BCA Sem-5 **PAPER: BCAB3104T**

Java Programming

UNIT No. 1

Center for Distance and Online Education, PunjabiUniversity,Patiala

Lesson No:

(Syllabus)

BCAB3104T: Java Programming

Max Marks: 75 **Maximum Time: 3 Hrs**
Min Pass Marks: 35%

(A.) INSTRUCTION FOR THE PAPER SETTER

The question paper will consist of three sections A, B and C. Section A and B will have four questions from the respective section of the syllabus carrying 15 marks for each question. Section C will consist of 5-10 short answer type questions carrying a total of 15 marks, which will cover the entire syllabus uniformly. . Candidates are required to ***attempt five questions in all by selecting at least two questions each from the section A and B. Section C is compulsory.***

(B.) INSTRUCTIONS FOR THE CANDIDATES

Candidates are required to attempt five questions in all by selecting at least two questions each from the section A and B. Section C is compulsory.

SECTION-A

Introduction to java: evolution, features, comparison with C and C++; Java program structure; tokens, keywords, constants, variables, data types, type casting, statements. **Operators and expressions:** arithmetic, relational, logical, assignment, increment, decrement, conditional, bitwise and special operators. Operator precedence & associativity rules.

Control statements: if else, switch case, for, while, do while, break, continue, labeled loops.

Class: syntax, instance variable, class variables, methods, constructors, overloading of constructors and methods.

SECTION B

Inheritance: types of inheritance, use of super, method overriding, final class, abstract class, wrapper classes.

Arrays, Strings and Vectors, Packages and Interfaces, visibility controls

Errors and Exceptions: Types of errors, Exception classes, Exception handling in java, use of try, catch, finally, throw and throws. Taking user input, Command line arguments.

Multithreaded Programming: Creating Threads, Life cycle of thread, Thread priority, Thread synchronization, Inter-thread communication.

Text Book:

1. Patrick Naughton and Herbert Schildt, “The Complete Reference Java 2”, TMH

References:

2. Horstmann, Cay S. and Gary Cornell, “Core Java 2: Fundamentals Vol. 1”, Pearson Education. 3. E. Balagurusamy “Programming with Java”, TMH

BCA SEM-5 **Paper: BCAB3104T**
Java Programming

Lesson No. 1 **Author : Kanwal Preet Singh**
Converted into SLM by: Dr. Vishal Singh

Introduction to Java and Object Oriented Programming

1.1 Objectives

The objective of the lesson is to make students familiar with the history of Java. The chapter also introduces the concept of Object Oriented Programming and we will also discuss different characteristics of Object Oriented Programming and Java. After going through the lesson, you will have understanding of Java Virtual Machine and why Java is known as portable language.

1.2 Introduction

Java is a high-level, third generation programming language, like C, FORTRAN, Perl and many others. You can use Java to write computer applications that crunch numbers, process words, play games, store data or do any of the thousands of other things computer software can do.

Compared to other programming languages, Java is most similar to C. However although Java shares much of C's syntax, it is not C. Knowing how to program in C or better yet, C++, will certainly help you to learn Java more quickly, but you don't need to know C to learn Java. Unlike C++ Java is not a superset of C. A Java compiler won't compile C code and most large C programs need to be changed substantially before they can become Java programs.

What's most special about Java in relation to other programming languages is that it lets you write special programs called *applets* that can be downloaded from the Internet and played safely within a web browser. Traditional computer programs have far too much access to your system to be downloaded and executed. Although you generally trust the maintainers of various ftp archives and bulletin boards to do basic virus checking and not to post destructive software, a lot still slips through the cracks. Even more dangerous software would be propagated if any web page you visited could run programs on your system. You have no way of checking these programs for bugs or for out-and-out malicious behavior before downloading and running them.

Java solves this problem by severely restricting what an applet can do. A Java applet cannot write to your hard disk without your permission. It cannot write to arbitrary addresses in memory and thereby introduce a virus into your computer. It should not crash your system.

1.3 History

Java was started as a project called "Oak" by James Gosling in June 1991. Gosling's goals were to implement a virtual machine and a language that had a familiar C-like notation but with greater uniformity and simplicity than C/C++. The first public implementation was Java 1.0 in 1995. It made the promise of "Write Once, Run Anywhere", with free runtimes on popular platforms. It was fairly secure and its security was configurable, allowing for network and file access to be limited. The major web browsers soon incorporated it into their standard configurations in a secure "applet" configuration. It got popular quickly. New versions for large and small platforms (J2EE and J2ME) soon were designed with the advent of "Java 2". In 1997, Sun approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a proprietary de facto standard that is controlled through the Java Community Process. Sun makes most of its Java implementations available without charge, with revenue being generated by specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) which is a subset of the SDK, the primary distinction being that in the JRE the compiler is not present.

1.4 Object-oriented programming (OOP)

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. It utilizes several techniques from previously established paradigms, including inheritance, modularity, polymorphism and encapsulation. Today, many popular programming languages (such as Ada, C++, Delphi, Java, Lisp, SmallTalk, Perl, PHP, Python, Ruby, VB.Net, Visual FoxProand Visual Prolog) support OOP.

Object-oriented programming's roots reach all the way back to the 1960s, when the growing field of software engineering had begun to discuss the idea of a software crisis. As hardware and software became increasingly complex, how could software quality be maintained? Object-oriented programming addresses this problem by strongly emphasizing modularity in software.

The Simula programming language was the first to introduce the concepts underlying object-oriented programming (objects, classes, subclasses, virtual methods, coroutines, garbage collection and discrete event simulation) as a superset of Algol. Smalltalk was the first programming language to be called "object-oriented".

Object-oriented programming may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility.

Object-oriented programming came into existence because human consciousness, understanding and logic are highly object-oriented. By way of "objectifying" software modules, it is intended to promote greater flexibility and maintainability in programming, and is widely popular in large-scale software engineering. By virtue of its strong emphasis on modularity, object oriented code is intended to be simpler to develop and easier to understand later on, lending itself to more direct analysis, coding and understanding of complex situations and procedures than less modular programming methods.

1.5 Fundamental Concepts of Object Oriented Programming

A survey of computing literature, identified a number of fundamental concepts, identified in the strong majority of definitions of OOP. They are:

Class

A class defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes or properties) and the things it can do (its behaviors or methods or features). For example, the class Dog would consist of traits shared by all dogs, for example breed, fur color, and the ability to bark. Classes provide modularity and structure in an object-oriented computer program. A class should typically be recognizable to a non-programmer familiar with the problem domain, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained. Collectively, the properties and methods defined by a class are called members.

Object

A particular instance of a class is called an object. The class of Dog defines all possible dogs by listing the characteristics that they can have; the object Labbie is one particular dog, with particular versions of the characteristics. A Dog has fur; Labbie has black fur. In programmer jargon, the object Labbie is an instance of the Dog class. The set of values of the attributes of a particular object is called its state.

Method

An object's abilities. Labbie, being a Dog, has the ability to bark. So bark() is one of Labbie's methods. She may have other methods as well, for example sit() or eat(). Within the program, using a method should only affect one particular object; all Dogs can bark, but you need one particular dog to do the barking.

Inheritance

In some cases, a class will have "subclasses" i.e more specialized versions of a class. For example, the class Dog might have sub-classes called Labrador, Doberman and GoldenRetriever. In this case, Labbie would be an instance of the Labrador subclass. Subclasses inherit attributes and behaviors from their parent classes, and can introduce their own. Suppose the Dog class defines a method called bark() and a property called furColor. Each of its sub-classes (Labrador, Doberman, and GoldenRetriever) will inherit these members, meaning that the programmer only needs to write the code for them once. Each subclass can alter its inherited traits. So, for example, the Labrador class might specify that the default furColor for a Labrador is black. The Doberman subclass might specify that the bark() method is high-pitched by default. Subclasses can also add new members. The Doberman subclass could add a method called tremble(). So an individual Doberman instance would use a high-pitched bark() from the Doberman subclass, which in turn inherited the usual bark() from Dog. The Doberman object would also have the tremble() method, but Labbie would not, because she is a Labrador, not a Doberman. In fact, inheritance is an "is-a" relationship: Labbie is a Labrador. A Labrador is a Dog. Thus, Labbie inherits the members of both Labradors and Dogs. When an object or class inherits its traits from more than one ancestor class, and neither of these ancestors is an ancestor of the other, then it's called multiple inheritance.

Encapsulation

One major difference between conventional structured programming and object-oriented programming is encapsulation. **Encapsulation enables you to hide, inside the object, both the data fields and the methods that act on that data**. (In fact, data fields and methods are the two main elements of an object in the Java programming language.) After you do this, you can control access to the data, forcing programs to retrieve or modify data only through the object's interface. In strict object-oriented design, an object's data is always private to the object. Other parts of a program should never have direct access to that data.

Abstraction

Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem. In simple words when you are using a class you what work it does without knowing how it accomplishes it. Abstraction refers to the process of concentrating on the most essential aspects of the problem and ignoring the details.

Polymorphism

Polymorphism is the ability of behavior to vary based on the conditions in which the behavior is invoked, that is, two or more methods, as well as operators (such as +, -, *, among others) can fit to many different conditions. For example, if a Dog is commanded to speak() this may elicit a Bark; if a Pig is commanded to speak() this may elicit an Oink. This is expected because Pig has a particular implementation inside the speak() method. The same happens to class Dog. Considering both of them inherit speak() from Animal, this is an example of Overriding Polymorphism

1.6 Java Platform

Java is a platform for application development. A platform is a loosely defined computer industry buzzword that typically means some combination of hardware and system software that will mostly run all the same software. For instance PowerMacs running Mac OS 9.2 would be one platform. DEC Alphas running Windows NT would be another.

There's another problem with distributing executable programs from web pages. Computer programs are very closely tied to the specific hardware and operating system they run. A Windows program will not run on a computer that only runs DOS. A Mac application can't run on a UNIX workstation. VMS code can't be executed on an IBM mainframe, and so on. Therefore major commercial applications like Microsoft Word or Netscape have to be written almost independently for all the different platforms they run on. Netscape is one of the most cross-platform of major applications and it still only runs on a minority of platforms.

Java solves the problem of platform-independence by using byte code. The Java compiler does not produce native executable code for a particular machine like a C compiler would. Instead it produces a special format called *byte code*. Java byte code written in hexadecimal, byte by byte, looks like this:

```
CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08
```

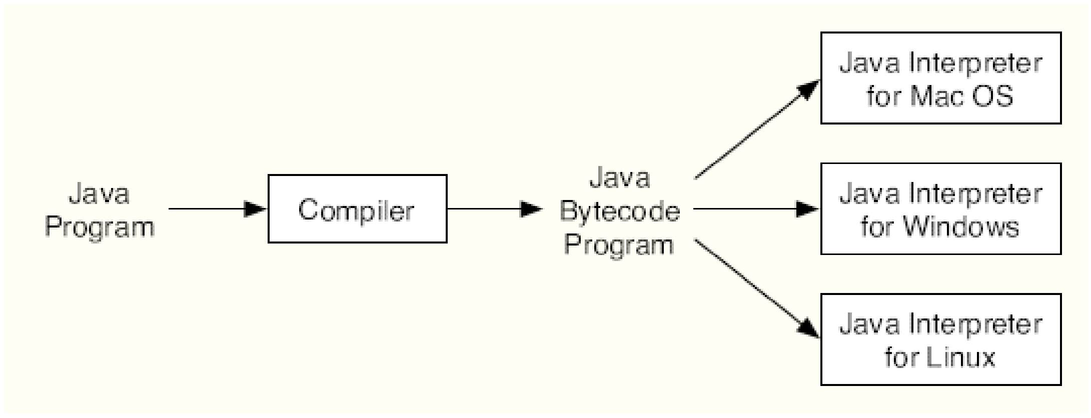
This looks a lot like machine language, but unlike machine language Java byte code is exactly the same on every platform. This byte code fragment means the same thing on a Solaris workstation as it does on a Macintosh PowerBook. Java programs that have been compiled into byte code still need an interpreter to execute them on any given platform. The interpreter reads the byte code and translates it into the native language of the host machine on the fly. The most common such interpreter is Sun's program java (with a little j). Since the byte code is completely platform independent, only the interpreter and a few native libraries need to be ported to get Java to run on a new computer or operating system. The rest of the runtime environment including the compiler and most of the class libraries are written in Java.

All these pieces, the javac compiler, the java interpreter, the Java programming language and more are collectively referred to as Java.

1.7 The Java Virtual Machine

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Pascal or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a compiler. A **compiler** takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language. There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an **interpreter**, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so. One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: they can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, there is a program called "Virtual PC" that runs on Macintosh computers. Virtual PC is an interpreter that executes machine-language programs written for IBM-PC-clone computers. If you run Virtual PC on your Macintosh, you can run any PC program, including programs written for Windows. (Unfortunately, a PC program will run much more slowly than it would on an actual IBM clone. The problem is that Virtual PC executes several Macintosh machine-language instructions for each PC machine-language instruction in the program it is interpreting. Compiled programs are inherently faster than interpreted programs.)

The designers of Java chose to use a combination of compilation and interpretation. Pro-grams written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the Java Virtual Machine(JVM). The machine language for the Java virtual machine is called Java byte-code. There is no reason why Java bytecode could not be used as the machine language of a real computer, rather than a virtual computer. However, one of the main selling points of Java is that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the Java virtual machine in the same way that Virtual PC simulates a PC computer. Of course, a different Jave bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program. And the same Java bytecode program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.


Java Program
Compiler
Java Bytecode Program
Java Interpreter for Mac OS
Java Interpreter for Windows
Java Interpreter for Linux

Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are many reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program. A Java bytecode interpreter, on the other hand, is a fairly small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer can run any compiled Java program. It would be much harder to write a Java compiler for the same computer. Furthermore, many Java programs are meant to be downloaded over a network. This leads to obvious security concerns: you don't want to download and run a program that will damage your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download. You are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect you from potentially dangerous actions on the part of that program. You should note that there is no necessary connection between Java and Java bytecode. A pro-gram written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages could be compiled into Java bytecode. However, it is the combination of Java and Java bytecode that is platform-independent, secure and network-compatible while allowing you to program in a modern high-level object-oriented language.

1.8 Java portability

So applets are really JVM code that is translated into the correct machine code by the browser. That's why applets can run on any machine and C and C++ programs cannot. This same two step translation can be used for Java applications as well as applets. If a program is in JVM code, then it is *portable* and can be run on any machine that has a JIT compiler on it.

C and C++ object programs are not portable because they are always translated into the machine language for a specific machine. If you are very careful when you write your C programs, you may be able to compile and run the source program on many machines, but there are often big problems. For example, how big is a C int? On some machines it is 16 bits, on some machines it is 32 bits, and some it is 64 bits. Java has defined the basic types very exactly (eg, ints are always 32 bits). Another big problem is that lack of standard library functions in C/C++. There are some standard library

functions for C/C++, but they are very small compared to Java and don't do many of the things that programmers need to do (*eg* build a graphical user interface).
Portability is extremely important to many companies, since large companies usually have many kinds of computers. If a company can "write once and run everywhere" they can save an enormous amount of money. If a developer can do this, they will have a much larger market for their software.

bytecode

Programming code that, once compiled, is run through a virtual machine instead of the computers processors. By using this approach, source code can be run on any platform once it has been compiled and run through the virtual machine.

Bytecode is the compiled format for Java programs. Once a java program has been converted to bytecode, it can be transferred across a network and executed by Java Virtual Machine (JVM). Bytecode files generally have a .class extension.

1.9 Features of Java

The main features of Java are:

Simple

Java was designed to make it much easier to write bug free code. Java has the bare bones functionality needed to implement its rich feature set. It does not add lots of syntactic sugar or unnecessary features. Despite its simplicity Java has considerably more functionality than C, primarily because of the large class library.

About half of the bugs in C and C++ programs are related to memory allocation and deallocation. Therefore the second important addition Java makes to providing bug-free code is automatic memory allocation and deallocation. The C library memory allocation functions malloc() and free() are gone as are C++'s destructors.

Java is an excellent teaching language and an excellent choice with which to learn programming. The language is small so it's easy to become fluent. The language is interpreted so the compile-run-link cycle is much shorter. The runtime environment provides automatic memory allocation and garbage collection so there's less for the programmer to think about. Java is object-oriented unlike Basic so the beginning programmer doesn't have to unlearn bad programming habits when moving into real world projects. Finally, it's very difficult (if not quite impossible) to write a Java program that will crash your system, something that you can't say about any other language.

Object-Oriented

Object oriented programming is the catch phrase of computer programming in the 1990's. Although object oriented programming has been around in one form or another since the Simula language was invented in the 1960's, it's really begun to take hold in modern GUI environments like Windows, Motif and the Mac. In object-oriented programs data is represented by objects. Objects have two sections, fields (instance variables) and methods. Fields tell you what an object is. Methods tell you what an object does. These fields and methods are closely tied to the object's real world characteristics and behavior. When a program is run messages are passed back and forth between objects. When an object receives a message it responds accordingly as defined by its methods.

Object oriented programming is alleged to have a number of advantages including:

- Simpler, easier to read programs
- More efficient reuse of code
- Faster time to market
- More robust, error-free code

Platform Independent(Portable)

Java was designed to not only be cross-platform in source form like C, but also in compiled binary form. Since this is frankly impossible across processor architectures Java is compiled to an intermediate form called byte-code. A Java program never really executes natively on the host machine. Rather a special native program called the Java interpreter reads the byte code and executes the corresponding native machine instructions. Thus to port Java programs to a new platform all that is needed is to port the interpreter and some of the library routines. Even the compiler is written in Java. The byte codes are precisely defined and remain the same on all platforms.

