyword, followed by a parameterized list representing the arguments of the super class constructor you want to call. If you're using the `super` keyword in a constructor, to call a superclass constructor, it must appear as the first statement in the constructor body.

For example, we'll need to write a `CheckingAccount` constructor to initialize `CheckingAccount` objects. We want to create `CheckingAccount` objects with an initial balance, so we'll pass in a dollar amount. This is just like a constructor in the `BankAccount` class, so we'll use that constructor to do all the work for us, as shown below.

```
class CheckingAccount extends BankAccount {

    public CheckingAccount(float balance) {
        super(balance);
    }

    public void cashCheck(float amount) {
        withdraw(amount);
    }
}
```

You can also use the `super` keyword to explicitly refer to superclass members from a subclass. We'll explore this use of the `super` keyword in the next section.

Overloading and overriding

Like C++, the Java language allows you to define several methods with the same name, as long as they take different parameters. For example, we can define a second `cashCheck()` method that takes the amount of the check to be cashed and a fee to be applied for the service. This is called method *overloading*.

```
public void cashCheck(float amount) {
    withdraw(amount);
}

public void cashCheck(float amount, float fee) {
    withdraw(amount+fee);
}
```

Often, when you create a subclass, you'll want to *override* the behavior of a method that is inherited from a superclass. For example, let's say that one difference between a `CheckingAccount` and a `BankAccount` is that there is a fee applied when you withdraw money from a `CheckingAccount`. We'll need to override the `withdraw()` method in the `CheckingAccount` class so that a $0.25 fee is applied. In fact, we'll define our `CheckingAccount` `withdraw()` method in terms of the `BankingAccount` `withdraw()` method, by using the `super` keyword, as shown below.

```
public void withdraw(float amount) {
    super.withdraw(amount+0.25F);
}
```

The Object class

The `Object` class is a special class in the Java class hierarchy. All Java classes are ultimately subclasses of class `Object`. In other words, the Java language supports a centrally rooted class hierarchy, and the `Object` class is the root class of that hierarchy. (C++ does not have a class hierarchy; this concept was borrowed from SmallTalk.)

Why is this important? Well, because all Java objects inherit from the `Object` class, you can call the methods defined in `Object` for any Java object, and expect similar behavior for each. For example, the `Object` class defines a `toString()` method, which returns a `String` object that represents the object. You can call the `toString()` method for any Java object and expect to get back a string representation of that object. Most class definitions will override the `toString()` method so that it returns a specialized string representation for that particular class.

The other implication of having `Object` at the root of the Java class hierarchy is that all objects can be cast down to `Object` objects. In C++, you can use templates to define data structures that take objects of different types. In the Java language, you can define data structures that take objects of class `Object`, and these data structures can hold any Java object.

Abstract classes

An *abstract class* is a class that cannot be instantiated. You may be wondering why anyone would ever want to create a class that you can't use to create objects. The answer is that you want other programmers to extend your class, but never create an instance of it. You can explicitly stop other programmers from instantiating your class by using the `abstract` modifier.

You can make any ordinary class an abstract class, just by adding the `abstract` modifier. However, usually an abstract class will have one or more *abstract* methods. As you might be able to guess, an abstract method is a method that is declared, but not implemented, much like a C function prototype. When a class extends an abstract class, it must provide an implementation for each abstract method, or else it must also be declared as an abstract class. You can define an abstract method by using the `abstract` modifier.

For example, let's define an `AbstractAccount` class, which represents a very basic bank account. The designer of this class does not want other programmers to use this class to create `AbstractAccount` objects. Instead, other programmers should use this class as a base class for creating concrete implementations of bank accounts. For example, you could create a `SavingsAccount` class and a `CheckingAccount` class, both of which subclass `BankAccount`. It doesn't make sense to create an instance of an `AbstractAccount` because an `AbstractAccount` doesn't really exist. An `AbstractAccount` is an abstract idea, but a `SavingsAccount` and a `CheckingAccount` are real kinds of bank accounts, and they do exist. After all, if you went into a bank and said, "I'd like to open a bank account," the first thing they'd ask is, "Do you want a savings account or a checking account?"

