

Introduction to Object Oriented Programming:

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object oriented features were influenced by C++.

Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades.

The C language shook (past participle of shake) the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs. As you probably know, when a computer language is designed, trade-offs are often made, such as the following:

- ✓ Ease-of-use versus power
- ✓ Safety versus efficiency
- ✓ Rigidity versus extensibility

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn't very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult.

Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles. Instead, they relied upon the GOTO as a primary means of program control. As a result, programs written using these languages tended to produce "spaghetti code"—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand. While languages like Pascal are structured, they were not designed for efficiency, and failed to include certain features necessary to make them applicable to a wide range of programs.

So, just prior to the invention of C, no one language had reconciled the conflicting attributes that had dogged earlier efforts. Yet the need for such a language was pressing. By the early 1970s, the computer revolution was beginning to take hold, and the demand for software was rapidly outpacing programmers' ability to produce it.

A great deal of effort was being expended in academic circles in an attempt to create a better computer language. But, and perhaps most importantly, a secondary force was beginning to be felt. *Computer hardware was finally becoming common enough that a critical mass was being reached. No longer were computers kept behind locked doors.*

For the first time, programmers were gaining virtually unlimited access to their machines. This allowed the freedom to experiment. It also allowed programmers to begin to create their own tools. On the eve of C's creation, the stage was set for a quantum leap forward in computer languages.

Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an older language called **BCPL**, developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s.

For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (PrenticeHall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The first widespread language was, of course, FORTRAN. While FORTRAN was an impressive first step, it is hardly a language that encourages clear and easy-to-understand programs. The 1960s gave birth to structured programming. This is the method of programming championed by languages such as C.

The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage.

By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called object-oriented programming (OOP). *OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.*

In the final analysis, although C is one of the world's great programming languages, there is a limit to its ability to handle complexity. Once the size of a program exceeds a certain point, it becomes so complex that it is difficult to grasp as a totality. While the precise size at which this occurs differs, depending upon both the nature of the program and the programmer, there is always a threshold at which a program becomes unmanageable.

C++ added features that enabled this threshold to be broken, allowing programmers to comprehend and manage larger programs. C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language "C with Classes." However, in 1983, the name was changed to C++.

C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C's features, attributes, and benefits. This is a crucial reason for the success of C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

The Creation of Java:

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.

As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU.

The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web.

Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure (difficult to understand or unclear) language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront (front position) of computer language design, because the Web, too, demanded portable programs.

Most programmers learn early in their careers that portable programs are as elusive (hard to define) as they are desirable. While the quest for a way to create efficient, portable (platform-independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems.

Further, because (at that time) much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified (prepared) boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance (revenge).

After all, the Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs. Even though many kinds of platforms are attached to the Internet, users would like them all to be able to run the same program. What was once an irritating but low priority problem had become a high-profile necessity.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale.

This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers.

In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it.

Thus, Java is a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++.

For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

How Java Changed the Internet:

The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content.

Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let's look more closely at each of these.

Java Applets:

An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser.

Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs.

For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm.

It must also be able to run in a variety of different environments and under different operating systems. As you will see, Java solved these problems in an effective and elegant way. Let's look a bit more closely at each.

Security:

As you are likely aware, every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources.

For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached may have been the single most innovative aspect of Java.

Portability:

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.

For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed.

Java's Magic: The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode.

Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*. In essence, the original JVM was designed as an *interpreter for bytecode*.

This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns.

However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs. Here is why.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.

Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.

If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code.

However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java's initial release.

HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis.

It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time.

Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost.

Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

Java Buzzwords:

No discussion of Java's history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

1.	Simple	Java has a concise, cohesive set of features that makes it easy to learn and use.
2.	Secure	Java provides a secure means of creating Internet applications.
3.	Portable	Java programs can execute in any environment for which there is a Java run-time system.
4.	Object-oriented	Java embodies the modern, object-oriented programming philosophy.
5.	Robust	Java encourages error-free programming by being strictly typed and performing run-time checks.
6.	Multithreaded	Java provides integrated support for multithreaded programming.
7.	Architecture-neutral	Java is not tied to a specific machine or operating system architecture.
8.	Interpreted	Java supports cross-platform code through the use of Java bytecode.
9.	High performance	The Java bytecode is highly optimized for speed of execution.
10.	Distributed	Java was designed with the distributed environment of the Internet in mind.
11.	Dynamic	Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.

Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies:

Simple:

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort.

Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented:

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate.

One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model.

The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

Robust:

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development.

At the same time, Java frees you from having to worry about many of the most common causes of programming errors. *Because Java is a strictly typed language*, it checks your code at compile time. However, it also checks your code at run time.

Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments.

For example, in C/C++, the programmer will often manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using.

Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.)

Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs.

Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded:

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral:

A central issue for the Java designers was that of code longevity and portability. At the time of Java’s creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine.

Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, anytime, forever.” To a great extent, this goal was accomplished.

Interpreted and High Performance:

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance.

As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed:

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic:

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java / History of Java:

The initial release of Java (1996) was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1.

The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." The first release of Java 2 carried the version number 1.2.

It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the **Thread** class in which the methods **suspend()**, **resume()**, and **stop()** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword **assert**, chained exceptions, and a channel-based I/O subsystem.

It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language.

To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

1. Generics
2. Annotations
3. Autoboxing and auto-unboxing
4. Enumerations
5. Enhanced, for-each style **for** loop
6. Variable-length arguments (varargs)
7. Static import
8. Formatted I/O
9. Concurrency utilities

This is not a list of minor tweaks or incremental upgrades. Each item in the list represented a significant addition to the Java language. Some, such as generics, the enhanced **for**, and varargs, introduced new syntax elements. Others, such as autoboxing and auto-unboxing, altered the semantics of the language.

Annotations added an entirely new dimension to programming. In all cases, the impact of these additions went beyond their direct effects. They changed the very character of Java itself. The importance of these new features is reflected in the use of the version number "5."