The second important part of making Java cross-platform is the elimination of undefined or architecture dependent constructs. Integers are always four bytes long, and floating point variables follow the IEEE 754 standard for computer arithmetic exactly. You don't have to worry that the meaning of an integer is going to change if you move from a Pentium to a PowerPC. In Java everything is guaranteed.

However the virtual machine itself and some parts of the class library must be written in native code. These are not always as easy or as quick to port as pure Java programs.

Secure

Java was designed from the ground up to allow for secure execution of code across a network, even when the source of that code was untrusted and possibly malicious. This required the elimination of many features of C and C++. Most notably there are no pointers in Java. Java programs cannot access arbitrary addresses in memory. All memory access is handled behind the scenes by the (presumably) trusted runtime environment. Furthermore Java has strong typing. Variables must be declared, and variables do not change types when you aren't looking. Casts are strictly limited to casts between types that make sense. Thus you can cast an int to a long or a byte to a short but not a long to a boolean or an int to a String.

Java implements a robust exception handling mechanism to deal with both expected and unexpected errors. The worst that an applet can do to a host system is bring down the runtime environment. It cannot bring down the entire system.

Most importantly Java applets can be executed in an environment that prohibits them from introducing viruses, deleting or modifying files or otherwise destroying data and crashing the host computer. A Java enabled web browser checks the byte codes of an applet to verify that it doesn't do anything nasty before it will run the applet.

However the biggest security problem is not hackers. It's not viruses. It's not even insiders erasing their hard drives and quitting your company to go to work for your competitors. No, the biggest security issue in computing today is bugs. Regular, ordinary, non-malicious unintended bugs are responsible for more data loss and lost productivity than all other factors combined. Java, by making it easier to write bug-free code, substantially improves the security of all kinds of programs.

Multi-Threaded

Java is inherently multi-threaded. A single Java program can have many different threads executing independently and continuously. Three Java applets on the same page can run together with each getting equal time from the CPU with very little extra effort on the part of the programmer. This makes Java very responsive to user input. It also helps to contribute to Java's robustness and provides a mechanism whereby the Java environment can ensure that a malicious applet doesn't steal all of the host's CPU cycles.

There is a cost associated with multi-threading. Multi-threading is to Java what pointer arithmetic is to C, that is, a source of devilishly hard to find bugs. Nonetheless, in simple programs it's possible to leave multi-threading alone and normally be OK.

Dynamically linked(Dynamic)

Java does not have an explicit link phase. Java source code is divided into .java files, roughly one per class in your program. The compiler compiles these into .class files containing byte code. Each .java file generally produces exactly one .class file. (There are a few exceptions we'll discuss later, non-public classes and inner classes).

The compiler searches the current directory and directories specified in the CLASSPATH environment variable to find other classes explicitly referenced by name in each source code file. If the file you're compiling depends on other, non-compiled files the compiler will try to find them and compile them as well. The compiler is quite smart and can handle circular dependencies as well as methods that are used before they're declared. It also can determine whether a source code file has changed since the last time it was compiled.

More importantly, classes that were unknown to a program when it was compiled can still be loaded into it at runtime. For example, a web browser can load applets of differing classes that it's never seen before without recompilation.

Furthermore, Java .class files tend to be quite small, a few kilobytes at most. It is not necessary to link in large runtime libraries to produce a (non-native) executable. Instead the necessary classes are loaded from the user's CLASSPATH.

Automatic Garbage Collection

You do not need to explicitly allocate or deallocate memory in Java. Memory is allocated as needed, both on the stack and the heap and reclaimed by the *garbage collector* when it is no longer needed. There's no malloc(), free() or destructor methods. There are constructors and these do allocate memory on the heap, but this is transparent to the programmer.

The exact algorithm used for garbage collection varies from one virtual machine to the next. The most common approach in modern VMs is generational garbage collection for short-lived objects, followed by mark and sweep for longer lived objects.

1.10 Java and Internet

Computers can be connected together on networks. A computer on a network can communicate with other computers on the same network by exchanging data and files or by sending and receiving messages. Computers on a network can even work together on a large computation. Today, millions of computers throughout the world are connected to a single huge network called the Internet. New computers are being connected to the Internet every day. Computers can join the Internet by using a modem to establish a connection through telephone lines. Broadband connections to the Internet, such as DSL and cable modems are increasingly common. They allow faster data transmission than is possible through telephone modems.

Java is intimately associated with the Internet and the World-Wide Web. Special Java programs called applets are meant to be transmitted over the Internet and displayed on Web pages. A Web server transmits a Java applet just as it would transmit any other type of information. A Web browser that understands Java—that is, that includes an interpreter for the Java virtual machine—can then run the applet right on the Web page. Since applets are programs, they can do almost anything, including complex interaction with the user. With Java, a Web page becomes more than just a passive display of information. It becomes anything that programmers can imagine and implement. But applets are only one aspect of Java's relationship with the Internet, and not the major one. In fact, as both Java and the Internet have matured, applets have become less important. At the same time, however, Java has increasingly been used to write complex, stand-alone applications that do not depend on a web browser. Many of these programs are network-related. For example many of the largest and most complex web sites use web server software that is written in Java. Java includes excellent support for network protocols and its platform independence makes it possible to write network programs that work on many different types of computer. Its association with the Internet is not Java's only advantage. But many good programming languages have been invented only to be soon forgotten. Java has had the good luck to ride on the coattails of the Internet's immense and increasing popularity.

1.11 Advantages of JAVA

Java™ has significant advantages over other languages and environments that make it suitable for just about any programming task.

The advantages of Java are as follows:

- Java is easy to learn. Java was designed to be easy to use and is therefore easy to write, compile, debug and learn than other programming languages.
- Java is object-oriented. This allows you to create modular programs and reusable code.
- Java is platform-independent. One of the most significant advantages of Java is its ability to move easily from one computer system to another. The ability to run the same program on many different systems is crucial to World Wide Web software, and Java succeeds at this by being platform-independent at both the source and binary levels.

Because of Java's robustness, ease of use, cross-platform capabilities and security features, it has become a language of choice for providing worldwide Internet solutions.

1.12 Difference between C++ and Java

The different goals in the development of C++ and Java resulted in different principles and design trade-offs between the languages. The differences are as follows :

C++	**Java**
Compatible with C source code, except for a few corner cases.	No backward compatibility with any previous language. The syntax is however strongly influenced by C/C++.
Write once compile anywhere (WOCA)	Write once run anywhere / everywhere (WORA / WORE)
Allows both procedural programming and object-oriented programming.	Encourages an object oriented programming paradigm.
Allows direct calls to native system libraries.	Call through the Java Native Interface and recently Java Native Access
Exposes low-level system facilities.	Runs in a protected virtual machine.
Only provides object types and type names.	Is reflective, allowing metaprogramming and dynamic code generation at runtime.
Has multiple binary compatibility standards (commonly Microsoft and Itanium/GNU)	Has a binary compatibility standard, allowing runtime check of correctness of libraries.
Optional automated bounds checking. (e.g. the at() method invector and string containers)	Normally performs bounds checking. HotSpot can remove bounds checking.
Supports native unsigned arithmetic.	No native support for unsigned arithmetic.
Standardized minimum limits for all numerical types, but the actual sizes are implementation-defined. Standardized types are available as typedefs (uint8_t, ..., uintptr_t).	Standardized limits and sizes of all primitive types on all platforms.
Pointers, References and pass by value are supported	Primitive and reference data types always passed by value.
Explicit memory management, though third party frameworks exist to provide garbage collection. Supports destructors.	Automatic garbage collection (can be triggered manually). Doesn't have the concept of Destructor and usage of finalize() is not recommended.
Supports class, struct, and union and can	Supports only class and allocates them

allocate them on heap orstack	on the heap. Java SE 6optimizes with escape analysis to allocate some objects on the stack.
Allows explicitly overriding types.	Rigid type safety except for widening conversions. Autoboxing/Unboxing added in Java 1.5.
The C++ Standard Library has a much more limited scope and functionality than the Java standard library but includes: Language support, Diagnostics, General Utilities, Strings, Locales, Containers, Algorithms, Iterators, Numerics, Input/Output and Standard C Library. The Boost library offers much more functionality including threads and network I/O. Users must choose from a plethora of (mostly mutually incompatible) third-party libraries for GUI and other functionality.	The standard library has grown with each release. By version 1.6 the library included support for locales, logging, containers and iterators, algorithms, GUI programming (but not using the system GUI), graphics, multi-threading, networking, platform security, introspection, dynamic class loading, blocking and non-blocking I/O, and provided interfaces or support classes for XML, XSLT, MIDI, database connectivity, naming services (e.g. LDAP), cryptography, security services (e.g. Kerberos), print services, and web services. SWT offers an abstraction for platform specific GUIs.
Operator overloading for most operators	The meaning of operators is generally immutable, however the + and += operators have been overloaded for Strings.
Full multiple inheritance, including virtual inheritance.	Single inheritance only from classes, multiple from interfaces.
Compile time Templates	Generics are used to achieve an analogous effect to C++ templates, however they do not translate from source code to byte code due to the use of Type Erasure by the compiler.
Function pointers, function objects, lambdas (in C++0x) and interfaces	No function pointer mechanism. Instead idioms such as Interfaces, Adapters and Listeners are extensively used.
No standard inline documentation mechanism. 3rd party software (e.g.Doxygen) exists.	Javadoc standard documentation
const keyword for defining immutable variables	final provides a limited version of const,

and member functions that do not change the object.	equivalent to type* constpointers for objects and plain const of primitive types only. No constmember functions, nor any equivalent to const type* pointers.
Supports the goto statement.	Supports labels with loops and statement blocks.
Source code can be written to be platform independent (can be compiled for Windows, BSD, Linux, Mac OS X, Solaris etc. without needing modification) and written to take advantage of platform specific features. Is typically compiled into native machine code.	Is compiled into byte code for the JVM. Is dependent on the Java platform but the source code is typically written not to be dependent on operating system specific features.

1.13 Summary

Java is a high-level, third generation programming language, like C, FORTRAN, Perl, and many others. Java was started as a project called "Oak" by James Gosling in June 1991. Gosling's goals were to implement a virtual machine and a language that had a familiar C-like notation but with greater uniformity and simplicity than C/C++. The first public implementation was Java 1.0 in 1995. Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. It utilizes several techniques from previously established paradigms, including inheritance, modularity, polymorphism and encapsulation. The designers of Java chose to use a combination of compilation and interpretation. Pro-grams written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the Java Virtual Machine(JVM). The machine language for the Java virtual machine is called Java byte-code. The main features of Java are: simple, object-oriented, platform independent, secure, multithreaded, dynamic and automatic garbage collection.

1.14 Short Answer Type Questions

1. When and by whom was Java created?
2. What is Java Virtual Machine?
3. Why is Java known as a platform independent language?

1.15 Long Answer Type Questions

1. What is Object Oriented Programming? Explain its different characteristics.
2. Explain different features of Java in detail.

1.16 Suggested Readings

- The Complete Reference by Herbert Scheild
- Programming with Java by E.Balagurusamy
- Java : A Beginner's Guide by Herbert Schildt
- Introduction to Java Programming by Y.Daniel Liang.
- Object Oriented Programming in Java by G.T. Thampi

BCA SEM-5 **Paper : BCAB3104T**
Java Programming

Lesson No. 2 **Author : Kanwal Preet Singh**
Converted into SLM by: Dr. Vishal Singh

Program Structure and Data Types

2.1 Objectives

After reading this lesson you will be able to understand:

- How to compile and run a java program
- The different program elements used in Java
- Keywords
- Constants
- Variables
- Datatypes

2.2 Introduction

In Programming languages such as C and C++, a program is either compiled or interpreted. But in Java, the program is both compiled and interpreted. The program is first compiled to generate Java Byte Code which is then interpreted to machine language. A Java program consists of comments, keywords, constants, variables, etc. The main method acts as a starting point of the program and main method is itself defined within some class.

2.3 The First Java Program

A Java program is a collection of one or more java classes. A Java source file can contain more than one class definition and has a .java extension. Each class definition in a source file is compiled into a separate class file. The name of this compiled file is comprised of the name of the class with .class as an extension. Below is a java sample code for the traditional Hello World program. Basically, the idea behind this Hello World program is to learn how to create a program, compile and run it. To create your java source code you can use any editor (Text pad/Edit plus etc).

```
public class HelloWorld
 {
  public static void main(String[] args)
    {
      System.out.println("Hello World");
    }//End of main
 }//End of HelloWorld Class
```

Output is:

```
Hello World
```

About The Program

We created a class named "HelloWorld" containing a simple main function within it. The keyword class specifies that we are defining a class. The name of a public class is spelled exactly as the name of the file (Case Sensitive). All java programs begin execution with the method named main(). main method that gets executed has the following signature : **public static void main(String args[])**.Declaring this method as public means that it is accessible from outside the class so that the JVM can find it when it looks for the program to start it. It is necessary that the method is declared with return type void (i.e. no arguments are returned from the method). The main method contains a String argument array that can contain the command line arguments. The brackets { and } mark the beginning and ending of the class. The program contains a line 'System.out.println("Hello World");' that tells the computer to print out on one line of text namely 'Hello World'. The semi-colon ';' ends the line of code. The double slashes '//' are used for comments that can be used to describe what a source code is doing. Everything to the right of the slashes on the same line does not get compiled, as they are simply the comments in a program.

Java Main method Declarations

```
class MainExample1 {public static void main(String[] args) {}}
class MainExample2 {public static void main(String []args) {}}
class MainExample3 {public static void main(String args[]) {}}
```

All the 3 valid main method's shown above accepts a single String array argument.

Compiling and Running an Application

To compile and run the program you need the JDK distributed by Sun Microsystems. The JDK contains documentation, examples, installation instructions, class libraries and packages and tools. Download an editor like Textpad/EditPlus to type your code. You must save your source code with a .java extension. The name of the file must be the name of the public class contained in the file.

Steps for Saving, compiling and Running a Java

Step 1:Save the program With .java Extension.

Step 2:Compile the file from DOS prompt by typing javac <filename>.

Step 3:Successful Compilation, results in creation of .class containing byte code

Step 4:Execute the file by typing java <filename without extension>

2.4 Basic Language Elements (Lexical Structure)

The lexical structure of a programming language is the set of elementary rules that define what are the ***tokens*** or basic atoms of the program. It is the lowest level syntax of a language and specifies what is punctuation, reserved words, identifiers, constants and operators. **Some of the basic rules for Java are:**

- Java ***is*** case sensitive.
- Whitespace, tabs, and newline characters are ignored except when part of string constants. They can be added as needed for readability.
- Single line comments begin with //
- Multiline comments begin with /* and end with */
- Statements terminate in semicolons! Make sure to ***always*** terminate statements with a semicolon.
- Commas are used to separate words in a list.
- Round brackets are used for operator precedence and argument lists.
- Square brackets are used for arrays and square bracket notation.
- Curly or brace brackets are used for blocks.
- ***Keywords*** are reserved words that have special meanings within the language syntax.
- ***Identifiers*** are names for constants, variables, functions, properties, methods and objects. The first character ***must*** be a letter, underscore or dollar sign. Following characters can also include digits. Letters are A to Z, a to z, and Unicode characters above hex 00C0. Java styling uses initial capital letter on object identifiers, uppercase for constant ids and lowercase for property, method and variable ids.

  Note: an identifier must ***NOT*** be any word on the **Java Reserved Word List (Keywords)**.

2.5 Keywords

There are certain words with a specific meaning in java which tell (help) the compiler what the program is supposed to do. These Keywords cannot be used as variable names, class names or method names. Keywords in java are case sensitive, all characters being lower case. Keywords are reserved words that are predefined in the language; see the table below (Taken from Sun Java Site). All the keywords are in lowercase.

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

2.6 Comments

Comments are descriptions that are added to a program to make code easier to understand. The compiler ignores comments and hence they are only for documentation of the program. Java supports two comment styles.

Block style comments begin with **/*** and terminate with ***/** that spans multiple lines.

Line style comments begin with **//** and terminate at the end of the line.

2.7 Constants And Variables

If there's one thing that every computer program has in common, it's that they always process data input and produce some sort of output based on that data. And because data is so important to a computer program, it stands to reason that there must be plenty of different ways to store data so that programs can do their processing correctly and efficiently. In order to keep track of data, programs use constants and variables. In this lesson, you discover what constants and variables are, as well as learn to use them in the Java language.

Constants

If you think about the term "constant" for a few moments, you might conclude that constants must have something to do with data that never changes. And your conclusion would be correct. A constant is nothing more than a value, in a program, that stays the same throughout the program's execution. However, while the definition of a constant is fairly simple, constants themselves can come in many different guises. For example, the numeral 2, when it's used in a line of program code, is a constant. If you place the word "Java" in a program, the characters that comprise the word are also constants. In fact, these constant characters taken together are often referred to as a string constant.

NOTE: To be entirely accurate, I should say that text and numerals that are placed in program code are actually called literals, because the value is literally, rather than symbolically, in the program. If this literal and symbolic stuff is confusing you, you'll probably have it figured out by the end of this lesson. For now, just know that I'm lumping literals in with constants to simplify the discussion. Such values as the numeral 2 and the string constant "Java" are sometimes called hard-coded values because the values that represent the constants are placed literally in the program code. For example, suppose you were writing a program and wanted to calculate the amount of sales tax on a purchase. Suppose further that the total purchase in question is Rs. 120.00 and the sales tax in your state is 10 percent. The calculation that'll give you the sales tax would look like this:

```
tax = 120 * .10;
```

Suppose now that you write a large program that uses the sales tax percentage in many places. Then, after you've been happily using your program for a few months, the state suddenly decides to raise the sales tax to eleven percent. In order to get your program working again, you have to go through every line of code, looking for the .10 values that represent the sales tax and changing them to .11. Such a modification can be a great deal of work in a large program. Worse, you may miss one or two places in the code that need to be changed, leaving your program with some serious bugs.