AbstractAccount.java

```
abstract class AbstractAccount {

    private float balance;
    private static float interest;

    public void setBalance(float balance) {
        this.balance = balance;
    }

    public void deposit(float amount) {
        balance += amount;
    }

    public abstract void withdraw(float amount);

    public static void setInterest(float interestRate) {
        interest = interestRate;
    }
}
```

Interfaces

We've already noted that a Java class can only extend one class. If you're coming from a background in C++, where classes can inherit from any number of classes, you may see this as extremely limiting, and you're right. The designers of the Java language felt that multiple inheritance was too complicated, so instead they introduced a new concept: *interfaces*.

An *interface* is like an abstract class, with the following exceptions:

- All interfaces are implicitly abstract, so you don't need to use the `abstract` keyword. Methods defined in an interface are also implicitly abstract, so you don't need to use the `abstract` keyword for methods. If you try to declare a method with a body in an interface, you'll get a compile-time error.
- All members of an interface are implicitly public.
- All fields defined in an interface are implicitly static and final.
- An interface cannot be instantiated, so an interface does not define a constructor.

An interface is declared just like a class, except the `interface` keyword is used instead of the `class` keyword. An interface can `extend` any number of superinterfaces. Methods inside an interface cannot include an implementation. Interface methods are simply method definitions; they do not have bodies.

Account.java

The code sample below shows how we might write a basic account interface, which defines a base set of functionality for bank accounts. Notice that there are no bodies for the methods declared in an interface.

```
interface Account {

    public static final float INTEREST = 0.35F;

    public void withdraw(float amount);

    public void deposit(float amount);
}
```

Implementing interfaces

A Java class can extend only one class, but it can *implement* any number of interfaces. When a class implements an interface, it must implement every method defined in that interface.

Let's define a `SavingsAccount` class that implements the `Account` interface. Because the `Account` interface defines two methods, `withdraw(float amount)` and `public void deposit(float amount)`, the `SavingsAccount` class must provide an implementation for them. The `SavingsAccount` class can still extend another class, and it can implement any other interfaces, as long as they don't define the same members as the `Account` interface.

```
class SavingsAccount implements Account {

    private float balance;

    public SavingsAccount(float balance) {
        this.balance = balance;
    }

    public void cashCheck(float amount, float fee) {
        withdraw(amount+fee);
    }

    public void withdraw(float amount) {
        balance += balance;
    }

    public void deposit(float amount) {
        balance -= balance;
    }
}
```

Inner classes and inner interfaces

As we mentioned earlier, a class can have fields, methods, classes, and interfaces as members. A class or interface that is defined inside another class or interface is called an *inner class* or *inner interface*. Inner classes and interfaces were added to the Java language in version 1.1.

Classes and interfaces that are not inner classes or inner interfaces are called *top-level classes* and *top-level interfaces*. We've only been dealing with top-level classes and interfaces up to this point. You can always use a top-level class and interfaces in place of nested classes and interfaces; they are simply added as a convenience feature.

You generally use an *inner class* or *inner interface* to represent a class or interface that you will only need to use inside the class or interface in which it is declared. Let's say you are defining a `LinkedList` class. You will probably use a `Node` class to represent the individual nodes that are chained together to create the linked list. Since this `Node` class will only be used internally in the `LinkedList` class, we can declare it as an inner class.

LinkedList.java

As you can see in the code listing for the `LinkedList` class, there is a special `Node` class that is defined as a `private` member of the `LinkedList` class. The `Node` class could have been written as a normal, top-level class, but because it is only used inside the `LinkedList` class, it is most appropriate to define it as a `private` member of that class, as shown below.

```
class LinkedList {

    private Node head;

    public LinkedList() { head=null; }

    public boolean isEmpty() { return (head==null); }

    public void add(int data) {
        Node node = new Node(data);
        node.next = head;
        head = node;
    }

    public void remove() {
        head = head.next;
    }

    /**
     *  Returns a String representation of the list.
     */
    public void display() {
        Node currentNode = head;
        while(currentNode != null) {
            System.out.println(currentNode.toString());
            currentNode = currentNode.next;
        }
    }

    /**
     *  Represents a node in a linked list.
     */
    private class Node {
        public int data;
        public Node next;

        public Node(int data) {
            this.data = data;
        }

        public String toString() {
            return "(" + data + ")";
        }
    }

    public static void main(String[] args)
    {
        LinkedList list = new LinkedList();
        list.add(8);
        list.add(6);
        list.add(5);
        list.add(3);
        list.remove();
        list.display();
    }
}
```

Section 8. Wrapup

Tutorial summary

In this tutorial we've gone over the most essential components of the Java language, using working examples and comparison to C/C++ to aid you in your learning. At this point, you should feel comfortable writing simple Java programs. Briefly, you should be able to do the following:

- Write a Java class with a `main()` method, compile it, and run it.
- Write a Java interface and compile it.
- Write one or more constructors for your class.