The next version number for Java would normally have been 1.5. However, the new features were so significant that a shift from 1.4 to 1.5 just didn't seem to express the magnitude of the change. Instead, Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the developer's kit was called JDK 5.

However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number, which is also referred to as the *developer version* number. The "5" in J2SE 5 is called the *product version* number.

The next release of Java was called Java SE 6. Sun once again decided to change the name of the Java platform. First, notice that the "2" was dropped. Thus, the platform was now named *Java SE*, and the official product name was *Java Platform, Standard Edition 6*. The Java Development Kit was called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.

Java SE 6 built on the base of J2SE 5, adding incremental improvements. Java SE 6 added no major features to the Java language proper, but it did enhance the API libraries, added several new packages, and offered improvements to the runtime. It also went through several updates during its (in Java terms) long life cycle, with several upgrades added along the way.

In general, Java SE 6 served to further solidify the advances made by J2SE 5. Java SE 7 was the next release of Java, with the Java Development Kit being called JDK 7, and an internal version number of 1.7. Java SE 7 was the first major release of Java since Sun Microsystems was acquired by Oracle.

Java SE 7 contained many new features, including significant additions to the language and the API libraries. Upgrades to the Java run-time system that support non-Java languages were also included, but it is the language and library additions that were of most interest to Java programmers. The new language features were developed as part of *Project Coin*.

The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7. Although these features were collectively referred to as “small,” the effects of these changes have been quite large in terms of the code they impact. In fact, for many programmers, these changes may well have been the most important new features in

Java SE 7. Here is a list of the language features added by JDK 7:

- A **String** can now control a **switch** statement.
- Binary integer literals.
- Underscores in numeric literals.
- An expanded **try** statement, called *try-with-resources*, that supports automatic resource management. (For example, streams can be closed automatically when they are no longer needed.)
- Type inference (via the *diamond* operator) when constructing a generic instance.
- Enhanced exception handling in which two or more exceptions can be caught by a single **catch** (multi-catch) and better type checking for exceptions that are rethrown.
- Although not a syntax change, the compiler warnings associated with some types of varargs methods were improved, and you have more control over the warnings.

As you can see, even though the Project Coin features were considered small changes to the language, their benefits were much larger than the qualifier “small” would suggest.

In particular, the **try-with-resources** statement has profoundly affected the way that stream-based code is written. Also, the ability to use a **String** to control a **switch** statement was a long desired improvement that simplified coding in many situations.

Java SE 7 made several additions to the Java API library. Two of the most important were the enhancements to the NIO Framework and the addition of the Fork/Join Framework.

NIO (which originally stood for New I/O) was added to Java in version 1.4. However, the changes added by Java SE 7 fundamentally expanded its capabilities. So significant were the changes, that the term NIO.2 is often used.

The Fork/Join Framework provides important support for parallel programming. Parallel programming is the name commonly given to the techniques that make effective use of computers that contain more than one processor, including multicore systems.

The advantage that multicore environments offer is the prospect of significantly increased program performance. The Fork/Join Framework addressed parallel programming by

- Simplifying the creation and use of tasks that can execute concurrently
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can easily create scaleable applications that automatically take advantage of the processors available in the execution environment. Of course, not all algorithms lend themselves to parallelization, but for those that do, a significant improvement in execution speed can be obtained.

Java SE 8

The newest release of Java is Java SE 8, with the developer's kit being called JDK 8. It has an internal version number of 1.8. JDK 8 represents a very significant upgrade to the Java language because of the inclusion of a far-reaching new language feature: the *lambda expression*. The impact of lambda expressions will be profound, changing both the way that programming solutions are conceptualized and how Java code is written.

Lambda expressions add functional programming features to Java. In the process, lambda expressions can simplify and reduce the amount of source code needed to create certain constructs, such as some types of anonymous classes. The addition of lambda expressions also causes a new operator (the `->`) and a new syntax element to be added to the language. Lambda expressions help ensure that Java will remain the vibrant, nimble language that users have come to expect.

The inclusion of lambda expressions has also had a wide-ranging effect on the Java libraries, with new features being added to take advantage of them. One of the most important is the new stream API, which is packaged in *java.util.stream*. The stream API supports pipeline operations on data and is optimized for lambda expressions.

Another very important new package is *java.util.function*. It defines a number of functional interfaces, which provide additional support for lambda expressions. Other new lambda-related features are found throughout the API library.

Another lambda-inspired feature affects interface. Beginning with JDK 8, it is now possible to define a default implementation for a method specified by an interface. If no implementation for a default method is created, then the default defined by the interface is used.

This feature enables interfaces to be gracefully evolved over time because a new method can be added to an interface without breaking existing code. It can also streamline the implementation of an interface when the defaults are appropriate. Other new features in

JDK 8 include a new time and date API, type annotations, and the ability to use parallel processing when sorting an array, among others. JDK 8 also bundles support for JavaFX 8, the latest version of Java's new GUI application framework. JavaFX is expected to soon play an important part in nearly all Java applications, ultimately replacing Swing for most GUI-based projects.

Java Environment:

Although the Java language is highly standardized, the procedures for creating, compiling, and editing Java programs vary widely from one programming environment to another.

There are two basic approaches: a command line environment , where the user types commands and the computer responds, and an integrated development environment (IDE), where the user uses the keyboard and mouse to interact with a graphical user interface.

While there is just one common command line environment for Java programming, there is a wide variety of IDEs.

Java Development Kit:

The basic development system for Java programming is usually referred to as the JDK (Java Development Kit). It is a part of J2SE, the Java 2 Platform Standard Edition. *Note* that J2SE comes in *two versions*, **a Development Kit version** and **a Runtime version**.

The Runtime can be used to run Java programs and to view Java applets in Web pages, but it does not allow you to compile your own Java programs. The Development Kit includes the Runtime and adds to it the JDK which lets you compile programs.