To avoid these situations, programmers often use something called symbolic constants, which are simply words that represent values in a program. In the case of your sales tax program, you could choose a word like SALESTAX (no spaces) to represent the current sales tax percentage for your state. Then, at the beginning of your program, you set SALESTAX to be equal to the current state sales tax. In the Java language, such a line of program code might look like this:

```
final float SALESTAX = 0.10;
```

In the preceding line, the word final tells Java that this data object is going to be a constant. The float is the data type, which, in this case, is a floating point. (You'll learn more about data types later in this lesson.) The word SALESTAX is the symbolic constant. The equals sign tells Java that the word on the left should be equal to the value on the right, which, in this case, is 0.10. After defining the symbolic constant SALESTAX, you can rewrite any lines that use the current sales tax value to use the symbolic constant rather than the hard-coded value. For example, the calculation for the sales tax on that Rs.120.00 purchase might now look something like this:

```
tax = 120 * SALESTAX;
```

TIP: In order to differentiate symbolic constants from other values in a program, programmers often use all uppercase letters when naming these constants. Now, when your state changes the sales tax to 11 percent, you need only change the value you assign to the symbolic constant and the rest of the program automatically fixes itself. The change would look like this:

```
final float SALESTAX = 0.11;
```

Variables

If constants are program values that cannot be changed throughout the execution of a program, what are variables? **Variables** are values that can change as much as needed during the execution of a program. Because of a variable's changing nature, there's no such thing as a hard-coded variable. That is, hard-coded values in a program are always constants (or, more accurately, literals). Why do programs need variables? Think back to the sales tax program from the previous section. You may recall that you ended up with a program line that looked like this:

```
tax = 12 * SALESTAX;
```

In this line, the word tax is a variable. So, one reason you need variables in a program is to hold the results of a calculation. In this case, you can think of the word tax as a kind of digital bucket into which the program dumps the result of its sales tax calculation. When you need the value stored in tax, you can just reach in and take it out-figuratively speaking, of course. As an example, to determine the total cost of a Rs.120.00 purchase, plus the sales tax, you might write a line like this:

```
total = 120 + tax;
```

In this line, the word total is yet another variable. After the computer performs the requested calculation, the variable total will contain the sum of 120 and whatever value is stored in tax. For example, if the value 12 is stored in tax, after the calculation, total would be equal to 132.

Do you see another place where a variable is necessary? How about the hard-coded value 120? Such a hard-coded value makes a program pretty useless because every person that comes into your store to buy something isn't going to spend exactly Rs.120.00. Because the amount of each customer's purchase will change, the value used in your sales tax calculation must change, too. That means you need another variable. How about creating a variable named purchase? With such a variable, you can rewrite the calculations like this:

```
tax = purchase * SALESTAX;
total = purchase + tax;
```

Now you can see how valuable variables can be. Every value in the preceding lines is represented by a variable. (Although you're using SALESTAX as a symbolic constant, because of Java's current lack of true constants, it's really a variable, too.) All you have to do to get the total to charge your customer is plug the cost of his purchase into the variable purchase.

2.8 Literals

By literal we mean any number, text or other information that represents a value. This means what you type is what you get. We will use literals in addition to variables in Java statement. While writing a source code as a character sequence, we can specify any value as a literal such as an integer. This character sequence will specify the syntax based on the value's type. This will give a literal as a result. For instance

```
int month  = 10;
```

In the above statement the literal is an integer value i.e 10. The literal is 10 because it directly represents the integer value. In Java programming language there are some special type of literals that represent numbers, characters, strings and boolean values. Lets have a closer look on each of the following.

Integer Literals

Integer literals is a sequence of digits and a suffix as L. To represent the type as long integer we use L as a suffix. We can specify the integers either in decimal, hexadecimal or octal format. To indicate a **decimal format** put the left most digit as nonzero. Similarly put the characters as *ox* to the left of at least one hexadecimal digit to indicate **hexadecimal format**. Also we can indicate the **octal format** by a zero digit followed by the digits 0 to 7. Lets tweak the table below.

659L	Decimal integer literal of type long integer
0x4a	Hexadecimal integer literal of type integer
057L	Octal integer literal of type long integer

Character Literals

We can specify a character literal as a single printable character in a pair of single quote characters such as 'a', '#', and '3'. You must be knowing about the ASCII character set. The ASCII character set includes 128 characters including letters, numerals, punctuations etc. There are few character literals which are not readily printable through a keyboard. The table below shows the codes that can represent these special characters. The letter d such as in the octal, hex etc represents a number.

Escape	**Meaning**
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Formfeed
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\d	Octal
\xd	Hexadecimal
\ud	Unicode character

It is very interesting to know that if we want to specify a single quote, a backslash or a nonprintable character as a character literal use an **escape sequence.** An escape sequence uses a special syntax to represents a character. The syntax begins with a single backslash character. Lets see the table below in which the character literals use Unicode escape sequence to represent printable and nonprintable characters both.

'u0041'	Capital letter A
'\u0030'	Digit 0
'\u0022'	Double quote "
'\u003b'	Punctuation ;
'\u0020'	Space

'\u0009'	Horizontal Tab

Boolean Literals

The values true and false are also treated as literals in Java programming. When we assign a value to a boolean variable, we can only use these two values. Unlike C, we can't presume that the value of 1 is equivalent to true and 0 is equivalent to false in Java. We have to use the values true and false to represent a Boolean value. Like boolean chosen = true; Remember that the **literal true** is not represented by the quotation marks around it. The Java compiler will take it as a string of characters, if its in quotation marks.

Floating-point literals

Floating-point numbers are like real numbers in mathematics, for example, 4.13179, -0.000001. Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double.

Type	**Size**		**Range**	**Precision**
name	bytes	bits	approximate	in decimal digits
float	4	32	+/- 3.4 * 10^{38}	6-7
double	8	64	+/- 1.8 * 10^{308}	15

A floating-point literal can be denoted as a decimal point, a fraction part, an exponent (represented by E or e) and as an integer. We also add a suffix to the floating point literal as D, d, F or f. The type of a floating-point literal defaults to double-precision floating-point. The following floating-point literals represent double-precision floating-point and floating-point values.

6.5E+32 (or 6.5E32)	Double-precision floating-point literal
7D	Double-precision floating-point literal
.01f	Floating-point literal

String Literals

The string of characters is represented as String literals in Java. In Java a string is not a basic data type, rather it is an object. These strings are not stored in arrays as in C language. There are few methods provided in Java to combine strings, modify strings and to know whether to strings have the same value.

We represent string literals as

String myString = "How are you?";

The above example shows how to represent a string. It consists of a series of characters inside double quotation marks.

Lets see some more examples of string literals:

```
""                          // the empty string
"\""                        // a string containing "
"This is a string"          // a string containing 16 characters
```

```
"This is a " +                // actually a string-valued constant expression,
"two-line string"             // formed from two string literals
```

Strings can include the character escape codes as well, as shown here:

```
String example = "Your Name, \"Sumit\"";
System.out.println("Thankingyou,\nRichards\n");
```

Null Literals

The final literal that we can use in Java programming is a Null literal. We specify the Null literal in the source code as 'null'. To reduce the number of references to an object, use null literal. The type of the null literal is always null. We typically assign null literals to object reference variables. For instance

```
    s = null;
```

2.9 Naming Constants and Variables

The first computer languages were developed by mathematicians. For that reason, the calculations and the variables used in those calculations were modeled after the types of equations mathematicians were already accustomed to working with. For example, in the old days, the lines in your tax program might have looked like this:

```
a = b * c;
 d = b + a;
```

As you can see, this type of variable-naming convention left a lot to be desired. It's virtually impossible to tell what type of calculations are being performed. In order to understand their own programs, programmers had to use tons of comments mixed in with their source code so they could remember what the heck they were doing from one programming session to the next. Although adding comments to the program lines helps a little, the code is still pretty confusing, because you don't really know what the variables a, b, c, and d stand for. After a while (probably when mathematicians weren't the only programmers), someone came up with the idea of allowing more than one character in a variable name, which would enable the programmer to create mathematical expressions that read more like English. Thus, the confusing example above would be written as:

```
// Calculate the amount of sales tax.
tax = purchase * SALESTAX;
// Add the sales tax to the purchase amount.
 total = purchase + tax;
```

By using carefully chosen variable names, you can make your programs self documenting, which means that the program lines themselves tell whoever might be reading the program what the program does. If you strip away the comments from the preceding example, you can still see what calculations are being performed. Of course, there are rules for choosing constant and variable names (also known as identifiers because they identify a program object). You can't just type a bunch of characters on your keyboard and expect Java to accept them. The rules for naming identifiers are as follows:

1) First, every Java identifier must begin with one of these characters:
    - **A-Z**
    - **a-z**

- _
- **$**

The preceding characters are any uppercase letter from A through Z, any lowercase letter from a through z, an underscore and the dollar sign.

2) Following the first character, the rest of the identifier can use any of these characters:
    - **A-Z**
    - **a-z**
    - _
    - **$**
    - **0-9**

    As you may have noticed, this second set of characters is very similar to the first. In fact, the only difference is the addition of the digits from 0 through 9.

Using the rules given, the following are valid identifiers in a Java program:

- number
- number2
- amount_of_sale
- $amount

The following identifiers are not valid in a Java program:

- 1number
- amount of sale
- &amount
- item#

2.10 Data Types

Variables have a **data type**, that indicates the kind of value they can store and the amount of memory space they occupy. You may remember my mentioning two data types already, these being floating point (represented by the float keyword) and integer (represented by the int keyword). Java has eight different data types, all of which represent different kinds of values in a program. These data types are byte, short, int, long, float, double, char and boolean. In this section, you'll learn what kinds of values these various data types represent.

2.10.1 Integer Values

The most common values used in computer programs are integers, which represent whole number values such as 17, 1978, and -26. Integer values can be both positive or negative, or even the value 0. The size of the value that's allowed depends on the integer data type you choose. Java features four integer data types, which are **byte, short, int, and long**. Although some computer languages allow both signed and unsigned integer values, all of Java's integers are signed, which means they can be positive or negative. (Unsigned values, which Java does not support, can hold only positive numbers.)

byte

The first integer type, **byte**, takes up the least amount of space in a computer's memory. When you declare a constant or variable as byte, you are limited to values in the range **-128 to 127**. Why would you want to limit the size of a value in this way? Because the smaller the data type, the faster the computer can manipulate it. For example, your computer can move a byte value, which consumes only eight bits(One Byte) of memory, much faster than an int value, which, in Java, is four times as large. In Java, you declare a byte value like this:

byte identifier;

In the preceding line, byte is the data type for the value and identifier is the variable's name. You can also simultaneously declare and assign a value to a variable like this:

byte count = 100;

After Java executes the preceding line, your program will have a variable named count that currently holds the value of 100. Of course, you can change the contents of count at any time in your program. It only starts off holding the value 100.

short

The next biggest type of Java integer is short. It takes 2 Bytes of memory. A variable declared as short can hold a value from -32,768 to 32,767. You declare a short value like this:

short identifier;

or

short identifier = value;

In the preceding line, value can be any value from -32,768 to 32,767, as described previously. In Java, short values are twice as big in memory-16 bits (or two bytes)-as byte values.

int

Next in the integer data types is int, which can hold a value from -2,147,483,648 to 2,147,483,647. Now you're getting into some big numbers! The int data type can hold such large numbers because it takes up 32 bits (four bytes) of computer memory. You declare int values like this:

int identifier;

or

int identifier = value;

long

The final integer data type in the Java language is long, which takes up a whopping 64 bits (eight bytes) of computer memory and can hold truly immense numbers. Unless you're calculating the number of molecules in the universe, you don't even have to know how big a long number can be. I'd figure it out for you, but I've never seen a calculator that can handle numbers that big. You declare a long value like this:

long identifier;

or

long identifier = value;

Note: How do you know which integer data type to use in your program? Choose the smallest data type that can hold the largest numbers you'll be manipulating. Following this rule keeps your programs running as fast as possible. However, having said that, I should tell you that most programmers (including me) use the int data type a lot, even when they can get away with a byte.

2.10.2 Floating-Point Values

Whereas integer values can hold only whole numbers, the floating-point data types can hold values with both whole number and fractional parts. Examples of floating-point values include 61.8, 123.284, and -23.456. As you can see, just like integers, floating-point values can be either positive or negative.

Java includes two floating-point types, which are **float** and **double**. Each type allows greater precision in calculations. What does this mean? Floating-point numbers can become very complex when they're used in calculations, particularly in multiplication and division. For example, when you divide 3.9 by 2.7, you get 1.44444444. In actuality, though, the fractional portion of the number goes on forever. That is, if you were to continue the division calculation, you'd discover that you keep getting more and more fours in the fractional part of the answer. The answer to 3.9 divided by 2.7 is not really 1.44444444, but rather something more like 1.4444444444444444. But even that answer isn't completely accurate. A more accurate answer would be 1.444444444444444444444444444444444. The more 4s you add to the answer the more accurate the answer becomes-yet, because the 4s extend on into infinity, you can never arrive at a completely accurate answer. Dealing with floating-point values frequently means deciding how many decimal places in the answer is accurate enough. That's where the difference between the float and double data types shows up. In Java, a value declared as float can hold a number in the range from around -3.402823 x 10 38 to around 3.402823 x 10 38. These types of values are also known as single-precision floating-point numbers and take up 32 bits (four bytes) of memory. You declare a single-precision floating-point number like this:

float identifier;

or

float identifier = value;

In the second line, value must be a value in the range given in the previous paragraph, followed by an upper- or lowercase F. However, you can write floating-point numbers in a couple of ways, using regular digits and a decimal point or using scientific notation. This value is the type of floating-point number you're used to seeing:

356.552

Now, here's the same number written using Java's rules, in both the number's normal form and in the form of scientific notation:

356.552f
3.56552e2f

Both of the preceding values are equivalent and you can use either form in a Java program. The e2 in the second example is the equivalent of writing x 102 and is a short form of scientific notation that's often used in programming languages.

Note: If you're not familiar with scientific notation, the value 3.402823 x 10 38 is equal to 3.402823 times a number that starts with a 1 and is followed by 38 zeroes. Computer languages shorten this scientific notation to 3.402823e38.

The second type of floating-point data, **double**, represents a double-precision value, which is a much more accurate representation of floating-point numbers because it allows for more decimal places. A double value can be in the range from -1.79769313486232 x 10 308 to 1.79769313486232 x 10 308 and is declared like this:

double identifier;

or

double identifier = value;

Floating-point values of the double type are written exactly as their float counterparts, except you use an upper- or lowercase D as the suffix, rather than an F. Here are a few examples:

```
3.14d
344.23456D
3.4423456e2d
```

Note: When using floating-point numbers in your programs, the same rule that you learned about integers applies: Use the smallest data type you can. This is especially true for floating-point numbers, which are notorious for slowing computer programs to a crawl. Unless you're doing highly precise programming, such as 3-D modeling, the single-precision float data type should do just fine.

2.10.3 Character Values

Often in your programs, you'll need a way to represent character values rather than just numbers. A character is a symbol that's used in text. The most obvious examples of characters are the letters of the alphabet, in both upper- and lowercase varieties. There are, however, many other characters, including not only things such as spaces, exclamation points and commas, but also tabs, carriage returns, and line feeds. The symbols 0 through 9 are also characters when they're not being used in mathematical calculations.

In order to provide storage for character values, Java features the char data type, which is 16 bits. However, the size of the char data type has little to do with the values it can hold. Basically, you can think of a char as being able to hold a single character. (The 16 bit length accommodates Unicode characters, which you don't need to worry about in this book.) You declare a char value like this:

char c;

or

char c = 'A';

In the second example, you're not only declaring the variable c as a char, but also setting its value to an uppercase A. Notice that the character that's being assigned is enclosed in single quotes.

Some characters cannot be written with only a single symbol. For example, the tab character is represented in Java as \t, which is a backslash followed by a lowercase t. There are several of these special characters, as shown in the following Table.

Special Character Literals

Character	**Symbol**
Backslash	\\
Backspace	\b
Carriage return	\r
Double quote	\"
Form feed	\f
Line feed	\n
Single quote	\'
Tab	\t

Although the special characters in the above Table are represented by two symbols, the first of which is always a backslash, you still use them as single characters. For example, to define a char variable as a backspace character, you might write something like the following in your Java program:

```
char backspace = '\b';
```

When Java's compiler sees the backslash, it knows that it's about to encounter a special character of some type. The symbol following the backslash tells the compiler which special character to use. Because the backslash is used to signify a special character, when you want to specify the backslash character yourself, you must use two backslashes, which keeps the compiler from getting confused.

Other special characters that might confuse the compiler because they are used as part of the Java language are single and double quotes. When you want to use these characters in your program's data, you must also precede them with a backslash.

2.10.4 Boolean Values

Many times in a program, you need a way to determine if a specific condition has been met. For example, you might need to know whether a part of your program executed properly. In such cases, you can use Boolean values, which are represented in Java by the **boolean data type**. Boolean values are unique in that they can be only one of two possible values: **true or false**. You declare a boolean value like this:

boolean identifier;

or

boolean identifier = value;

In the second example, value must be true or false. In an actual program, you might write something like this:

```
boolean file_okay = true;
```

Boolean values are often used in if statements, which enable you to do different things depending on the value of a variable.

The following Table summarizes Java's various data types. Take some time now to look over the table and make sure you understand how the data types differ from each other. You might also want to think of ways you might use each data type in an actual program.