- Write a class that extends another class and implements one or more interfaces.
- Create and use objects using the `new` keyword and constructor calls.

You should also have enough of a grasp of the Java language to begin examining and playing around with more advanced Java code. A good place to begin would be with the Java platform classes themselves. The best way to gain experience using the language is to browse the API documentation and start writing programs that use these classes. You may also want to undertake one or more of the advanced exercises in the next section. See Resources for further references to any of the topics covered in this tutorial.

Advanced exercises

To increase your proficiency in Java programming, you may want to further practice implementing and extending the Java classes we've worked with here. Start with one of the exercises below:

- Write a Java interface called `Account`, which will define the basic behavior of a typical bank account. This interface should declare a `withdraw()` method and a `deposit()` method. What, if any, parameters should these methods take, and what should their types be? Look at the `Account` interface from the section on inheritance if you need help getting started.
- Write a Java class called `SavingsAccount` that implements the `Account` interface. Add a method called `getBalance()` that returns the current balance for the account. What type should this method return? Do you need to declare a field to hold the balance for this class?
- Write a Java class called `Check` that represents a typical check. You should write a method called `getAmount()` that returns the amount for the check that was written. What type should this method return? What, if any, fields should this class declare, and what should its type be?
- Write a Java class called `CheckingAccount` that extends the `SavingsAccount` class. Add a method called `cashCheck()` that deducts the amount of a specified `Check` object from the account. What, if any, parameters should this method take, and what should their types be?
- Write a Java class called `Fibonacci`, which takes a number argument and computes the Fibonacci number for that index in the sequence 0, 1,

1, 2, 3, 5, 8, 13. In other words, you start with 0 and 1, and every successive number is the sum of the previous two. If you call the program with the number 0, it should return 0, the number 6 should return 5, and the number 7 should return 13. Look at the `Factorial` program if you need help getting started.

Resources

Learn

- If you want to learn more about Unicode, visit the Unicode homepage.
- To learn more about the Java programming language, see Ken Arnold and James Gosling's *The Java Programming Language: Third Edition* (Addison-Wesley, 2000,).
- David Flanagan's *Java in a Nutshell, Third Edition* is also essential reading for the beginning Java programmer (O'Reilly, 1999).
- As you advance, you will find your bookshelf is incomplete without a well-thumbed copy of *Design Patterns: Elements of Reusable Object-Oriented Software* , by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional Computing Series, 1994).
- To learn more about design patterns, you may also want to check out Paul Monday's tutorial, "Java design patterns 101" (*developerWorks*, January 2002).
- Once you're comfortable with the basics of Java programming, you may want to check out my tutorial, "Java programming with JNI (*developerWorks*, March 2002)," which covers the two most common uses of the Java Native Interface: calling C/C++ code from Java programs, and calling Java code from C/C++ programs.
- Get another perspective on the Java Native Interface (and native compilation on the Java platform), in the article, "Weighing in on Java native compilation" (*developerWorks*, January 2002).
- Find out why WebSphere Studio Application Developer, IBM's integrated development environment for building Java enterprise applications, won the Java Pro 2002 Most Valuable Product and Best Java Deployment Tool awards.
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.
- See the *developerWorks* tutorials page for a complete listing of free Java technology tutorials from *developerWorks*.

Get products and technologies

- Download javac-cpp-source.zip, the source for this tutorial.
- You can download and install any Java SDK (and all related documentation) for free from the Java Developer Connection.
- You'll have no trouble finding a workable text editor on Tucows.com.

About the author

Scott Stricker
Scott Stricker is an enterprise application developer working in the Business Innovation Services group, part of IBM Global Services. He specializes in object-oriented technologies, particularly in Java and C++ programming. Scott has a Bachelor of Science degree in Computer Science from the University of Cincinnati. He is a Sun Certified Java 2 Programmer and Developer.