Java was developed by Sun Microsystems, Inc., which makes it's JDK for Windows and Linux available for free download at its Java Web site, java.sun.com. If you have a Windows computer, it might have come with a Java Runtime, but you might still need to download the JDK.

If a JDK is installed on your computer, you can use the command line environment to compile and run Java programs. Some IDEs depend on the JDK, so even if you plan to use an IDE for programming, you might still need a JDK.

Command Line Environment:

Many modern computer users find the command line environment to be pretty alien (unfamiliar) and unintuitive (unaware). It is certainly very different from the graphical user interfaces that most people are used to.

However, it takes only a little practice to learn the basics of the command line environment and to become productive using it. To use a command line programming environment, you will have to open a window where you can type in commands.

In Windows, you can open such a command window by running the program named `cmd`. One way to run `cmd` is to use the "Run Program" feature in the Start menu, and enter "cmd" as the name of the program.

You should create a directory (that is, a folder) to hold your Java work. For example, create a directory named *javawork* in your home directory. You can do this using your computer's GUI; another way to do it is to open a command window and enter the command *mkdir javawork*.

When you want to work on programming, open a command window and enter the command *cd javawork* to change into your work directory. Of course, you can have more than one working directory for your Java work; you can organize your files any way you like.

The most basic commands for using Java on the command line are *javac* and *java*; *javac* is used to compile Java source code, and *java* is used to run Java stand-alone applications. If a JDK is correctly installed on your computer, it should recognize these commands when you type them in on the command line.

Try typing the commands *java -version* and *javac -version* which should tell you which version of Java is installed. If you get a message such as "Command not found," then Java is not correctly installed. If the "java" command works, but "javac" does not, it means that a Java Runtime is installed rather than a Development Kit.

To create your own programs, you will need a text editor. A text editor is a computer program that allows you to create and save documents that contain plain text. It is important that the documents be saved as plain text, which is without any special encoding or formatting information.

Word processor documents are not appropriate, unless you can get your word processor to save as plain text. A good text editor can make programming a lot more pleasant. On Windows, you can use notepad in a pinch, but you will probably want something better.

One possibility that will work on any platform is to use *jedit*, a good programmer's text editor that is itself written in Java and that can be downloaded for free from www.jedit.org.

To create your own programs, you should open a command line window and `cd` into the working directory where you will store your source code files. Start up your text editor program, such as by double-clicking its icon or selecting it from a Start menu. Type your code into the editor window, or open an existing source code file that you want to modify. Save the file.

Remember that the name of a Java source code file must end in `".java"`, and the rest of the file name must match the name of the class that is defined in the file. Once the file is saved in your working directory, go to the command window and use the `javac` command to compile it, as discussed above.

If there are syntax errors in the code, they will be listed in the command window. Each error message contains the line number in the file where the computer found the error. Go back to the editor and try to fix the errors, save your changes, and they try the `javac` command again. (It's usually a good idea to just work on the first few errors; sometimes fixing those will make other errors go away.)

Remember that when the `javac` command finally succeeds, you will get no message at all. Then you can use the `java` command to run your program, as described above. Once you've compiled the program, you can run it as many times as you like without recompiling it.

IDEs and Eclipse:

In an **I**ntegrated **D**evelopment **E**nvironment, everything you need to create, compile, and run programs is integrated into a single package, with a graphical user interface that will be familiar to most computer users.

There are many different IDEs for Java program development, ranging from fairly simple wrappers around the JDK to highly complex applications with a multitude of features. For a beginning programmer, there is a danger in using an IDE, since the difficulty of learning to use the IDE, on top of the difficulty of learning to program, can be overwhelming.

Eclipse has a variety of features that are very useful for a beginning programmer. And even though it has many advanced features, its design makes it possible to use Eclipse without understanding its full complexity.

Eclipse is used by many professional programmers and is probably the most commonly used Java IDE. (In fact, Eclipse is actually a general development platform that can be used for other purposes besides Java development, but its most common use is Java.)

Eclipse is itself written in Java. It requires Java 1.4 (or higher) to run, so it works on any computer platform that supports Java 1.4, including Linux, Windows, and recent versions of Mac OS.

If you want to use Eclipse to compile and run Java 5.0 programs, you need Eclipse version 3.1 (or higher). Furthermore, Eclipse requires a JDK. You should make sure that JDK 5.0 (or higher) is installed on your computer, as described above, before you install Eclipse. Eclipse can be downloaded for free from www.eclipse.org.

The first time you start Eclipse, you will be asked to specify a workspace, which is the directory where all your work will be stored. You can accept the default name, or provide one of your own.

When startup is complete, the Eclipse window will be filled by a large "Welcome" screen that includes links to extensive documentation and tutorials. You can close this screen, by clicking the "X" next to the word "Welcome"; you can get back to it later by choosing "Welcome" from the "Help" menu.

The Eclipse GUI consists of one large window that is divided into several sections. Each section contains one or more views. If there are several views in one section, there will be tabs at the top of the section to select the view that is displayed in that section. Each view displays a different type of information. The whole set of views is called a perspective.

Eclipse uses different perspectives, which are different sets of views of different types of information, for different tasks. The only perspective that you will need is the "Java Perspective." Select the "Java Perspective" from the "Open Perspective" submenu of the "Window" menu. (You will only have to do this once, the first time you start Eclipse.)

The Java perspective includes a large area in the center of the window where you will create and edit your Java programs. To the left of this is the Package Explorer view, which will contain a list of your Java projects and source code files. To the right is an "Outline" view which shows an outline of the file that you are currently editing.

Several other views that will be useful while you are compiling and running programs appear in a section of the window below the editing area. If you accidentally close one of the important views, such as the Package Explorer, you can get it back by selecting it from the "Show View" submenu of the "Window" menu.

To do any work in Eclipse, you need a project. To start a Java project, go to the "New" submenu in the "File" menu, and select the "Project" command. In the window that pops up, make sure "Java Project" is selected, and click the "Next" button.