Summary of Java's Data Types				
Type	**Values**	**Default**	**Size**	**Range**
byte	signed integers	0	8 bits	-128 to 127
short	signed integers	0	16 bits	-32768 to 32767
int	signed integers	0	32 bits	-2147483648 to 2147483647
long	signed integers	0	64 bits	-9223372036854775808 to 9223372036854775807
float	floating point	0.0	32 bits	+/-1.4E-45 to +/-3.4028235E+38
double	floating point	0.0	64 bits	+/-4.9E-324 to +/-1.7976931348623157E+308
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
boolean	true, false	False	1 bit used in 32 bit integer	NA

2.11 Variable Scope

When you write your Java programs, you can't just declare your variables willy-nilly all over the place. You first have to consider how and where you need to use the variables. This is because variables have an attribute known as scope, which determines where in your program variables can be accessed. In Java, a variable's scope is determined by the program block in which the variable first appears.

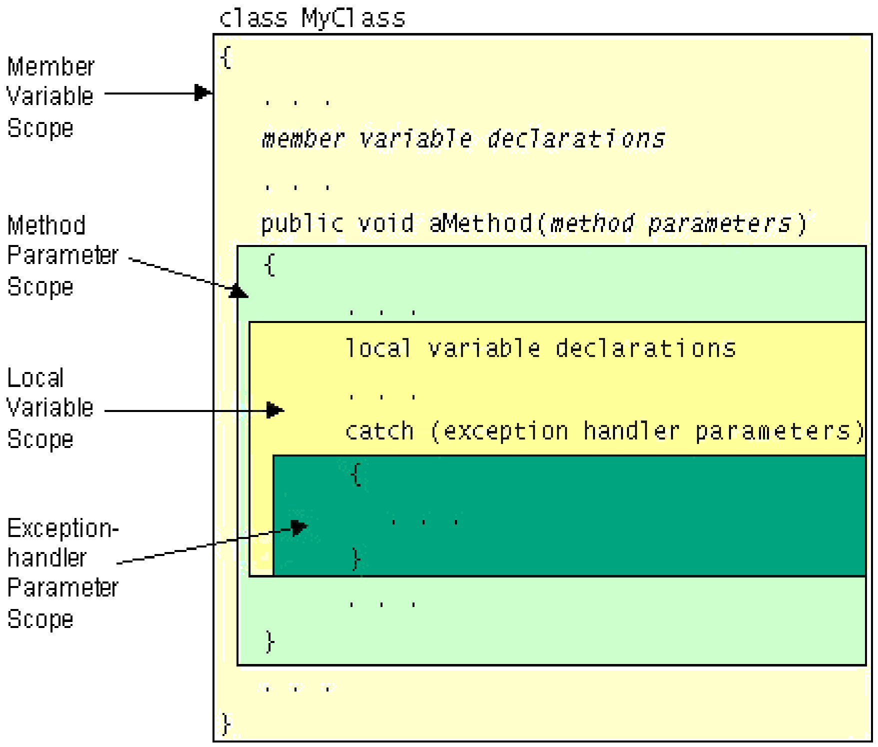
The variable is "visible" to the program only from the beginning of its program block to the end of the program block. When a program's execution leaves a block, all the variables in the block disappear, a phenomenon that programmers call "going out of scope."

Now you're probably wondering, "What is a program block?" Generally, a program block is a section of program code that starts with an opening curly brace ({) and ends with a closing curly brace (}). (Sometimes, the beginning and ending of a block are not explicitly defined, but you don't have to worry about that just yet.) Specifically, program blocks include things like classes, functions and loops, all of which you'll learn about later. Of course, things aren't quite as simple as all that. The truth is that you can have program blocks within other program blocks. When you have one block inside another, the inner block is considered to be nested.

The ability to nest program blocks adds a wrinkle to the idea of variable scope. Because a variable remains in scope from the beginning of its block to the end of its block, such a variable is also in scope in any blocks that are nested in the variable's block.

A variable's scope is the block of code within which the variable is accessible and determines when the variable is created and destroyed. The location of the variable declaration within your program establishes its scope and places it into one of these four categories:

- Member variable
- Local variable
- Method parameter
- Exception-handler parameter


class MyClass
{
. . .
member variable declarations
. . .
public void aMethod(method parameters)
{
. . .
local variable declarations
. . .
catch (exception handler parameters)
{
. . .
}
. . .
}
. . .
}
Member Variable Scope
Method Parameter Scope
Local Variable Scope
Exception-handler Parameter Scope

A member variable is a member of a class or an object. It can be declared anywhere in a class but not in a method. The member is available to all code in the class. You can declare local variables anywhere in a method or within a block of code in a method. In general, a local variable is accessible from its declaration to the end of the code block in which it was declared.

Method parameters are formal arguments to methods and constructors and are used to pass values into methods and constructions. The scope of a method parameter is the entire method or constructor for which it is a parameter. Exception-handler parameters are similar to method parameters but are arguments to an exception handler rather than to a method or a constructor.

2.12 Dynamic Initialization

There are two types of variable mainly:

1. Instance Variable or Class Variable
2. Local Variable or method variable

Instance variable are initialized by JVM to their default values if not defined explicitly.Whereas the local variables needs to be defined each time time they are declared.But the local variables can be used to a greater effect by using the concept of dynamic initialization. Dynamic Initialization can be defined as the dynamic operation that allows variables to be initialized dynamically using any expression valid at the time of the variable declaration.

Above definition implies that if you need a variable to store value of an expression you can use dynamic initialization.In which value of an expression is assigned to a variable.

Dynamic Initialization can be clear by understanding following example:-

```
        class DynamicInit{
public static void main(String a[]){
int a=2;
int b=5;
int c=a*a+b*b;
System.out.println("value of c is "+c);
}
}
```

Output :-The output of above program is value of c is 29

In the above program we have three variables a,b and c.Each has been declared as ' int '.Variable a and b are declared and provided values on declaration whereas the variable c has been assigned a expression to evaluate and store its value.This is done through dynamic evaluation.

2.13 Summary

In Java, the program is both compiled and interpreted. The program is first compiled to generate Java Byte Code which is then interpreted to machine language. A Java program is a collection of one or more java classes. A Java source file can contain more than one class definition and has a .java extension. Each class definition in a source file is compiled into a separate class file. The name of this compiled file is comprised of the name of the class with .class as an extension. To compile and run the program you need the JDK distributed by Sun Microsystems. There are certain words with a specific meaning in java which tell (help) the compiler what the program is supposed to do. These Keywords cannot be used as variable names, class names, or method names. Comments are descriptions that are added to a program to make code easier to understand. The compiler ignores comments and hence they are only for documentation of the program. A constant is nothing more than a value, in a program, that stays the same throughout the program's execution. **Variables** are values that can change as much as needed during the execution of a program. Variables have a **data type** that indicates the kind of value they can store and the amount of memory space they occupy. In Java, we can make use of integer, floating point, character and Boolean data types. In Java, a variable's scope is determined by the program block in which the variable first appears.

2.14 Short Answer Type Questions

1. What are keywords?
2. What are constants and variables?
3. What do you mean by variable scope?

2.15 Long Answer Type Questions

1. Explain with the help of an example how to create, save, compile and execute a program in java.
2. Explain different data types used in Java in detail.
3. WAP to print "Hello World".

2.16 Suggested Readings

- The Complete Reference by Herbert Scheild
- Programming with Java by E.Balagurusamy
- Java : A Beginner's Guide by Herbert Schildt
- Introduction to Java Programming by Y.Daniel Liang.
- Object Oriented Programming in Java by G.T. Thampi
- Java Programming by C. Xavier

Lesson No. 3 **Author : Kanwal Preet Singh**
Converted into SLM by: Dr. Vishal Singh

Operators in Java

3.1 Objectives

After reading this lesson you will be able to understand:

- Java Arithmetic Operators
- Java Assignment Operators
- Java Relational Operators
- Java Boolean Operators
- Java Conditional Operators
- Bitwise operators
- Operator Precedence

3.2 Introduction

Most of the expressions in Java use operators. Operators are special symbols for things like arithmetic, various forms of assignment, increment and decrement and logical operations. An operator performs a function on either one, two or three operands. An operator that requires one operand is called a **unary operator**. For example, ++ is a unary operator that increments the value of its operand by 1. An operator that requires two operands is a **binary operator**. For example, = is a binary operator that assigns the value from its right-hand operand to its left-hand operand. And finally a **ternary operator** is one that requires three operands. The Java programming language has one ternary operator, ?:, which is a short-hand if-else statement. The unary operators support either prefix or postfix notation. Prefix notation means that the operator appears *before* its operand:

operator op

Postfix notation means that the operator appears *after* its operand:

op operator

All of the binary operators use infix notation, which means that the operator appears *between* its operands:

op1 operator op2

The ternary operator is also infix; each component of the operator appears between operands:

expr ? op1 : op2

In addition to performing the operation, an operator also returns a value. The return value and its type depends on the operator and the type of its operands. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and subtraction, return numbers--the result of the operation. The data type returned by the arithmetic operators depends on the type of its operands: If you add two integers, you get an integer back. An operation is said to *evaluate to* its result. It's useful to divide the operators into different categories.

3.3 Java Arithmetic Operators

The Java programming language includes five simple arithmetic operators. They are **+ (addition), - (subtraction), * (multiplication), / (division)**, and **% (modulo).** The following table summarizes the binary arithmetic operators in the Java programming language.

Use	**Returns true if**
op1 **+** op2	op1 added to op2
op1 **-** op2	op2 subtracted from op1
op1 ***** op2	op1 multiplied with op2
op1 **/** op2	op1 divided by op2
op1 **%** op2	Computes the remainder of dividing op1 by op2

The following java program, defines two integers and two double-precision floating-point numbers and uses the five arithmetic operators to perform different arithmetic operations. This program also uses + to concatenate strings. The arithmetic operations are shown in boldface.

```
public class ArithmeticProg
 {
  public static void main(String[] args)
    {
        //a few numbers
        int i = 10;
        int j = 20;
        double x = 10.5;
        double y = 20.5;

        //adding numbers
        System.out.println("Adding");
        System.out.println(" i + j = " + (i + j));
        System.out.println(" x + y = " + (x + y));

        //subtracting numbers
        System.out.println("Subtracting");
        System.out.println(" i - j = " + (i - j));
        System.out.println(" x - y = " + (x - y));

        //multiplying numbers
        System.out.println("Multiplying");
        System.out.println(" i * j = " + (i * j));
        System.out.println(" x * y = " + (x * y));

        //dividing numbers
        System.out.println("Dividing");
        System.out.println(" i / j = " + (i / j));
        System.out.println(" x / y = " + (x / y));

        //computing the remainder resulting
        //from dividing numbers
        System.out.println("Modulus");
        System.out.println(" i % j = " + (i % j));
        System.out.println(" x % y = " + (x % y));
    }
}
```

3.4 Java Assignment Operators

Java Variables are assigned or given, values using one of the assignment operators. The variable are always on the left-hand side of the assignment operator and the value to be assigned is always on the right-hand side of the assignment operator. The assignment operator is evaluated from right to left, so a = b = c = 0; would assign 0 to c, then c to b then b to a.

```
i = i + 2;
```

Here we say that we are assigning i's value to the new value which is i+2. A shortcut way to write assignments like this is to use the += operator. It is one operator symbol so don't put blanks between the + and =.

```
i += 2; // Same as "i = i + 2"
```

The shortcut assignment operator can be used for all Arithmetic Operators_i.e. you can use this style with all arithmetic operators (+, -, *, /, and even %). Here are some examples of assignments:

```
//assign 1 to variable a
int a = 1;
//assign the result of 2 + 2 to b
int b = 2 + 2;
//assign the literal "Hello" to str
String str = new String("Hello");
//assign b to a, then assign a
//to d; results in d, a, and b being equal
int d = a = b;
```

Java Increment and Decrement Operators

There are 2 Increment or decrement operators ++ and --. These two operators are unique in that they can be written both before the operand they are applied to called prefix increment/decrement, or after, called postfix increment/decrement. The meaning is different in each case.

Example:

```
x = 1;
y = ++x;
System.out.println(y);
```

prints 2, but

```
x = 1;
y = x++;
System.out.println(y);
```

prints 1

Source Code

```
//Count to ten
class UptoTen
 {
  public static void main (String args[])
   {
     int i;
     for (i=1; i <=10; i++)
      {
        System.out.println(i);
      }
   }
 }
```

When we write i++ we're using shorthand for i = i + 1. When we say i-- we're using shorthand for i = i - 1. Adding and subtracting one from a number are such common operations that these special increment and decrement operators have been added to the language. There's another short hand for the general add and assign operation, +=. We would normally write this as i += 15. Thus if we wanted to count from 0 to 20 by two's we'd write:

Source Code

```
class CountToTwenty
  {
     public static void main (String args[])
      {
       int i;
       for (i=0; i <=20; i += 2)
        {  //Note Increment Operator by 2
          System.out.println(i);
        }
      } //main ends here
   }
```

As you might guess there is a corresponding -= operator. If we wanted to count down from twenty to zero by twos we could write: -=

```
class CountToZero
  {
    public static void main (String args[])
     {
      int i;
      for (i=20; i >= 0; i -= 2)
       {  //Note Decrement Operator by 2
        System.out.println(i);
       }
     }
  }
```

The ? : operator in Java

The value of a variable often depends on whether a particular boolean expression is or is not true and on nothing else. For instance one common operation is setting the value of a variable to the maximum of two quantities. In Java you might write

```
if (a > b) {
  max = a;
}
else {
  max = b;
}
```

Setting a single variable to one of two states based on a single condition is such a common use of if-else that a shortcut has been devised for it, the conditional operator, ?:. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

(a > b) ? a : b; is an expression which returns one of two values, a or b. The condition, (a > b), is tested. If it is true the first value, a, is returned. If it is false, the second value, b, is returned. Whichever value is returned is dependent on the conditional test, a > b. The condition can be any expression which returns a boolean value.

3.5 Java Relational Operators

A relational operator compares two values and determines the relationship between them. For example, != returns true if its two operands are unequal. Relational operators are used to test whether two values are equal, whether one value is greater than another, and so forth. The relation operators in Java are: ==, !=, <, >, <=, and >=. The meanings of these operators are:

Use	**Returns true if**
op1 > op2	op1 is greater than op2
op1 >= op2	op1 is greater than or equal to op2
op1 < op2	op1 is less than to op2
op1 <= op2	op1 is less than or equal to op2
op1 == op2	op1 and op2 are equal
op1 != op2	op1 and op2 are not equal

The main use for the above relational operators are in CONDITIONAL phrases. The following java program is an example, that defines three integer numbers and uses the relational operators to compare them.

```
public class RelationalProg
 {
   public static void main(String[] args)
    {
       //a few numbers
       int i = 37;
       int j = 42;
       int k = 42;
       //greater than
       System.out.println("Greater than...");
       System.out.println(" i > j = " + (i > j)); //false
       System.out.println(" j > i = " + (j > i)); //true
```

```
        System.out.println(" k > j = " + (k > j)); //false
        //(they are equal)

        //greater than or equal to
        System.out.println("Greater than or equal to...");
        System.out.println(" i >= j = " + (i >= j)); //false
        System.out.println(" j >= i = " + (j >= i)); //true
        System.out.println(" k >= j = " + (k >= j)); //true

        //less than
        System.out.println("Less than...");
        System.out.println(" i < j = " + (i < j)); //true
        System.out.println(" j < i = " + (j < i)); //false
        System.out.println(" k < j = " + (k < j)); //false

        //less than or equal to
        System.out.println("Less than or equal to...");
        System.out.println(" i <= j = " + (i <= j)); //true
        System.out.println(" j <= i = " + (j <= i)); //false
        System.out.println(" k <= j = " + (k <= j)); //true

        //equal to
        System.out.println("Equal to...");
        System.out.println(" i == j = " + (i == j)); //false
        System.out.println(" k == j = " + (k == j)); //true

        //not equal to
        System.out.println("Not equal to...");
        System.out.println(" i != j = " + (i != j)); //true
        System.out.println(" k != j = " + (k != j)); //false
    }
}
```

3.6 Java Boolean Operators

The Boolean logical operators are : **| , & , ^ , ! , || , &&** . Java supplies a primitive data type called Boolean, instances of which can take the value true or false only, and have the default value false. The major use of Boolean facilities is to implement the expressions which control if decisions and while loops. These operators act on Boolean operands according to this table:

A	B	A|B	A&B	A^B	!A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true

true true true true false false

The details of the various operators are given in the following table:

Boolean Operators x and y are boolean types. x and y can be expressions that result in a boolean value. Result is a boolean true or false value.		
x && y	Conditional AND	If both x and y are true, result is true. If either x or y are false, the result is false If x is false, y is not evaluated.
x & y	Boolean AND	If both x and y are true,the result is true. If either x or y are false, the result is false Both x and y are evaluated before the test.
x || y	Conditional OR	If either x or y are true, the result is true. If x is true, y is not evaluated.
x | y	Boolean OR	If either x or y are true, the result is true. Both x & y are evaluated before the test.
!x	Boolean NOT	If x is true, the result is false. If x is false, the result is true.
x ^ y	Boolean XOR	If x is true and y is false, the result is true. If x is false and y is true, the result is true. Otherwise, the result is false. Both x and y are evaluated before the test.

Example

```
class Bool1
 {
   public static void main(String args[])
    {
      // these are boolean variables
      boolean A = true;
      boolean B = false;
      System.out.println("A|B = "+(A|B));
      System.out.println("A&B = "+(A&B));
      System.out.println("!A = "+(!A));
      System.out.println("A^B = "+(A^B));
      System.out.println("(A|B)&A = "+((A|B)&A));
    }
  }
```

3.7 Java Conditional Operators

Java has the conditional operator. It's a ternary operator that is, it has three operands and it comes in two pieces, ? and :, that have to be used together. It takes the form:

Boolean-expression ? expression-1 : expression-2

The JVM tests the value of **Boolean-expression**. If the value is true, it evaluates e**xpression-1**; otherwise, it evaluates **expression-2**. For Example:

```
if (a > b)
 {
   max = a;
 }
else
 {
   max = b;
 }
```

Setting a single variable to one of two states based on a single condition is such a common use of if-else that a shortcut has been devised for it, the conditional operator, ?:. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

3.8 Bitwise operators

The bitwise operators allow you to manipulate individual bits in an integral primitive data type. Bitwise operators perform Boolean algebra on the corresponding bits in the two arguments to produce the result. The bitwise operators come from C's low-level orientation; you were often manipulating hardware directly and had to set the bits in hardware registers. Java was originally designed to be embedded in TV set-top boxes, so this low-level orientation still made sense. However, you probably won't use the bitwise operators much.

The bitwise AND operator (&) produces a one in the output bit if both input bits are one; otherwise it produces a zero. The bitwise OR operator (|) produces a one in the output bit if either input bit is a one and produces a zero only if both input bits are zero. the bitwise, EXCLUSIVE OR, or XOR (^), produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise NOT (~, also called the ones complement operator) is a unary operator; it takes only one argument. (All other bitwise operators are binary operators.) Bitwise NOT produces the opposite of the input bit – a one if the input bit is zero, a zero if the input bit is one.

Bitwise operators can be combined with the = sign to unite the operation and assignment: &=, |= and ^= are all legitimate. (Since ~ is a unary operator it cannot be combined with the = sign.) The **boolean** type is treated as a one-bit value so it is somewhat different. You can perform a bitwise AND, OR and XOR, but you can't perform a bitwise NOT (presumably to prevent confusion with the logical NOT). For **boolean**s the bitwise operators have the same effect as the logical operators except that they do not short circuit. Also, the bitwise operators on **boolean**s gives you a XOR logical operator that is not included under the list of "logical" operators.

The shift operators shift the individual bits of the operand to the left or right as indicated by the operator. Each shift operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself. For example, the following statement shifts the bits of the integer 13 to the right by one position:

13 >> 1;

The binary representation of the number 13 is 1101. The result of the shift operation is 1101 shifted to the right by one position--110 or 6 in decimal. Note that the bit farthest to the right falls off the end into the bit bucket. The different bit manipulation operators are summarized in the table below:

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Zero fill right shift
~	Bitwise complement
<<=	Left shift assignment (x = x << y)
>>=	Right shift assignment (x = x >> y)
>>>=	Zero fill right shift assignment (x = x >>> y)
x&=y	AND assignment (x = x & y)
x|=y	OR assignment (x = x | y)
x^=y	XOR assignment (x = x ^ y)

3.9 Operator Precedence

Operator precedence determines the order in which expressions are evaluated. This, in some cases, can determine the overall value of the expression. For example, take the following expression:

y = 6 + 4 / 2

Depending on whether the 6 + 4 expression or the 4 / 2 expression is evaluated first, the value of y can end up being 5 or 8. Operator precedence determines the order in which expressions are evaluated, so you can predict the outcome of an expression. In general, increment and decrement are evaluated before arithmetic, arithmetic expressions are evaluated before comparisons and comparisons are evaluated before logical expressions. Assignment expressions are evaluated last.

The following table shows the specific precedence of the various operators in Java. Operators further up in the table are evaluated first; operators on the same line have the same precedence and are evaluated left to right based on how they appear in the expression itself. For example, given that same expression y = 6 + 4 / 2, you now know, according to this table, that division is evaluated before addition, so the value of y will be 8.

Operator precedence

Operator	**Explanation**
. [] ()	Parentheses (()) are used to group expressions; dot (.) is used for access to methods and variables within objects and classes (discussed tomorrow); square brackets ([]) are used for arrays
++ -- ! ~ instanceof	The instanceof operator returns true or false based on whether the object is an instance of the named class or any of that class's subclasses
new (type)expression	The new operator is used for creating new instances of classes; () in this case is for casting a value to another type
* / %	Multiplication, division, modulus
+ -	Addition, subtraction
<< >> >>>	Bitwise left and right shift
< > <= >=	Relational comparison tests
== !=	Equality
&	AND
^	XOR
|	OR
&&	Logical AND
||	Logical OR
? :	Shorthand for if...then...else (discussed on Day 5)
= += -= *= /= %= ^=	Various assignments
&= |= <<= >>= >>>=	More assignments

3.10 Type Casting and Conversion

It is sometimes necessary to convert a data item of one type to another type. Conversion of data from one type to another type is known as type casting. In java one object reference can be type cast into another object reference. This casting may be of its own type or to one of its subclass or superclasss types or interfaces. Some compile time or runtime type casting rules are there in java. Some circumstances requires automatic type conversion, while in other cases it must be "forced" manually (explicitly).

Automatic Conversion

Java performs automatic type conversion when the type of the expression on the right hand side of an assignment operator safely promotes to the type of the variable on the left hand side of the assignment operator. Thus we can safely assign: byte -> short -> int -> long -> float -> double. Symbol (->) used here interprets to "to a".

Lets take an example,

```
// 64 bit long integer
long myLongInteger;
// 32 bit standard integer
int myInteger;
myLongInteger = myInteger;
```

In the above example, extra storage associated with the long integer, simply results in padding with extra zeros. Any object can reference to a reference variable of the type Object, as Object class comes at the top in the hierarchy of every Java class. Java follows two types of casting

- Upcasting
- Downcasting

Upcasting

Casting a reference with the class hierarchy in a direction from the sub classes towards the root then this type of casting is termed as upcasting. Upcasting does not require a cast operator.

DownCasting

On the other hand, casting a reference with hierarchal class order in a direction from the root class towards the children or subclasses, then it is known as downcasting.

Explicit Conversion (Casting)

Automatic type casting does work in case of narrowing i.e. when a data type requiring more storage is converted into a data type that requires less storage. E.g. conversion of a long to an int does not perform because the first requires more storage than the second and consequently information may be lost. In such situations an explicit conversion is required.

3.11 Summary

Operators are special symbols for things like arithmetic, various forms of assignment, increment and decrement and logical operations. An operator performs a function on one, two, or three operands. An operator that requires one operand is called a **unary operator**. An operator that requires two operands is a **binary operator a**nd a **ternary operator** is one that requires three operands. The Java programming language includes five simple arithmetic operators. They are **+ (addition), - (subtraction), * (multiplication), / (division)**, and **% (modulo).** Java Variables are assigned, or given, values using one of the assignment operators. The variable are always on the left-hand side of the assignment operator and the value to be assigned is always on the right-hand side of the assignment operator. There are 2 Increment or decrement operators ++ and – to increase and decrease the value of a variable by one respectively. A relational operator compares two values and determines the relationship between them. The relation operators in Java are: ==, !=, <, >, <=, and >=. The Boolean logical operators are : **| , & , ^ , ! , || , && .** Java supplies a primitive data type called Boolean, instances of which can take the value true or false only, and have the default value false. Java has the conditional operator. It's a ternary operator that is, it has three operands and it comes in two pieces, ? and :, that have to be used together. The bitwise operators allow you to manipulate individual bits in an integral primitive data type. Bitwise operators perform Boolean algebra on the corresponding bits in the two arguments to produce the result. Operator precedence determines the order in which expressions are evaluated.

3.12 Short Answer Type Questions

1. What is an operator and what are operands?
2. Explain different arithmetic operators?
3. What do you mean by precedence of operators?

3.13 Long Answer Type Questions

1. Explain in detail different operators used in java.
2. WAP to find roots of a quadratic equation.
3. WAP to show the all of unary operator.

3.14 Suggested Readings

- The Complete Reference by Herbert Scheild
- Programming with Java by E.Balagurusamy
- Java : A Beginner's Guide by Herbert Schildt
- Introduction to Java Programming by Y.Daniel Liang.
- Object Oriented Programming in Java by G.T. Thampi
- Java Programming by C. Xavier

Lesson No. 4 **Author : Kanwal Preet Singh**
Converted into SLM by: Dr. Vishal Singh

Control Statements

4.1 Objectives

After reading this lesson you will have understanding of various decision and iteration statements. You will also have an understanding of return, break, continue and switch statements.

4.2 Introduction

Java uses all of C's execution control statements, so if you've programmed with C or C++ then most of what you see will be familiar. Most procedural programming languages have some kind of control statements and there is often overlap among languages. In Java, the keywords include **if-else**, **while**, **do-while**, **for**, and a selection statement called **switch**. Java does not, however, support the **goto** (which can still be the most expedient way to solve certain types of problems). You can still do a goto-like jump, but it is much more constrained than a typical **goto**.

4.3 true and false

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is **A == B**. This uses the conditional operator **==** to see if the value of **A** is equivalent to the value of **B.** The expression returns **true** or **false**. Any of the relational operators you've seen earlier can be used to produce a conditional statement. Note that Java doesn't allow you to use a number as a **boolean**, even though it's allowed in C and C++ (where truth is nonzero and falsehood is zero). If you want to use a non-**boolean** in a **boolean** test, such as **if(a)**, you must first convert it to a **boolean** value using a conditional expression, such as **if(a != 0)**.

4.4 if-else

The **if-else** statement is probably the most basic way to control program flow. The **else** is optional, so you can use **if** in two forms:

```
if(Boolean-expression)
statement
or
if(Boolean-expression)
statement
else
statement
```

The conditional must produce a Boolean result. The *statement* means either a simple statement terminated by a semicolon or a compound statement, which is a group of simple statements enclosed in braces. Anytime the word "*statement*" is used, it always implies that the statement can be simple or compound. As an example of **if-else**, here is a **test( )** method that will tell you whether a guess is above, below or equivalent to a target number:

```
static int test(int testval)
 {
  int result = 0;
  if(testval > target)
   result = -1;
  else if(testval < target)
   result = +1;
  else
   result = 0; // match
  return result;
}
```

It is conventional to indent the body of a control flow statement so the reader might easily determine where it begins and ends.

4.5 return

The **return** keyword has two purposes: it specifies what value a method will return (if it doesn't have a **void** return value) and it causes that value to be returned immediately. The **test()** method above can be rewritten to take advantage of this:

```
static int test2(int testval)
 {
  if(testval > target)
   return -1;
  if(testval < target)
   return +1;
  return 0; // match
}
```

There's no need for **else** because the method will not continue after executing a **return**.

4.6 Iteration

while, **do-while** and **for** control looping and are sometimes classified as *iteration statements*. A *statement* repeats until the controlling *Boolean-expression* evaluates to false. The form for a **while** loop is

```
while(Boolean-expression)
statement
```

The *Boolean-expression* is evaluated once at the beginning of the loop and again before each further iteration of the *statement*. Here's a simple example that generates random numbers until a particular condition is met:

```
//WhileTest.java
// Demonstrates the while loop
public class WhileTest
 {
   public static void main(String[] args)
     {
       double r = 0;
       while(r < 0.99d)
         {
           r = Math.random();
           System.out.println(r);
         }
     }
  }
```

This uses the **static** method **random( )** in the **Math** library, which generates a **double** value between 0 and 1. (It includes 0, but not 1.) The conditional expression for the **while** says "keep doing this loop until the number is 0.99 or greater." Each time you run this program you'll get a different-sized list of numbers.

do-while

The form for **do-while** is

```
do
statement
while(Boolean-expression);
```

The sole difference between **while** and **do-while** is that the statement of the **do-while** always executes at least once, even if the expression evaluates to false the first time. In a **while**, if the conditional is false the first time the statement never executes. In practice, **do-while** is less common than **while**.

for

A **for** loop performs initialization before the first iteration. Then it performs conditional testing and at the end of each iteration, some form of "stepping." The form of the **for** loop is:

```
for(initialization; Boolean-expression; step)
statement
```

Any of the expressions *initialization*, *Boolean-expression* or *step* can be empty. The expression is tested before each iteration and as soon as it evaluates to **false** execution will continue at the line following the **for** statement. At the end of each loop, the *step* executes. **for** loops are usually used for "counting" tasks:

```
//ListCharacters.java
// Demonstrates "for" loop by listing
// all the ASCII characters.
public class ListCharacters
 {
   public static void main(String[] args)
    {
     for( char c = 0; c < 128; c++)
       if (c != 26 ) // ANSI Clear screen
         System.out.println("value: " + (int)c +" character: " + c);
    }
  }
```

Note that the variable **c** is defined at the point where it is used, inside the control expression of the **for** loop, rather than at the beginning of the block denoted by the open curly brace. The scope of **c** is the expression controlled by the **for**. Traditional procedural languages like C require that all variables be defined at the beginning of a block so when the compiler creates a block it can allocate space for those variables. In Java and C++ you can spread your variable declarations throughout the block, defining them at the point that you need them. This allows a more natural coding style and makes code easier to understand. You can define multiple variables within a **for** statement, but they must be of the same type:

```
for(int i = 0, j = 1;i < 10 && j != 11;i++, j++)
/* body of for loop */;
```

The **int** definition in the **for** statement covers both **i** and **j**. The ability to define variables in the control expression is limited to the **for** loop. You cannot use this approach with any of the other selection or iteration statements.

Nested for loops

We saw how to create a for loop in Java. This is generally where a single variable cycles over a range of values.

Very often, it's useful to cycle through combinations of two or more variables. For example, we printed out a single times table, but what if we wanted to print out all the times tables, say, between 2 and 12 ?

One common way to do this is to use a nested loop. That's just a fancy way to saying one loop inside another. So we start with one loop, which goes through all the "times table numbers" in turn that we want to print (in this case, we said between 2 and 12).

```
for (int timsTableNo = 2; timesTableNo <= 12; timesTableNo++) {
}
```

Then, inside this loop, we place our other loop that printed the 7 times table. Only this time, we don't want it to print the 7 times table each time – we want to print the *timesTableNo* times table each time. So the program looks like this :

```
for (int timesTableNo = 2; timesTableNo <= 12; timesTableNo++)        {
system.out.printIn("The" + timesTableNo + "times table:");
int result = n * timesTableNo;
System.out.printIn(timesTableNo + "times" + n + "eqals" + n);
}
}
```

Now, the lines in bold will run for all combinations of time table number and n. Notice that each time table is also preceded by a "heading" that announces it as "The 2 times table" etc. The line to do that sits only inside the timesTableNo loop, so it only gets run once for every times table, not once for every combination. Try running the program and confirming that it prints all the times tables, for every value between 1 and 12.

4.7 The comma operator

Earlier in this lesson we have seen that the comma *operator* (not the comma *separator*, which is used to separate function arguments) has only one use in Java: in the control expression of a **for** loop. In both the initialization and step portions of the control expression you can have a number of statements separated by commas, and those statements will be evaluated sequentially. The previous bit of code uses this ability. Here's another example:

```
//CommaOperator.java
public class CommaOperator
 {
  public static void main(String[] args)
   {
    for(int i = 1, j = i + 10; i < 5;i++, j = i * 2)
     {
       System.out.println("i= " + i + " j= " + j);
     }
   }
 }
```

The output is:

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

You can see that in both the initialization and step portions the statements are evaluated in sequential order. Also, the initialization portion can have any number of definitions *of one type*.

4.8 break and continue

Inside the body of any of the iteration statements you can also control the flow of the loop by using **break** and **continue**. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin a new iteration. This program shows examples of **break** and **continue** within **for** and **while** loops:

```
//BreakAndContinue.java
// Demonstrates break and continue keywords
public class BreakAndContinue
 {
  public static void main(String[] args)
   {
    for(int i = 0; i < 100; i++)
     {
      if(i == 58) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.println(i);
     }
    int i = 0;
    // An "infinite loop":
    while(true)
     {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.println(i);
     }
   }
 }
```

In the **for** loop the value of **i** never gets to 100 because the **break** statement breaks out of the loop when **i** is 58. Normally, you'd use a **break** like this only if you didn't know when the terminating condition was going to occur. The **continue** statement causes execution to go back to the top of the iteration loop (thus incrementing **i**) whenever **i** is not evenly divisible by 9. When it is, the value is printed.

The second portion shows an "infinite loop" that would, in theory, continue forever. However, inside the loop there is a **break** statement that will break out of the loop. In addition, you'll see that the **continue** moves back to the top of the loop without completing the remainder. (Thus printing happens only when the value of **i** is divisible by 9.)

The output is:

```
0
9
18
```

```
27
36
45
54
10
20
30
40
```

The value 0 is printed because 0 % 9 produces 0.

A second form of the infinite loop is **for(;;)**. The compiler treats both **while(true)** and **for(;;)** in the same way so whichever one you use is a matter of programming taste.