In the next window, it should only be necessary to fill in a "Project Name" for the project and click the "Finish" button. The project should appear in the "Package Explorer" view. Click on the small triangle next to the project name to see the contents of the project. At the beginning, it contains only the "JRE System Library"; this is the collection of standard built-in classes that come with Java.

To run the program, right-click the file name in the Package Explorer view (or control-click in Mac OS). In the menu that pops up, go to the "Run As" submenu, and select "Java Application". The program will be executed.

Eclipse has several features that aid you as you type your code. It will underline any syntax error with a jagged red line, and in some cases will place an error marker in the left border of the edit window. If you hover the mouse cursor over the error marker, a description of the error will appear.

Note that you do not have to get rid of every error immediately as you type; some errors will go away as you type in more of the program. If an error marker displays a small "light bulb," Eclipse is offering to try to fix the error for you. Click the light bulb to get a list of possible fixes, then double click the fix that you want to apply.

For example, if you use an undeclared variable in your program, Eclipse will offer to declare it for you. You can actually use this error-correcting feature to get Eclipse to write certain types of code for you!

Unfortunately, you'll find that you won't understand a lot of the proposed fixes until you learn more about the Java language. Another nice Eclipse feature is code assist. Code assist can be invoked by typing Control-Space. It will offer possible completions of whatever you are typing at the moment.

For example, if you type part of an identifier and hit Control-Space, you will get a list of identifiers that start with the characters that you have typed; use the up and down arrow keys to select one of the items in the list, and press Return or Enter. (Or hit Escape to dismiss the list.)

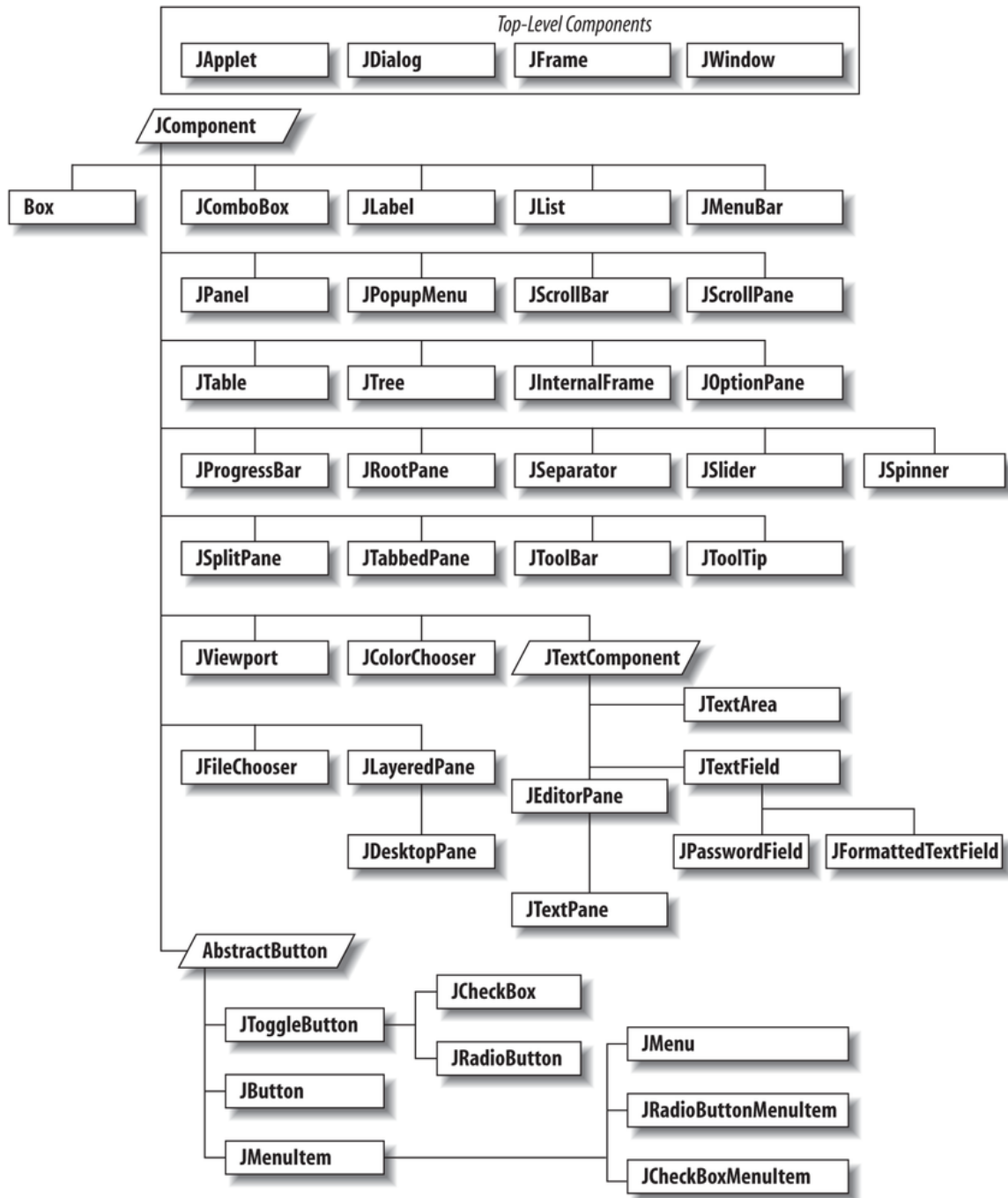
If there is only one possible completion when you hit Control-Space, it will be inserted automatically. By default, Code Assist will also pop up automatically, after a short delay, when you type a period or certain other characters.

Once you have an error-free program, you can *run it as described above*, by right-clicking its name in the Package Explorer and using "Run As / Java Application".