4.9 goto

The **goto** keyword has been present in programming languages from the beginning. Indeed, **goto** was the genesis of program control in assembly language: "if condition A, then jump here, otherwise jump there." If you read the assembly code that is ultimately generated by virtually any compiler, you'll see that program control contains many jumps. However, **goto** jumps at the source-code level and that's what brought it into disrepute. If a program will always jump from one point to another, isn't there some way to reorganize the code so the flow of control is not so jumpy?

goto fell into true disfavor with the publication of the famous "Goto considered harmful" paper by Edsger Dijkstra, and since then goto-bashing has been a popular sport, with advocates of the cast-out keyword scurrying for cover. As is typical in situations like this, the middle ground is the most fruitful. The problem is not the use of **goto** but the overuse of **goto** and in rare situations **goto** is the best way to structure control flow. Although **goto** is a reserved word in Java, it is not used in the language; Java has no **goto**. However, it does have something that looks a bit like a jump tied in with the **break** and **continue** keywords. It's not a jump but rather a way to break out of an iteration statement. The reason it's often thrown in with discussions of **goto** is because it uses the same mechanism: a label. A label is an identifier followed by a colon, like this:

```
label1:
```

The *only* place a label is useful in Java is right before an iteration statement. And that means *right* before – it does no good to put any other statement between the label and the iteration. And the sole reason to put a label before iteration is if you're going to nest another iteration or a switch inside it. That's because the **break** and **continue** keywords will normally interrupt only the current loop, but when used with a label they'll interrupt the loops up to where the label exists:

```
label1:
outer-iteration
 {
  inner-iteration
   {
   //...
   break; // 1
   //...
   continue; // 2
   //...
   continue label1; // 3
   //...
   break label1; // 4
  }
 }
```

In case 1, the **break** breaks out of the inner iteration and you end up in the outer iteration. In case 2, the **continue** moves back to the beginning of the inner iteration. But in case 3, the **continue label1** breaks out of the inner iteration *and* the outer iteration, all the way back to **label1**. Then it does in fact continue the iteration, but starting at the outer iteration. In case 4, the **break label1** also breaks all the way out to **label1**, but it does not re-enter the iteration. It actually does break out of both iterations. Here is an example using **for** loops:

```
//LabeledFor.java
// Java's "labeled for loop"
public class LabeledFor
 {
  public static void main(String[] args)
   {
    int i = 0; // i is initialized only once
    outer: // Can't have statements here
    for(; true ;)
     { // infinite loop
      inner: // Can't have statements here
       for(; i < 10; i++)
        {
         prt("i = " + i);
         if(i == 2)
          {
           prt("continue");
           continue;
          }
         if(i == 3)
          {
           prt("break");
           i++; // Otherwise i never
           // gets incremented.
           break;
          }
         if(i == 7)
```

```
          {
           prt("continue outer");
           i++; // Otherwise i never
           // gets incremented.
           continue outer;
          }
         if(i == 8)
          {
           prt("break outer");
           break outer;
          }
         for(int k = 0; k < 5; k++)
          {
           if(k == 3)
            {
             prt("continue inner");
             continue inner;
            }
          }
        }
     }
// Can't break or continue
// to labels here
  }
  static void prt(String s)
   {
    System.out.println(s);
   }
 }
```

The above program uses the **prt( )** method that has been defined at the end of the program to print a string. Note that **break** breaks out of the **for** loop and that the increment- expression doesn't occur until the end of the pass through the **for** loop. Since **break** skips the increment expression, the increment is performed directly in the case of **i == 3**. The **continue outer** statement in the case of **i == 7** also goes to the top of the loop and also skips the increment, so it too is incremented directly.

The output is:

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
```

```
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
```

If not for the **break outer** statement, there would be no way to get out of the outer loop from within an inner loop, since **break** by itself can break out of only the innermost loop. (The same is true for **continue.**) Of course, in the cases where breaking out of a loop will also exit the method, you can simply use a **return**. Here is a demonstration of labeled **break** and **continue** statements with **while** loops:

```
//LabeledWhile.java
// Java's "labeled while" loop
public class LabeledWhile
{
public static void main(String[] args)
 {
  int i = 0;
  outer:
  while(true)
   {
    prt("Outer while loop");
    while(true)
     {
      i++;
      prt("i = " + i);
      if(i == 1)
       {
        prt("continue");
        continue;
       }
      if(i == 3)
       {
        prt("continue outer");
        continue outer;
       }
      if(i == 5)
       {
        prt("break");
        break;
       }
      if(i == 7)
       {
        prt("break outer");
        break outer;
       }
     }
   }
 }
static void prt(String s)
 {
  System.out.println(s);
 }
}
```

The same rules hold true for **while**:

1. A plain **continue** goes to the top of the innermost loop and continues.
2. A labeled **continue** goes to the label and re-enters the loop right after that label.
3. A **break** "drops out of the bottom" of the loop.
4. A labeled **break** drops out of the bottom of the end of the loop denoted by the label.

The output of this method makes it clear:

```
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
```

It's important to remember that the *only* reason to use labels in Java is when you have nested loops and you want to **break** or **continue** through more than one nested level. In Dijkstra's "goto considered harmful" paper, what he specifically objected to was the labels, not the goto. He observed that the number of bugs seems to increase with the number of labels in a program. Labels and gotos make programs difficult to analyze statically, since it introduces cycles in the program execution graph. Note that Java labels don't suffer from this problem, since they are constrained in their placement and can't be used to transfer control in an ad hoc manner. It's also interesting to note that this is a case where a language feature is made more useful by restricting the power of the statement.

4.10 switch

The **switch** is sometimes classified as a *selection statement*. The **switch** statement selects from among pieces of code based on the value of an integral expression. Its form is:

```
switch(integral-selector)
 {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
  // ...
  default: statement;
 }
```

Integral-selector is an expression that produces an integral value. The **switch** compares the result of *integral-selector* to each *integral-value*. If it finds a match, the corresponding *statement* (simple or compound) executes. If no match occurs, the **default** *statement* executes. You will notice in the above definition that each **case** ends with a **break**, which causes execution to jump to the end of the **switch** body. This is the conventional way to build a **switch** statement, but the **break** is optional. If it is missing, the code for the following case statements execute until a **break** is encountered. Although you don't usually want this kind of behavior, it can be useful to an experienced programmer. Note the last statement, for the **default**, doesn't have a **break** because the execution just falls through to where the **break** would have taken it anyway. You could put a **break** at the end of the **default** statement with no harm if you considered it important for style's sake. The **switch** statement is a clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to an integral value such as **int** or **char**. If you want to use, for example, a string or a floating-point number as a selector, it won't work in a **switch** statement. For non-integral types, you must use a series of **if** statements. Here's an example that creates letters randomly and determines whether they're vowels or consonants:

```
//VowelsAndConsonants.java
// Demonstrates the switch statement
public class VowelsAndConsonants
 {
  public static void main(String[] args)
   {
```

```
   for(int i = 0; i < 100; i++)
    {
     char c = (char)(Math.random() * 26 + 'a');
     System.out.print(c + ": ");
     switch(c)
      {
       case 'a':
       case 'e':
       case 'i':
       case 'o':
       case 'u':
        System.out.println("vowel");
       break;
        case 'y':
        case 'w':
         System.out.println("Sometimes a vowel");
        break;
        default:
         System.out.println("consonant");
      }
    }
  }
}
```

Since **Math.random( )** generates a value between 0 and 1, you need only multiply it by the upper bound of the range of numbers you want to produce (26 for the letters in the alphabet) and add an offset to establish the lower bound. Although it appears you're switching on a character here, the **switch** statement is actually using the integral value of the character. The singly-quoted characters in the **case** statements also produce integral values that are used for comparison. Notice how the **case**s can be "stacked" on top of each other to provide multiple matches for a particular piece of code. You should also be aware that it's essential to put the **break** statement at the end of a particular case, otherwise control will simply drop through and continue processing on the next case.

Calculation details

The statement:

char c = (char)(Math.random() * 26 + 'a');

deserves a closer look. **Math.random( )** produces a **double**, so the value 26 is converted to a **double** to perform the multiplication, which also produces a **double**. This means that **'a'** must be converted to a **double** to perform the addition. The **double** result is turned back into a **char** with a cast. First, what does the cast to **char** do? That is, if you have the value 29.7 and you cast it to a **char**, is the resulting value 30 or 29? The answer to this can be seen in this example:

//CastingNumbers.java

```
// What happens when you cast a float or double
// to an integral value?
public class CastingNumbers
 {
  public static void main(String[] args)
   {
    double above = 0.7,below = 0.4;
    System.out.println("above: " + above);
    System.out.println("below: " + below);
    System.out.println("(int)above: " + (int)above);
    System.out.println("(int)below: " + (int)below);
    System.out.println("(char)('a' + above): " +(char)('a' + above));
    System.out.println("(char)('a' + below): " +(char)('a' + below));
   }
 }
```

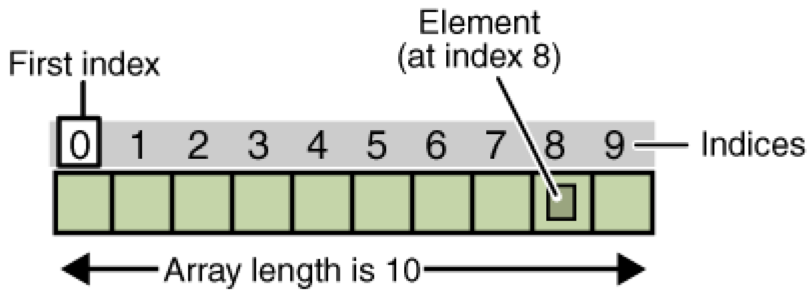
The output is:

```
above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

So the answer is that casting from a **float** or **double** to an integral value always truncates. The second question has to do with **Math.random( )**. Does it produce a value from zero to one, inclusive or exclusive of the value '1'? In math lingo, is it (0,1), or [0,1],or (0,1] or [0,1)? (The square bracket means "includes" whereas the parenthesis means "doesn't include.") The answer is that 0.0 *is* included in the output of **Math.random( )** and 1.0 is not. Or, in math lingo, it is [0,1).

4.11 Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. You've seen an example of arrays already, in the main method of the "Hello World!" application. This section discusses arrays in greater detail.



An array of ten elements

Each item in an array is called an *element* and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8. The following program, ArrayDemo, creates an array of integers, puts some values in it, and prints each value to standard output.

```
class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray;              // declares an array of integers
        anArray = new int[10];      // allocates memory for 10 integers
        anArray[0] = 100; // initialize first element
        anArray[1] = 200; // initialize second element
        anArray[2] = 300; // etc.
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;
        System.out.println("Element at index 0: " + anArray[0]);
        System.out.println("Element at index 1: " + anArray[1]);
        System.out.println("Element at index 2: " + anArray[2]);
        System.out.println("Element at index 3: " + anArray[3]);
        System.out.println("Element at index 4: " + anArray[4]);
        System.out.println("Element at index 5: " + anArray[5]);
        System.out.println("Element at index 6: " + anArray[6]);
        System.out.println("Element at index 7: " + anArray[7]);
        System.out.println("Element at index 8: " + anArray[8]);
        System.out.println("Element at index 9: " + anArray[9]);
    }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

In a real-world programming situation, you'd probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as shown above. However, this example clearly illustrates the array syntax. You'll learn about the various looping constructs (for, while and do-while) in the Control Flow section.

Declaring a Variable to Refer to an Array

The above program declares anArray with the following line of code:

```
int[] anArray;          // declares an array of integers
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as *type*[], where *type* is the data type of the contained elements; the square brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the naming section. As with variables of other types, the declaration does not actually create an array — it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the square brackets after the array's name:

```
float anArrayOfFloats[]; // this form is discouraged
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

Creating, Initializing, and Accessing an Array

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable.

```
anArray = new int[10];  // create an array of integers
```

If this statement were missing, the compiler would print an error like the following, and compilation would fail:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
```

```
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
```

Here the length of the array is determined by the number of values provided between *{* and *}*.

You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of square brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

```
class MultiDimArrayDemo {
   public static void main(String[] args) {
      String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                        {"Smith", "Jones"}};
      System.out.println(names[0][0] + names[1][0]); //Mr. Smith
      System.out.println(names[0][2] + names[1][1]); //Ms. Jones
   }
}
```

The output from this program is:

```
   Mr. Smith
   Ms. Jones
```

Finally, you can use the built-in length property to determine the size of any array. The code System.out.println(anArray.length); will print the array's size to standard output.

Copying Arrays

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

The two Object arguments specify the array to copy *from* and the array to copy *to*. The three int arguments specify the starting position in the source array, the starting position in the destination array and the number of array elements to copy.

The following program, ArrayCopyDemo, declares an array of char elements, spelling the word "decaffeinated". It uses arraycopy to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                                    'i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];
      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
   }
}
```

The output from this program is : caffein

4.12 Summary

Java uses all of C’s execution control statements. All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. Java doesn’t allow you to use a number as a **Booleank**. You have to use the Boolean values – true and false. The if-else statement is used to take a decision based on whether the given condition is true or not. The **return** keyword has two purposes: it specifies what value a method will return (if it doesn’t have a **void** return value) and it causes that value to be returned immediately. **while**, **do-while** and **for** control looping and are sometimes classified as *iteration statements*. A *statement* repeats until the controlling *Boolean-expression* evaluates to false. Inside the body of any of the iteration statements you can also control the flow of the loop by using **break** and **continue**. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin a new iteration. Java has no **goto**. However, it does have something that looks a bit like a jump tied in with the **break** and **continue** keywords. The **switch** statement selects from among pieces of code based on the value of an integral expression.

4.13 Short Answer Type Questions

1. Differentiate between while and do while?
2. What is return statement used for?
3. What do you mean by iteration?

4.14 Long Answer Type Questions

1. Write a program to calculate grade of a student based on the marks entered by the user in five subjects.
2. Explain with the help of an example how to use a label with break and continue in place of goto in java.
3. WAP to compare two numbers.
4. WAP to determine if year is leap year or not.
5. WAP to show the use of Switch statement.

4.15 Suggested Readings

- The Complete Reference by Herbert Scheild
- Programming with Java by E.Balagurusamy
- Java : A Beginner's Guide by Herbert Schildt
- Introduction to Java Programming by Y.Daniel Liang.
- Object Oriented Programming in Java by G.T. Thampi
- Java Programming by C. Xavier

Lesson No. 5

Author : Kanwal Preet Singh
Converted into SLM by: Dr. Vishal Singh

Introduction to Classes

5.1 Objectives

After reading this lesson you will be able to understand:

- Classes
- Objects
- Object References
- Member Variables
- Methods
- Passing arguments to methods

5.2 Introduction

In classic, procedural programming you try to make the real world problem you're attempting to solve fit a few, predetermined data types: integers, floats, Strings and arrays perhaps. In object oriented programming you create a model for a real world system. Classes are programmer-defined types that model the parts of the system. A *class* is a programmer defined type that serves as a blueprint for instances of the class. You can still have ints, floats, Strings, and arrays; but you can also have cars, motorcycles, people, buildings, clouds, dogs, angles, students, courses, bank accounts and any other type that's important to your problem.

Classes specify the data and behavior possessed both by themselves and by the objects built from them. A class has two parts: the fields and the methods. Fields describe what the class is. Methods describe what the class does. Using the blueprint provided by a class, you can create any number of objects, each of which is called an ***instance* of** the class. Different objects of the same class have the same fields and methods, but the values of the fields will in general differ. For example, all people have eye color; but the color of each person's eyes can be different from others. On the other hand, objects have the same methods as all other objects in the class except in so far as the methods depend on the value of the fields and arguments to the method. This dichotomy is reflected in the runtime form of objects. Every object has a separate block of memory to store its fields, but the bytes in the methods are shared between all objects in a class.

Following the principles of Object Oriented Programming (OOP), everything in Java is either a class, a part of a class or describes how a class behaves. Objects are the physical instantiations of classes. They are living entities within a program that have independent lifecycles and that are created according to the class that describes them. Just as many buildings can be built from one blueprint, many objects can be instantiated from one class. Many objects of different classes can be created, used, and destroyed in the course of executing a program. Programming languages provide a number of simple data types like int, float and String. However very often the data you want to work with may not be simple ints, floats or Strings. Classes let programmers define their own more complicated data types.

All the action in Java programs takes place inside class blocks. In Java almost everything of interest is either a class itself or belongs to a class. Methods are defined inside the classes they belong to. Even basic data primitives like integers often need to be incorporated into classes before you can do many useful things with them. The class is the fundamental unit of Java programs.

5.3 Introduction to Java Classes

A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object.

Methods are nothing but members of a class that provide a service for an object or perform some business logic. Java fields and member functions names are case sensitive. Current states of a class's corresponding object are stored in the object's instance variables. Methods define the operations that can be performed in java programming. A class has the following general syntax:

```
//Contents of SomeClassName.java
[ public ] [ ( abstract | final ) ] class SomeClassName [ extends SomeParentClass ] [
implements SomeInterfaces ]
{
    // variables and methods are declared within the curly braces
}
```

- A class can have public or default (no modifier) visibility. The ***visibility*** indicates scope or accessibility from other objects. ***public*** means visible everywhere. The default (ie. omitted) is ***package*** friendly or visible within the current package only.
- The second optional group indicates the capability of a class to be **inherited** or extended by other classes. It can be either abstract, final or concrete (no modifier). ***abstract*** classes must be extended and ***final*** classes can never be extended by inheritance. The default (ie. omitted) indicates that the class may or may not be extended at the programmers discretion.
- It must have the class keyword and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces '{}'.
- Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

Here is an example of a Horse class. Horse is a subclass of Mammal, and it implements the Hoofed interface.

```
public class Horse extends Mammal implements Hoofed
{
        //Horse's variables and methods go here
}
```

Example : The Car Class

Suppose you need to write a traffic simulation program that watches cars going past an intersection. Each car has a speed, a maximum speed and a license plate that uniquely identifies it. In traditional programming languages you'd have two floating point and one string variable for each car. With a class you combine these into one thing like this.

```
class Car
 {
  String licensePlate; // e.g. "PB11R2300"
  double speed;        // in kilometers per hour
  double maxSpeed;     // in kilometers per hour
 }
```

These variables (licensePlate, speed and maxSpeed) are called the *member variables*, i*nstance variables* or *fields* of the class. Fields tell you what a class is and what its properties are.

An ***object*** is a specific instance of a class with particular values for the fields. While a class is a general blueprint for objects, an instance is a particular object.

5.4 Constructing objects with new

To instantiate an object in Java, use the keyword new followed by a call to the class's constructor. First declare an instance variable of the class as follows:
Class_name instance_variable;

Then create an object of the class using new operator and assign it to the instance variable declared above as:
instance_variable = new class_name();
For example Here's how you'd create a new Car variable called c and the assign to it an object of Class **Car**:

```
Car c;
c = new Car();
```

The first word, Car, declares the type of the variable c. Classes are types and variables of a class type need to be declared just like variables that are ints or doubles. The equals sign is the assignment operator and new is the construction operator. Finally notice the Car() method. The parentheses tell you this is a method and not a data type like the Car on the left hand side of the assignment. This is a constructor, a method that creates a new instance of a class. You'll learn more about constructors shortly. However if you do nothing, then the compiler inserts a default constructor that takes no arguments. This is often condensed into one line like this:

```
Car c = new Car();
```

5.5 Object References

In Java, a class is a type, similar to the built-in types such as int and boolean. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter or the return type of a function. For example, a program could define a variable named std of type Student with the statement

```
Student std;
```

However, declaring a variable does not create an object! This is an important point, which is related to this Very Important Fact:

In Java, no variable can ever hold an object. A variable can only hold a reference to an object.

You should think of objects as floating around independently in the computer's memory. In fact, there is a special portion of memory called the heap where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a reference or pointer to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object. In a program, objects are created using an operator called new, which creates an object and returns a reference to that object. For example, assuming that std is a variable of type Student, declared as above, the assignment statement:

```
std = new Student();
```

would create a new object which is an instance of the class Student, and it would store a reference to that object in the variable std. The value of the variable is a reference to the object, not the object itself. It is not quite true, then, to say that the object is the "value of the variable std" (though sometimes it is hard to avoid using this terminology). It is certainly not at all true to say that the object is "stored in the variable std." The proper terminology is that "the variable std refers to the object," and we will try to stick to that terminology as much as possible. So, suppose that the variable std refers to an object belonging to the class Student. That object has instance variables name, test1, test2 and test3. These instance variables can be referred to as std.name, std.test1, std.test2 and std.test3. This follows the usual naming convention that when B is part of A, then the full name of B is A.B. For example, a program might include the lines

```
System.out.println("Hello, " + std.name + ". Your test grades are:");
System.out.println(std.test1);
System.out.println(std.test2);
System.out.println(std.test3);
```

This would output the name and test grades from the object to which std refers. Similarly, std can be used to call the getAverage() instance method in the object by saying std.getAverage(). To print out the student's average, you could say:

```
System.out.println( "Your average is " + std.getAverage() );
```

More generally, you could use std.name any place where a variable of type String is legal. You can use it in expressions. You can assign a value to it. You can even use it to call subroutines from the String class. For example, std.name.length() is the number of characters in the student's name. It is possible for a variable like std, whose type is given by a class, to refer to no object at all. We say in this case that std holds a null reference. The null reference is written in Java as "null". You can store a null reference in the variable std by saying

```
std = null;
```

and you could test whether the value of std is null by testing

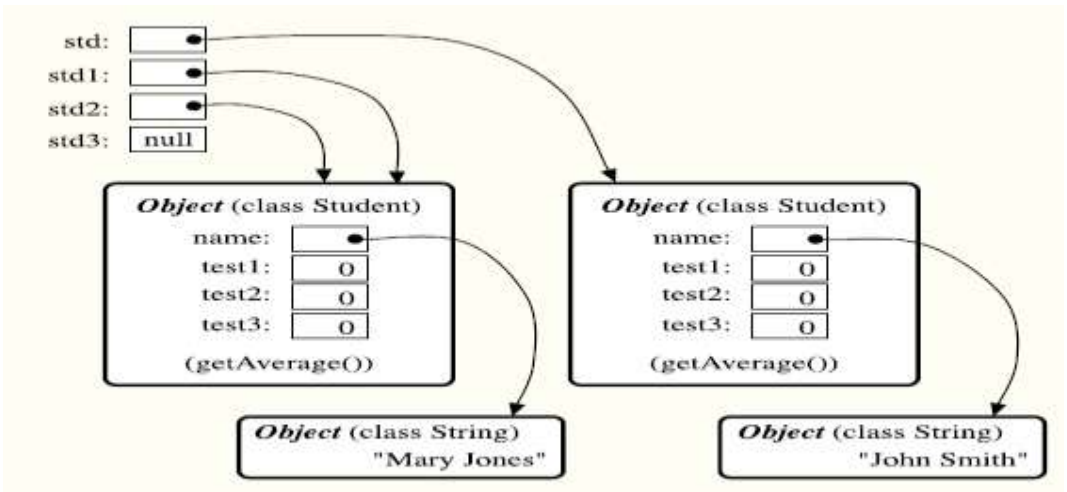
```
if (std == null) . . .
```

If the value of a variable is null, then it is, of course, illegal to refer to instance variables or instance methods through that variable—since there is no object, and hence no instance variables to refer to. For example, if the value of the variable std is null, then it would be illegal to refer to std.test1. If your program attempts to use a null reference illegally like this, the result is an error called a null pointer exception. Let's look at a sequence of statements that work with objects:

```
Student std, std1, // Declare four variables of
std2, std3; // type Student.
std = new Student(); // Create a new object belonging
// to the class Student, and
// store a reference to that
// object in the variable std.
std1 = new Student(); // Create a second Student object
// and store a reference to
// it in the variable std1.
```

```
std2 = std1; // Copy the reference value in std1
// into the variable std2.
std3 = null; // Store a null reference in the
// variable std3.
std.name = "John Smith"; // Set values of some instance variables.
std1.name = "Mary Jones";
// (Other instance variables have default
// initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:



This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. The variable std3, with a value of null, doesn't point anywhere. The arrows from std1 and std2 both point to the same object. This illustrates a Very Important Point:

When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.

When the assignment "std2 = std1;" was executed, no new object was created. Instead, std2 was set to refer to the very same object that std1 refers to. This has some consequences that might be surprising. For example, std1.name and std2.name are two different names for the same variable, namely the instance variable in the object that both std1 and std2 refer to. After the string "Mary Jones" is assigned to the variable std1.name, it is also be true that the value of std2.name is "Mary Jones". There is a potential for a lot of confusion here, but you can help protect yourself from it if you keep telling yourself, "The object is not in the variable. The variable just holds a pointer to the object." You can test objects for equality and inequality using the operators == and !=, but here again, the semantics are different from what you are used to. When you make a test "if (std1 == std2)", you are testing whether the values stored in std1 and std2 are the same. But the values are references to objects, not objects. So, you are testing whether std1 and std2 refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether

"std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3 == std2.test3 && std1.name.equals(std2.name)".

5.6 The Member Access Separator

Once you've constructed a car, you want to do something with it. To access the fields of the car you use the . separator. The Car class has three fields

- licensePlate
- speed
- maxSpeed

Therefore if c is a Car object, c has three fields as well:

- c.licensePlate
- c.speed
- c.maxSpeed

You use these just like you'd use any other variables of the same type. For instance:

```
Car c = new Car();
c.licensePlate = " PB11R2300";
c.speed = 64.0;
c.maxSpeed = 130.00;
System.out.println(c.licensePlate + " is moving at " + c.speed +
"kilometers per hour.");
```

The . separator selects a specific member of a Car object by name.

Using a Car object in a different class

The next program creates a new car, sets its fields, and prints the result:

```
class CarTest
 {
  public static void main(String args[])
   {
    Car c = new Car();
    c.licensePlate = " PB11R2300";
    c.speed = 64.0;
    c.maxSpeed = 130.00;
    System.out.println(c.licensePlate + " is moving at " + c.speed +
    " kilometers per hour.");
   }
 }
```

This program requires not just the CarTest class but also the Car class. To make them work together put the Car class in a file called Car.java. Put the CarTest class in a file called CarTest.java. Put both these files in the same directory. Then compile both files in the usual way. Finally run CarTest. For example,

```
% javac Car.java
% javac CarTest.java
% java CarTest
PB11R2300 is moving at 64.0 kilometers per hour.
```

Note that Car does not have a main() method so you cannot run it. It can exist only when called by other programs that do have main() methods. Many of the applications you write from now on will use multiple classes. It is customary in Java to put every class in its own file. Later you'll learn how to use packages to organize your commonly used classes in different directories. For now keep all your .java source code and .class byte code files in one directory.

Initializing Fields

Fields can (and often should) be initialized when they're declared, just like local variables.

```
class Car
 {
  String licensePlate = "";    // e.g. "PB11R2300"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed     = 130.0; // in kilometers per hour
}
```

The next program creates a new car and prints it:

```
class CarTest2
 {
  public static void main(String[] args)
   {
     Car c = new Car();
     System.out.println(c.licensePlate + " is moving at " + c.speed +
     "kilometers per hour.");
   }
 }
```

For example,

```
$ javac Car.java
$ javac CarTest2.java
$ java CarTest
```

Output is:

```
is moving at 0.0 kilometers per hour.
```

5.7 Methods

Data types aren't much use unless you can do things with them. For this purpose classes have methods. Fields say what a class *is*. Methods say what a class *does*. The fields and methods of a class are collectively referred to as the *members of the class*.

The classes you've encountered up till now have mostly had a single method, main(). However, in general classes can have many different methods that do many different things. For instance the Car class might have a method to make the car go as fast as it can. For example,

```
class Car
```

```
{
  String licensePlate = "";    // e.g. "PB11R2300"
  double speed        = 0.0;   // in kilometers per hour
  double maxSpeed     = 130.0; // in kilometers per hour
  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt()
   {
    this.speed = this.maxSpeed;
   }
}
```

The fields are the same as before, but now there's also a method called floorIt(). It begins with the Java keyword void which is the return type of the method. Every method must have a return type which will either be void or some data type like int, byte, float or String. The return type says what kind of the value will be sent back to the calling method when all calculations inside the method are finished. If the return type is int, for example, you can use the method anywhere you use an int constant. If the return type is void then no value will be returned.

floorIt is the name of this method. The name is followed by two empty parentheses. Any arguments passed to the method would be passed between the parentheses, but this method has no arguments. Finally an opening brace ({) begins the body of the method. There is one statement inside the method

this.speed = this.maxSpeed;

Notice that within the Car class the field names are prefixed with the keyword this to indicate that I'm referring to fields in the current object.

Finally the floorIt() method is closed with a } and the class is closed with another }.

Invoking Methods

Outside the Car class, you call the floorIt() method just like you reference fields, using the name of the object you want to accelerate to maximum and the . separator as demonstrated below

```
class CarTest3
 {
  public static void main(String args[])
   {
    Car c = new Car();
    c.licensePlate = " PB11R2300";
    c.maxSpeed = 130.0;
    System.out.println(c.licensePlate + " is moving at " + c.speed +
    " kilometers per hour.");
   c.floorIt();
   System.out.println(c.licensePlate + " is moving at " + c.speed +
    " kilometers per hour.");
  }
 }
```

The output is:

```
PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
```

The floorIt() method is completely enclosed within the Car class. Every method in a Java program must belong to a class. Unlike C++ programs, Java programs cannot have a method hanging around in global space that does everything you forgot to do inside your classes.

Implied this

```
class Car
 {
  String licensePlate = "";     // e.g. "PB11R2300"
  double speed        = 0.0;    // in kilometers per hour
  double maxSpeed     = 130.0; // in kilometers per hour
  void floorIt()
   {
     speed = maxSpeed;
   }
 }
```

Within the Car class, you don't absolutely need to prefix the field names with this. like this.licensePlate or this.speed. Just licensePlate and speed are sufficient. The this. may be implied. That's because the floorIt() method must be called by a specific instance of the Car class, and this instance knows what its data is. Or another way of looking at it, the every object has its own floorIt() method.

For clarity, we will use an explicit this, and I recommend you do so too, at least initially. As you become more comfortable with Java, classes, references, and OOP, you will be able to leave out the this without fear of confusion. Most real-world code does not use an explicit this.

5.8 Member Variables vs. Local Variables

```
class Car
  {
   String licensePlate = "";     // member variable
   double speed;       = 0.0;    // member variable
   double maxSpeed;    = 130.0; // member variable
   boolean isSpeeding()
    {
     double excess;    // local variable
     excess = this.maxSpeed - this.speed;
     if (excess < 0) return true;
     else return false;
    }
}
```

Until now all the programs you've seen were quite simple in structure. Each had exactly one class. This class had a single method, main(), which contained all the program logic and variables. The variables in those classes were all local to the main() method. They could not be accessed by anything outside the main() method. These are called **local variables.**

This sort of program is the amoeba of Java. Everything the program needs to live is contained inside a single cell. It's quite an efficient arrangement for small organisms, but it breaks down when you want to design something bigger or more complex.

The licensePlate, speed and maxSpeed variables of the Car class, however, belong to a Car object, not to any individual method. They are defined outside of any methods but inside the class and are used in different methods. They are called **member variables or fields**.

Member variable, instance variable, and field are different words that mean the same thing. *Field* is the preferred term in Java. Member variable is the preferred term in C++. A member is not the same as a member variable or field. Members include both fields and methods.

5.9 Passing Arguments to Methods

It's generally considered bad form to access fields directly. Instead it is considered good object oriented practice to access the fields only through methods. This allows you to change the implementation of a class without changing its interface. This also allows you to enforce constraints on the values of the fields.

To do this you need to be able to send information into the Car class. This is done by passing arguments. For example, to allow other objects to change the value of the speed field in a Car object, the Car class could provide an accelerate() method. This method does not allow the car to exceed its maximum speed, or to go slower than 0 kph.

```
void accelerate(double sp)
 {
   this.speed = this.speed + sp;
   if (this.speed > this.maxSpeed)
    {
      this.speed = this.maxSpeed;
    }
   if (this.speed <  0.0)
    {
      this.speed = 0.0;
    }
  }
```

The first line of the method is called its *signature*. The signature

void accelerate(double sp)

indicates that accelerate() returns no value and takes a single argument, a double which will be referred to as sp inside the method. sp is a purely *formal* argument. Java passes method arguments by value, not by reference.

Passing Arguments to Methods, An Example

```
class Car
 {
  String licensePlate = "";     // e.g. "PB11R2300"
  double speed        = 0.0;    // in kilometers per hour
  double maxSpeed     = 130.0; // in kilometers per hour

  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt()
   {
     this.speed = this.maxSpeed;
   }
  void accelerate(double sp)
   {
     this.speed = this.speed + sp;
     if (this.speed > this.maxSpeed)
      {
        this.speed = this.maxSpeed;
      }
     if (this.speed <  0.0)
      {
        this.speed = 0.0;
      }
   }
 }
class CarTest4
  {
    public static void main(String[] args)
     {
      Car c = new Car();
      c.licensePlate = " PB11R2300";
      c.maxSpeed = 130.0;
      System.out.println(c.licensePlate + " is moving at " + c.speed +
      " kilometers per hour.");
      for (int i = 0; i < 15; i++)
       {
         c.accelerate(10.0);
         System.out.println(c.licensePlate + " is moving at " + c.speed +
         " kilometers per hour.");
       }
     }
   }
```

Here's the output:

```
PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.
PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
```

5.10 Setter Methods

Setter methods, also known as mutator methods, merely set the value of a field to a value specified by the argument to the method. These methods almost always return void. One common idiom in setter methods is to use this.*name* to refer to the field and give the argument the same name as the field. For example,

```
class Car
 {
  String licensePlate; // e.g. "PB11R2300"
  double speed;        // kilometers per hour
  double maxSpeed;     // kilometers per hour
  // setter method for the license plate property
  void setLicensePlate(String licensePlate)
   {
     this.licensePlate = licensePlate;
   }
  // setter method for the maxSpeed property
  void setMaximumSpeed(double maxSpeed)
   {
    if (maxSpeed > 0)
      this.maxSpeed = maxSpeed;
    else
```

```
      this.maxSpeed = 0.0;
   }
  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt()
   {
     this.speed = this.maxSpeed;
   }
  void accelerate(double sp)
   {
     this.speed = this.speed + sp;
     if (this.speed > this.maxSpeed)
      {
        this.speed = this.maxSpeed;
      }
     if (this.speed <  0.0)
      {
        this.speed = 0.0;
       }
    }
  }
//Using Setter Methods, An Example
class CarTest5
 {
  public static void main(String args[])
   {
    Car c = new Car();
    c.setLicensePlate("New York A45 636");
    c.setMaximumSpeed(130.0);
    System.out.println(c.licensePlate + " is moving at " + c.speed +
    " kilometers per hour.");
    for (int i = 0; i < 15; i++)
     {
       c.accelerate(10.0);
       System.out.println(c.licensePlate + " is moving at " + c.speed +
       " kilometers per hour.");
      }
   }
  }
```
Here's the output:

```
PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
```

```
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.
PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
```

5.11 Returning Values From Methods

It's often useful to have a method return a value to the class that called it. This is accomplished by the return keyword at the end of a method and by declaring the data type that is returned by the method at the beginning of the method. For example, the following getLicensePlate() method returns the current value of the licensePlate field in the Car class.

```
 String getLicensePlate()
  {
    return this.licensePlate;
  }
```

A method like this that merely returns the value of an object's field or property is called a *getter* or *accessor* method. The signature String getLicensePlate() indicates that getLicensePlate() returns a value of type String and takes no arguments. Inside the method the line

return this.licensePlate;

returns the String contained in the licensePlate field to whoever called this method. It is important that the type of value returned by the return statement match the type declared in the method signature. If it does not, the compiler will complain.

Using Getter Methods, An Example

```
class CarTest6
  {
    public static void main(String args[])
     {
     Car c = new Car();
       c.setLicensePlate("New York A45 636");
       c.setMaximumSpeed(130.0);
          System.out.println(c.getLicensePlate() + " is moving at "
       + c.getSpeed() + " kilometers per hour.");
       for (int i = 0; i < 15; i++)
        {
         c.accelerate(10.0);
         System.out.println(c.getLicensePlate() + " is moving at "
         + c.getSpeed() + " kilometers per hour.");
        }

     }
  }
```

There's no longer any direct access to fields. Here's the output:

```
PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.
PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2330is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
```

5.12 Summary

A class is a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an object. To instantiate an object in Java, use the keyword new followed by a call to the class's constructor. In Java, a class is a type, similar to the built-in types such as int and boolean. In Java, no variable can ever hold an object. A variable can only hold a reference to an object. To access the fields of the class you use the . separator. Data types aren't much use unless you can do things with them. For this purpose classes have methods. Fields say what a class *is*. Methods say what a class *does*. The fields and methods of a class are collectively referred to as the *members of the class*. It's generally considered bad form to access fields directly. Instead it is considered good object oriented practice to access the fields only through methods. This allows you to change the implementation of a class without changing its interface. This also allows you to enforce constraints on the values of the fields.