Java Components:

A component is the fundamental user interface object in Java. Everything you see on the display in a Java application is a component. This includes things like windows, panels, buttons, checkboxes, scrollbars, lists, menus, and text fields. To be used, a component usually must be placed in a container.

Container objects group components, arrange them for display using a layout manager, and associate them with a particular display device. All Swing components are derived from the abstract *javax.swing.JComponent* class as you will see in the Figure as follows. For example, the *JButton* class is a subclass of *AbstractButton*, which is itself a subclass of the *JComponent* class.



JComponent is the root of the Swing component hierarchy, but it descends from the AWT *Container* class. At this bottom level, Swing is based on AWT, so our conversation occasionally delves into the AWT package.

Container's superclass is *Component*, the root of all AWT components, and *Component's* superclass is, finally, *Object*. Because *JComponent* inherits from *Container*, it has the capabilities of both a component and a container.

AWT and Swing, then, have parallel hierarchies. The root of AWT's hierarchy is *Component*, while Swing's components are based on *JComponent*. You'll find similar classes in both hierarchies, such as *Button* and *JButton*, *List*, and *JList*. But Swing is much more than a replacement for AWT—it contains sophisticated components as well as a real implementation of the Model-View-Controller (MVC) paradigm.

For the sake of simplicity, we can split the functionality of the *JComponent* class into two categories: appearance and behavior. The *JComponent* class contains methods and variables that control an object's general appearance.

This includes basic attributes, such as its visibility, its current size and location, and certain common graphical defaults, such as font and background color, used by different subclasses in different ways. The *JComponent* class also contains graphics and event-handling methods, which are overridden by subclasses to produce all of the different kinds of widgets that we will see.

When a component is first displayed, it's associated with a particular display device. The *JComponent* class encapsulates access to its display area on that device. It includes tools for rendering graphics, for working with off-screen resources, and for receiving user input.

Under the covers, *JComponent* makes heavy use of the Java 2D API to handle things like font smoothing, rendering optimizations, and rendering hints. With recent versions of Java (6 and later), rendering speed and quality are often indistinguishable from native applications on popular operating systems.

When we talk about a component's behavior, we mean the way it responds to user-driven events. When the user performs an action (such as pressing the mouse button) within a component's display area, a Swing thread delivers an event object that describes what happened.

The event is delivered to objects that have registered themselves as listeners for that type of event from that component. For example, when the user clicks on a button, the button generates an *ActionEvent* object. To receive those events, an object registers with the button as an *ActionListener*.

Events are delivered by invoking designated event handler methods within the receiving object (the "listener"). A listener object receives specific types of events through methods of its listener interfaces (for example, through the *actionPerformed()* method of the *ActionListener* interface) for the types of events in which it is interested.

Specific types of events cover different categories of component user interaction. For example, *MouseEvent*s describe activities of the mouse within a component's area, *KeyEvent*s describe keypresses, and higher-level events (such as *ActionEvents*) indicate that a user interface component has done its job.

Programming Paradigms:

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms.

Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar.

Common programming paradigms include:

- **imperative** in which the programmer instructs the machine how to change its state,
 - *procedural* which groups instructions into procedures,
 - *object-oriented* which groups instructions together with the part of the state they operate on
- **declarative** in which the programmer merely declares properties of the desired result, but not how to compute it
 - *functional* in which the desired result is declared as the value of a series of function applications,
 - *logic* in which the desired result is declared as the answer to a question about a system of facts and rules,
 - *mathematical* in which the desired result is declared as the solution of an optimization problem

Symbolic techniques such as reflection, which allow the program to refer to itself, might also be considered as a programming paradigm. However, this is compatible with the major paradigms and thus is not a real paradigm in its own right.

For example, languages that fall into the imperative paradigm have two main features: they state the order in which operations occur, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code. The communication between the units of code is not explicit.

Meanwhile, in object-oriented programming, code is organized into objects that contain state that is only modified by the code that is part of the object. Most object-oriented languages are also imperative languages. In contrast, languages that fit the declarative paradigm do not state the order in which to execute operations. Instead, they supply a number of operations that are available in the system, along with the conditions under which each is allowed to execute.

In general^{textbook data} there are two programming paradigms:

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around “what is happening” and others are written around “who is being affected.”

These are the two paradigms that govern how a program is constructed. The first way is called the *process-oriented model*. This approach characterizes a program as a series of linear steps (that is, code).

The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success. However, problems with this approach appear as programs grow larger and more complex.

To manage increasing complexity, the second approach, called *object-oriented programming*, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code*.

Naming Conventions:

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), \$ (dollar), _ (underscore).

Let's see some other rules that should be followed by identifiers.

Class

- It should start with the uppercase letter.
- It should be a noun such as Color, Button, System, Thread, etc.
- Use appropriate words, instead of acronyms.

Example: -

```
public class Employee
{
//code snippet
}
```

Interface

- It should start with the uppercase letter.
- It should be an adjective such as Runnable, Remote, ActionListener.
- Use appropriate words, instead of acronyms.

Example: -

```
interface Printable
{
//code snippet
}
```

Method

- It should start with lowercase letter.
- It should be a verb such as main(), print(), println().
- If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

Example:-

```
class Employee
{
//method
void draw()
{
//code snippet
}
```



```
}  
}
```

Variable

- It should start with a lowercase letter such as id, name.
- It should not start with the special characters like & (ampersand), \$ (dollar), _ (underscore).
- If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
- Avoid using one-character variables such as x, y, z.

Example :-

```
class Employee  
{  
  //variable  
  int id;  
  //code snippet  
}
```

Package

- It should be a lowercase letter such as java, lang.
- If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

Example :-

```
package com.javapack; //package  
class Employee  
{  
  //code snippet  
}
```

Constant

- It should be in uppercase letters such as RED, YELLOW.
- If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
- It may contain digits but not as the first letter.

Example :-

```
class Employee
{
//constant
    static final int MIN_AGE = 18;
//code snippet
}
```

CamelCase in java naming conventions:

Java follows camel-case syntax for naming the class, interface, method, and variable. If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Classes and Objects – Introduction to classes:

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

The most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class:

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
```

```
}  
type methodname2(parameter-list) {  
// body of method  
}  
// ...  
type methodnameN(parameter-list) {  
// body of method  
}  
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

A Simple Class:

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {  
double width;  
double height;  
double depth;  
}
```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will be an instance of **Box**. As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**.

To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object. One other point: Although commonly referred to as the dot *operator*, the formal specification for Java categorizes the . as a separator.

Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}
// This class declares an object of type Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main()** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**.

The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.
class Box {
    double width;
    double height;
    double depth;
}
class BoxDemo2 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        /* assign different values to mybox2's
        instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

Declaring Objects:

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process.

First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator.

The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object. The effect of these two lines of code is depicted in figure below:

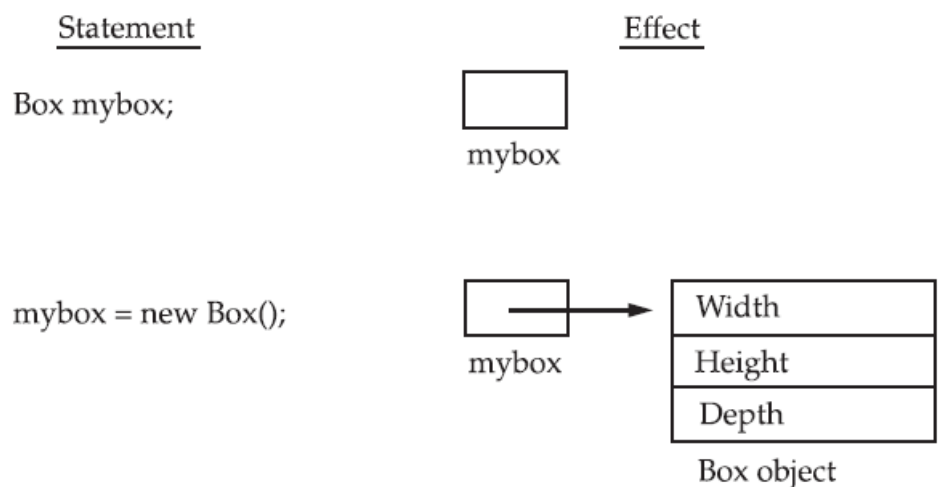


Figure: Declaring an object of type **Box**

A Closer Look at new:

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

```
class-var = new classname ( );
```

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created.

Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor.

At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur.

Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

Assigning Object Reference Variables:

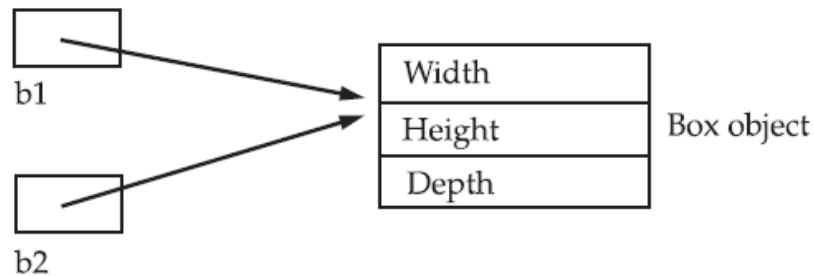
Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();  
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object.

The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```

Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
  
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

REMEMBER When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method:

```

type name(parameter-list) {
  // body of method
}
  
```

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.

The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called.

If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

```
return value;
```

Here, *value* is the value returned.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to the Box Class:

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.
```

```
class Box {
    double width;
    double height;
    double depth;
    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}
class BoxDemo3 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        // assign values to mybox1's instance variables
        mybox1.width = 10;
```

```
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

The first line here invokes the **volume()** method on **mybox1**. That is, it calls **volume()** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume()** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume()** displays the volume of the box defined by **mybox2**. Each time **volume()** is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume()** is executed, the Java run-time system transfers control to the code defined inside **volume()**. After the statements inside **volume()** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines.

There is something very important to notice inside the **volume()** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known.

Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume()** implicitly refer to the copies of those variables found in the object that invokes **volume()**.

Let's review: *When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.*

Returning a Value

While the implementation of **volume()** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement **volume()** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.
class Box {
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        /* assign different values to mybox2's
        instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
    }
}
```

```
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, when **volume()** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume()**. Thus, after

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume()** is 3,000 and this value then is stored in **vol**. There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume()** could have been used in the **println()** statement directly, as shown here:

```
System.out.println("Volume is" + mybox1.volume());
```

In this case, when **println()** is executed, **mybox1.volume()** will be called automatically and its value will be passed to **println()**.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square()** much more useful.

```
int square(int i)
{
return i * i;
}
```

```
}
```

Now, **square()** will return the square of whatever value it is called with. That is, **square()** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;  
x = square(5); // x equals 25  
x = square(9); // x equals 81  
  
y = 2;  
  
x = square(y); // x equals 4
```

In the first call to **square()**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example.

As these examples show, **square()** is able to return the square of whatever data it is passed. It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called.

For example, in **square()**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square()**, the parameter **i** receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```
mybox1.width = 10;  
mybox1.height = 20;  
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized method.  
class Box {
```

```
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}

class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

As you can see, the **setDim()** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim()** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

Constructors:

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim()**, it would be simpler and more concise to have all of the setup done at the time the object is first created.

Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the **new** operator completes.

Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim()** with a constructor. Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the dimensions of a box. */
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
```

```
vol = mybox2.volume();  
System.out.println("Volume is " + vol);  
}  
}
```

When this program is run, it generates the following results:

```
Constructing Box  
Constructing Box  
Volume is 1000.0  
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box()** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println()** statement inside **Box()** is for the sake of illustration only. Most constructors will not display anything.

They will simply initialize an object. Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname ( );
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

new Box() is calling the **Box()** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor.

The default constructor automatically initializes all instance variables to their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors:

While the **Box()** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor.

As you can probably guess, this makes it much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.


```
/* Here, Box uses a parameterized constructor to initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}

class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box()** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

Garbage Collection:

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.

Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The this Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object.

That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted. To better understand what **this** refers to, consider the following version of **Box()**:

```
// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

This version of **Box()** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box()**, **this** will always refer to the invoking object.

Instance Variable Hiding:

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.

However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box()** constructor inside the **Box** class.

If they had been, then **width**, for example, would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables.

For example, here is another version of **Box()**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

The finalize() Method: (Topic Beyond Syllabus)

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.

To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize()** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method, you will specify those actions that must be performed before an object is destroyed.

The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize()** method on the object.

The **finalize()** method has this general form:

```
protected void finalize( )
{
    // finalization code here
}
```

```
}
```

Here, the keyword **protected** is a specifier that limits access to **finalize()**.

It is important to understand that **finalize()** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize()** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

Access Control:

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering.

How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.)

Here, let’s begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

Java’s access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access modifiers are described next.

Let’s begin by defining **public** and **private**. When a member of a class is modified by **public**, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.

Now you can understand why **main()** has always been preceded by the **public** modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

In the classes developed so far, all members of a class have used the default access mode. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods.

Also, there will be times when you will want to define methods that are private to a class. An access modifier precedes the rest of a member’s type specification. That is, it must begin a member’s declaration statement. Here is an example:

```
public int i;
private double j;
private int myMethod(int a, char b) { //...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between public and private. */
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}
}

class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly.

It must be accessed through its public methods: **setc()** and **getc()**. If you were to remove the comment symbol from the beginning of the following line, `// ob.c = 100; // Error!` then you would not be able to compile this program because of the access violation.

Note: For more info. Refer Pg.No. 143 Java The Complete Reference 9th edition

Understanding static: (Topic Beyond Syllabus)

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance.

To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:

- They can only directly call other **static** methods.
- They can only directly access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a*4** or **12**. Then **main()** is called, which calls **meth()**, passing **42** to **x**. The three **println()** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

```
Static block initialized.  
x = 42  
a = 3  
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

```
classname.method( )
```

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object reference variables.

A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables. Here is an example. Inside **main()**, the **static** method **callme()** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}  
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```

Here is the output of this program:

```
a = 42  
b = 99
```

Features of Object Oriented Programming:

There are **three** main features of OOPS.

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**
4. Abstraction
5. Class
6. Object
7. Method
8. Message Passing

1. Encapsulation

Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a “black box,” or a device that you can use without regard to the internal mechanisms.

A programmer can access and use the methods and data contained in the black box but cannot change them. Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments. Properties can be accessed using getMethodName() methods which are having public access modifiers.

```
package oopsconcept;
public class Mobile {
    private String manufacturer;
    private String operating_system;
    public String model;
    private int cost;
    //Constructor to set properties/characteristics of object
    Mobile(String man, String o, String m, int c){
        this.manufacturer = man;
        this.operating_system=o;
        this.model=m;
        this.cost=c;
    }
    //Method to get access Model property of Object
    public String getModel(){
        return this.model;
    }
    // We can add other method to get access to other properties
}
```

2. Inheritance

An important feature of object-oriented programs is inheritance—the ability to create classes that shares the attributes and methods of existing classes, but with more specific features.

Inheritance is mainly used for code reusability. So you are making use of already written the classes and further extending on that. That why we discussed the code reusability the concept. In general one line definition, we can tell that deriving a new class from existing class, it's called as Inheritance. You can look into the following example for inheritance concept. Here we have Mobile class extended by other specific class like Android and Blackberry.

```
package oopsconcept;
public class Android extends Mobile{
    //Constructor to set properties/characteristics of object
    Android(String man, String o,String m, int c){
        super(man, o, m, c);
    }
    //Method to get access Model property of Object
    public String getModel(){
        return "This is Android Mobile- " + model;
    }
}
```

```
package oopsconcept;
public class Blackberry extends Mobile{
    //Constructor to set properties/characteristics of object
    Blackberry(String man, String o,String m, int c){
        super(man, o, m, c);
    }
    public String getModel(){
        return "This is Blackberry-"+ model;
    }
}
```

3. Polymorphism

In Core, Java Polymorphism is one of easy concept to understand. Polymorphism definition is that *Poly* means many and *morphos* means forms. It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. There are two types of Polymorphism available in Java.

For example, in English, the verb "run" means different things – if you use it with "a footrace," a "business," or "a computer." You understand the meaning of "run" based on the other words used with it. Object-oriented programs are written so that the methods having the same name works differently in different context. Java provides two ways to implement polymorphism.

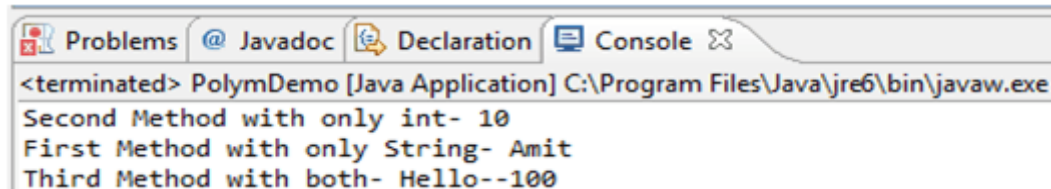
Static Polymorphism (compile time polymorphism/ Method overloading):

The ability to execute different method implementations by altering the argument used with the method name is known as method overloading.

In below program, we have three print methods each with different arguments. When you properly overload a method, you can call it providing different argument lists, and the appropriate version of the method executes.

```
package oopsconcept;
class Overloadsample {
    public void print(String s){
        System.out.println("First Method with only String- "+ s);
    }
    public void print (int i){
        System.out.println("Second Method with only int- "+ i);
    }
    public void print (String s, int i){
        System.out.println("Third Method with both- "+ s + "--" + i);
    }
}
public class PolymDemo {
    public static void main(String[] args) {
        Overloadsample obj = new Overloadsample();
        obj.print(10);      obj.print("Amit");  obj.print("Hello", 100);
    }
}
```

Output:



```
<terminated> PolymDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Second Method with only int- 10
First Method with only String- Amit
Third Method with both- Hello--100
```

Dynamic Polymorphism (run time polymorphism/ Method Overriding):

When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent.

Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members. Let's take the example used in inheritance explanation.

```
package oopsconcept;
public class OverridingDemo {
    public static void main(String[] args) {
        //Creating Object of SuperClass and calling getModel Method
        Mobile m = new Mobile("Nokia", "Win8", "Lumia",10000);
        System.out.println(m.getModel());
        //Creating Object of Sublcass and calling getModel Method
```

```
        Android a = new Android("Samsung", "Android", "Grand",30000);
        System.out.println(a.getModel());
        //Creating Object of Sublcass and calling getModel Method
        Blackberry b = new Blackberry("BlackB", "RIM", "Curve",20000);
        System.out.println(b.getModel());    }
    }
```

4. Abstraction:

All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. An essential element of object-oriented programming is an abstraction. Humans manage complexity through abstraction.

When you drive your car you do not have to be concerned with the exact internal working of your car (unless you are a mechanic). What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc.

Various manufacturers of car have different implementation of the car working but its basic interface has not changed (i.e. you still use the steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces.

From the outside, a car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.

For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

An abstract class is something which is incomplete and you cannot create an instance of the abstract class. If you want to use it you need to make it complete or concrete by extending it.

A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended. By the way, Java has a concept of abstract classes, abstract method but a variable cannot be abstract in Java.

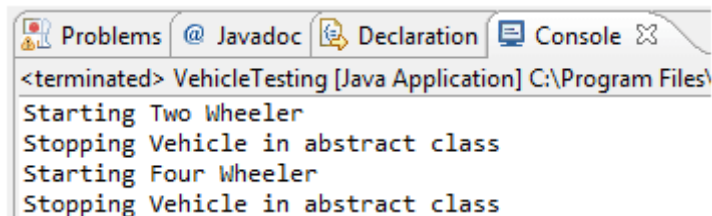
Let's take an example of Java Abstract Class called Vehicle. When I am creating a class called Vehicle, I know there should be methods like start() and Stop() but don't know start and stop mechanism of every vehicle since they could have different start and stop mechanism e.g. some can be started by a kick or some can be by pressing buttons.

The advantage of Abstraction is if there is a new type of vehicle introduced we might just need to add one class which extends Vehicle Abstract class and implement specific methods. The interface of start and stop method would be same.

```
package oopsconcept;
public abstract class VehicleAbstract {
    public abstract void start();
    public void stop(){
        System.out.println("Stopping Vehicle in abstract class");
    }
}
class TwoWheeler extends VehicleAbstract{
    @Override
    public void start() {
        System.out.println("Starting Two Wheeler");
    }
}
class FourWheeler extends VehicleAbstract{
    @Override
    public void start() {
        System.out.println("Starting Four Wheeler");
    }
}

package oopsconcept;
public class VehicleTesting {
    public static void main(String[] args) {
        VehicleAbstract my2Wheeler = new TwoWheeler();
        VehicleAbstract my4Wheeler = new FourWheeler();
        my2Wheeler.start();      my2Wheeler.stop();
        my4Wheeler.start();      my4Wheeler.stop();
    }
}
```

Output:



```
<terminated> VehicleTesting [Java Application] C:\Program Files\
Starting Two Wheeler
Stopping Vehicle in abstract class
Starting Four Wheeler
Stopping Vehicle in abstract class
```

5. Class:

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers:** A class can be public or has default access.
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

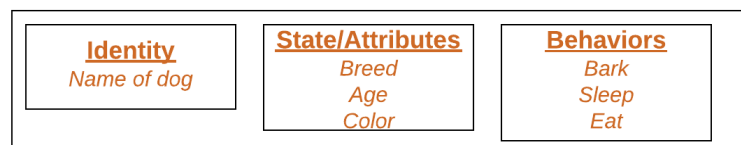
There are various types of classes that are used in real time applications such as nested classes, anonymous classes, lambda expressions.

6. Object:

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State:** It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example:



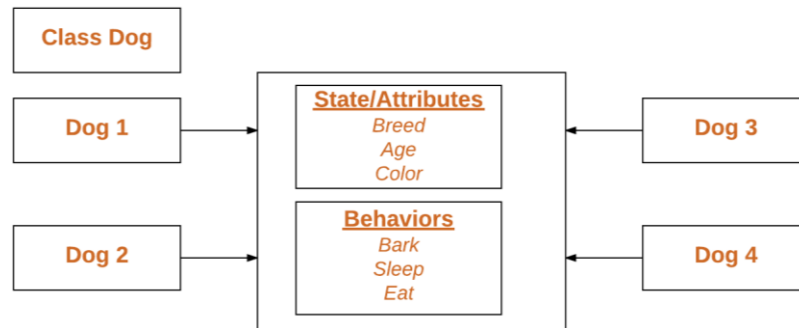
Objects correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object.

A single class may have any number of instances.

Example:



7. Methods in Java

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything.

Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++, and Python. Methods are **time savers** and help us to **reuse** the code without retyping the code.

8. Message Passing in Java

Message Passing in terms of computers is communication between processes. It is a form of communication used in object-oriented programming as well as parallel programming. Message passing in Java is like sending an object i.e. message from one thread to another thread.

It is used when threads do not have shared memory and are unable to share monitors or semaphores or any other shared variables to communicate. Suppose we consider an example of producer and consumer, likewise what producer will produce, the consumer will be able to consume that only. We mostly use **Queue** to implement communication between threads.

Message Passing in Java