5.13 Some Practice Questions

1. What is the relation between a class and an object?
2. What do you mean by object reference?
3. What are member variables and how are they accessed?

5.14 Suggested Readings

- The Complete Reference by Herbert Scheild, Mc-Graw Hil
- Programming with Java by E.Balagurusamy, Mc-Graw Hill
- Java : A Beginner's Guide by Herbert Scwildt , Mc-Graw Hill
- Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
- Object Oriented Programming in Java by G.T. Thampi
- Java Programming by C. Xavier

Lesson No. 6 **Author : Kanwal Preet Singh**
Converted into SLM by: Dr. Vishal Singh

Constructors

6.1 Objectives

After reading this lesson you will be able to understand:

- Constructors
- Default Constructor
- Parameterized Constructors
- Method Overloading
- Constructor Overloading
- Garbage Collection
- “this” Keyword

6.2 Introduction

In the previous lesson we used setter methods to set the values of the object variables. There is another way to set these values. This is by use of constructors. Constructors are also methods that are used to set the values of object variables, but unlike setter methods they don’t need to be invoked. The constructor methods are invoked themselves when an object of a class is created. In this lesson, we will also see how to define different methods with the same name using the concept of method overloading.

6.3 Constructors

A constructor creates a new instance of the class. It initializes all the variables and does any work necessary to prepare the class to be used. In the line

Car c = new Car();

Car() is the constructor. A constructor has the same name as the class. If no constructor exists Java provides a generic one that takes no arguments, but it's better to write your own. You make a constructor by writing a method that has the same name as the class. Thus the Car constructor is called Car(). Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit. The following method is a constructor that initializes license plate to an empty string, speed to zero and maximum speed to 120.0.

```
 Car()
 {
   licensePlate = "";
   speed  = 0.0;
   maxSpeed = 120.0;
  }
```

We can rewrite the program written in previous lesson to include a constructor in the following way:

```
class Car
 {
  String licensePlate;     // e.g. "PB11R2300"
  double speed;    // in kilometers per hour
  double maxSpeed; // in kilometers per hour
  public Car() //Constructor
   {
    licensePlate = "";
    speed  = 0.0;
    maxSpeed = 120.0;
   }
  // accelerate to maximum speed
  // put the pedal to the metal
  void floorIt()
   {
    this.speed = this.maxSpeed;
   }
  void accelerate(double sp)
   {
    this.speed = this.speed + sp;
    if (this.speed > this.maxSpeed)
     {
      this.speed = this.maxSpeed;
     }
    if (this.speed <  0.0)
     {
      this.speed = 0.0;
     }
   }
 }
class CarTest4
  {
    public static void main(String[] args)
     {
      Car c = new Car();
      c.licensePlate = " PB11R2300";
      c.maxSpeed = 130.0;
      System.out.println(c.licensePlate + " is moving at " + c.speed +
      " kilometers per hour.");
      for (int i = 0; i < 15; i++)
       {
         c.accelerate(10.0);
         System.out.println(c.licensePlate + " is moving at " + c.speed +
         " kilometers per hour.");
       }
     }
   }
```

Here's the output:

```
PB11R2300is moving at 0.0 kilometers per hour.
PB11R2300is moving at 10.0 kilometers per hour.
PB11R2300is moving at 20.0 kilometers per hour.
PB11R2300is moving at 30.0 kilometers per hour.
PB11R2300is moving at 40.0 kilometers per hour.
PB11R2300is moving at 50.0 kilometers per hour.
PB11R2300is moving at 60.0 kilometers per hour.
PB11R2300is moving at 70.0 kilometers per hour.
PB11R2300is moving at 80.0 kilometers per hour.
PB11R2300is moving at 90.0 kilometers per hour.
PB11R2300is moving at 100.0 kilometers per hour.
PB11R2300is moving at 110.0 kilometers per hour.
PB11R2300is moving at 120.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
PB11R2300is moving at 130.0 kilometers per hour.
```

The constructors have the following properties:

- Constructors are special methods.
- Constructor methods have the same name as the class itself.
- Constructors look just like methods, but they have no return type, not even void.

- Constructors are invoked only by using the *new* keyword, not the dot notation like methods.

6.4 Default Constructor

If you examine the first Car class code, the Car class did not have any constructors. However, you could still call "new Car()" and get a Car object. What constructor got called? The *Default* constructor is a no-arguments constructor that is provided by Java if you define a class without explicitly defining any constructors. The Default constructor allows you to create objects of classes that have no specifically designed constructors. The default constructor does not exist when the class contains any other constructor. One of the most common mistakes in Java is to rely on a Default constructor that no longer exists. Take, for example, the following class definition:

```
class Car
 {
  String licensePlate;     // e.g. "PB11R2300"
  double speed;    // in kilometers per hour
  double maxSpeed; // in kilometers per hour

  public Car(String licensePlate, double speed, double maxSpeed) //Constructor
   {
     this.licensePlate = licensePlate;
     if(maxSpeed > 0)
        this.maxSpeed = maxSpeed;
     else
        this.maxSpeed = 0.0;
     if (speed > this.maxSpeed)
        this.speed = this.maxSpeed;
     if (speed < 0)
        this.speed = 0.0;
     else
        this.speed = speed;
   }
 }
```

Given this definition of the Car class, what is wrong with the following code?

```
Car wifeCar = new Car("PB11AE2121", 60,130);
Car myCar = new Car();
myCar.licensePlate = PB11R2300;
myCar.speed = 80;
myCar.maxSpeed = 120;
```

Do you see it? Will "new Car()" compile? No, it won't, because there is no longer a constructor that takes no arguments. The Default constructor was there until you provided the additional constructor Car(String, double, double). To fix the above code, you could add the following constructor to your Car class.

```
Car(){}
```

6.5 Parameterized Constructors

As you may have already seen above, we can also pass arguments to the constructors. In fact a constructor without arguments is not of much use as it will initialize all the objects to the same value. Following is an example of a constructor that accepts three arguments:

```
 Car(String licensePlate, double speed, double maxSpeed)
 {
   this.licensePlate = licensePlate;
   if(maxSpeed > 0)
      this.maxSpeed = maxSpeed;
   else
      this.maxSpeed = 0.0;
   if (speed > this.maxSpeed)
      this.speed = this.maxSpeed;
   if (speed < 0)
      this.speed = 0.0;
   else
      this.speed = speed;
  }
```

Or perhaps you always want the initial speed to be zero, but require the maximum speed and license plate to be specified, then you can use a two argument constructor in the following way:

```
Car(String licensePlate, double maxSpeed)
 {
   this.licensePlate = licensePlate;
   this.speed  = 0.0;
   if (maxSpeed > 0)
     this.maxSpeed = maxSpeed;
   else
     this.maxSpeed = 0.0;
  }
```

Here's the complete class:

```
class Car
 {
  String licensePlate; // e.g. "PB11R2300"
  double speed;        // kilometers per hour
  double maxSpeed;     // kilometers per hour

  Car(String licensePlate, double maxSpeed)
 {
   this.licensePlate = licensePlate;
   this.speed  = 0.0;
   if (maxSpeed > 0)
      this.maxSpeed = maxSpeed;
   else
      this.maxSpeed = 0.0;
 }

// getter (accessor) methods
String getLicensePlate()
 {
   return this.licensePlate;
 }

double getMaxSpeed()
 {
   return this.maxSpeed;
 }

double getSpeed()
 {
   return this.speed;
 }
// accelerate to maximum speed
// put the pedal to the metal
void floorIt()
 {
   this.speed = this.maxSpeed;
 }
  void accelerate(double deltaV)
 {
   this.speed = this.speed + deltaV;
   if (this.speed > this.maxSpeed)
    {
      this.speed = this.maxSpeed;
    }
   if (this.speed <  0.0)
    {
      this.speed = 0.0;
    }
 }
```

```
  }
```

Notice that I've taken out several things:

- the initialization of the fields
- the setter methods

The next program uses the constructor to initialize a car rather than setting the fields directly.

```
class CarTest7
 {
  public static void main(String args[])
   {
     Car c = new Car("PB11R2300", 120.5);
     System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() +
     " kilometers per hour.");
     for (int i = 0; i < 15; i++)
      {
        c.accelerate(10.0);
        System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed()
        + " kilometers per hour.");
      }
   }
 }
```

You no longer need to know about the fields licensePlate, speed and maxSpeed. All you need to know is how to construct a new car and how to print it. You may ask whether the setLicensePlate() method is still needed since it's now set in a constructor. The general answer to this question depends on the use to which the Car class is to be put. The specific question is whether a car's license plate may need to be changed after the Car object is created. Some classes may not change after they're created; or, if they do change, they'll represent a different object. The most common such class is String. You cannot change a string's data. You can only create a new String object. Such objects are called *immutable*.

6.6 Constraints

One of the reasons to use constructors and setter methods rather than directly accessing fields is to enforce constraints. For instance, in the Car class it's important to make sure that the speed is always less than or equal to the maximum speed and that both speed and maximum speed are greater than or equal to zero. You've already seen one example of this in the accelerate() method which will not accelerate a car past its maximum speed.

```
  void accelerate(double deltaV)
   {
     this.speed = this.speed + deltaV;
     if (this.speed > this.maxSpeed)
        this.speed = this.maxSpeed;
     if (this.speed < 0.0)
        this.speed = 0.0;
    }
```

You can also insert constraints like that in the constructor. For example, this Car constructor makes sure that the maximum speed is greater than or equal to zero:

```
Car(String licensePlate, double maxSpeed)
 {
  this.licensePlate = licensePlate;
  this.speed  = 0.0;
  if (maxSpeed >= 0.0)
    this.maxSpeed = maxSpeed;
  else
    maxSpeed = 0.0;
 }
```

6.7 Method Overloading

Overloading is when the same method or operator can be used on many different types of data. For instance the + sign is used to add ints as well as concatenate strings. The plus sign behaves differently depending on the type of its arguments. Therefore the plus sign is inherently overloaded. Methods can be overloaded as well. System.out.println() can print a double, a float, an int, a long or a String. You don't do anything different depending on the type of number you want the value of. Overloading takes care of it.

Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. This allows the compiler to match parameters and choose the correct method when a number of choices exist. Changing just the return type is not enough to overload a method, and will be a compile-time error. They must have a different signature. When no method matching the input parameters is found, the compiler attempts to convert the input parameters to types of greater precision. A match may then be found without error. At compile time, the right implementation is chosen based on the signature of the method call. Below is an example of a class demonstrating Method Overloading

```
public class MethodOverloadDemo
 {
   void sumOfParams()
    { // First Version
       System.out.println(”No parameters”);
    }
   void sumOfParams(int a)
    { // Second Version
       System.out.println(”One parameter: ” + a);
    }
    int sumOfParams(int a, int b)
    { // Third Version
       System.out.println(”Two parameters: ” + a + ” , ” + b);
       return a + b;
     }
    double sumOfParams(double a, double b)
     { // Fourth Version
       System.out.println(”Two double parameters: ” + a + ” , ” + b);
       return a + b;
     }
    public static void main(String args[])
     {
       MethodOverloadDemo moDemo = new MethodOverloadDemo();
       int intResult;
       double doubleResult;
       moDemo.sumOfParams();
       System.out.println();
       moDemo.sumOfParams(2);
       System.out.println();
       intResult = moDemo.sumOfParams(10, 20);
       System.out.println(”Sum is  ” + intResult);
       System.out.println();
       doubleResult = moDemo.sumOfParams(1.1, 2.2);
       System.out.println(”Sum is  ” + doubleResult);
       System.out.println();
     }
 }
```

Output is:

```
 No parameters
 One parameter: 2
 Two parameters: 10 , 20
 Sum is 30
 Two double parameters: 1.1 , 2.2
 Sum is 3.3000000000000003
```

6.8 Constructor Overloading

Constructor Overloading is a logical extension of method overloading, A constructor is overloaded when the same constructor with different number and types of arguments initializes an object with valid initial values. In the section above, we saw several different versions of the Car constructor, one that took three arguments and one that took two arguments and one that took no arguments. We can use all of these in a single class, though here I only use two because there really aren't any good default values for licensePlate and maxSpeed. On the other hand, 0 is a perfectly reasonable default value for speed.

```
public class Car
 {
    private String licensePlate; // e.g. " PB11R2300"
    private double speed;        // kilometers per hour
    private double maxSpeed;     // kilometers per hour
    // constructors
   public Car(String licensePlate, double maxSpeed)
     {
      this.licensePlate = licensePlate;
      this.speed = 0.0;
      if (maxSpeed >= 0.0)
        {
         this.maxSpeed = maxSpeed;
        }
      else
        {
         maxSpeed = 0.0;
        }
     }
   public Car(String licensePlate, double speed, double maxSpeed)
     {
       this.licensePlate = licensePlate;
       if (maxSpeed >= 0.0)
         {
          this.maxSpeed = maxSpeed;
         }
    else
         {
          maxSpeed = 0.0;
         }
    if (speed < 0.0)
         {
          speed = 0.0;
         }
    if (speed <= maxSpeed)
         {
          this.speed = speed;
         }
    else
         {
          this.speed = maxSpeed;
         }
    }
  // other methods...
 }
```

The signature of the first constructor in the above program is Car(String, double). The signature of the second constructor is Car(String, double, double). Thus the first version of the Car() constructor is called when there is one String argument followed by one double argument and the second version is used when there is one String argument followed by two double arguments. If there are no arguments to the constructor or two or three arguments that aren't the right type in the right order, then the compiler generates an error because it doesn't have a method whose signature matches the requested method call. For example

```
Error:    Method Car(double) not found in class Car.
```

this in constructors

It is often the case that overloaded methods are essentially the same except that one supplies default values for some of the arguments. In this case, your code will be easier to read and maintain (though perhaps *marginally* slower) if you put all your logic in the method that takes the most arguments and simply invoke that method from all its overloaded variants that merely fill in appropriate default values. This technique should also be used when one method needs to convert from one type to another. For instance one variant can convert a String to an int, then invoke the variant that takes the int as an argument. This is straight-forward for regular methods, but doesn't quite work for constructors because you can't simply write a method like this:

```
  public Car(String licensePlate, double maxSpeed)
   {
     Car(licensePlate, 0.0, maxSpeed);
   }
```

Instead, to invoke another constructor in the same class from a constructor you use the keyword this like so:

```
  public Car(String licensePlate, double maxSpeed)
   {
      this(licensePlate, 0.0, maxSpeed);
   }
```

Must this be the first line of the constructor?
For example,

```
public class Car
 {
   private String licensePlate; // e.g. "PB11R2300"
   private double speed;        // kilometers per hour
   private double maxSpeed;     // kilometers per hour
   // constructors
   public Car(String licensePlate, double maxSpeed)
    {
```

```
    this(licensePlate, 0.0, maxSpeed);
   }
  public Car(String licensePlate, double speed, double maxSpeed)
   {
     this.licensePlate = licensePlate;
     if (maxSpeed >= 0.0)
      {
        this.maxSpeed = maxSpeed;
      }
     else
      {
        maxSpeed = 0.0;
      }
     if (speed < 0.0)
      {
        speed = 0.0;
      }
     if (speed <= maxSpeed)
      {
        this.speed = speed;
      }
     else
      {
        this.speed = maxSpeed;
      }
   }

  // other methods...
}
```

This approach saves several lines of code. In also means that if you later need to change the constraints or other aspects of construction of cars, you only need to modify one method rather than two. This is not only easier; it gives bugs fewer opportunities to be introduced either through inconsistent modification of multiple methods or by changing one method but not others.

6.9 Garbage Collection

So far, this section has been about creating objects. What about destroying them? In Java, the destruction of objects takes place automatically. An object exists in the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this):

```
Student std = new Student("Hardeep");
```

```
std = null;
```

In the first line, a reference to a newly created Student object is stored in the variable std. But in the next line, the value of std is changed and the reference to the Student object is gone. In fact, there are now no references whatsoever to that object stored in any variable. So there is no way for the program ever to use the object again. It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called **garbage collection** to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." In the above example, it was very easy to see that the Student object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programming languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

Because Java uses garbage collection, such errors are simply impossible. Garbage collection is an old idea and has been used in some programming languages since the 1960s. You might wonder why all languages don't use garbage collection. In the past, it was considered too slow and wasteful. However, research into garbage collection techniques combined with the incredible speed of modern computers have combined to make garbage collection feasible. Programmers should rejoice.

6.10 "this" Keyword

All instance methods have automatic access to other instance methods and any data (instance variables) defined for the object. In the example below, the pimpMyRide method calls the getDescription() method and uses Car instance variables.

```
public class Car
  {
   String color;
   String type;
      {
       color="red";
       type="sedan";
      }
  String getDescription()
```

```
    {
      String desc = "This is a " + color + " " + type;
      return desc;
    }
  void pimpMyRide(String newColor, String customized)
    {
     color = newColor;
     type = customized + " " + type;
     System.out.println(getDescription());
    }
  }
```
Within an instance method or a constructor, the keyword *this* is a reference to the current object. In other words, *this* refers to the current instance. You can use *this* to refer to any instance variable or instance method of the object from within an instance method or a constructor. Rewriting the code above should help demonstrate *this*.
```
      public class Car {
        String color;
        String type;
          {
          this.color="red";
          this.type="sedan";
        }
        String getDescription(){
          String desc = "This is a " + this.color + " " + this.type;
          return desc;
        }
        void pimpMyRide(String newColor, String customized) {
          this.color = newColor;
          this.type = customized + " " + this.type;
          System.out.println(this.getDescription());
        }
      }
```
Even though this code works, the use of *this* here doesn't seem to make much sense? In fact, it might just make things more complex. However, consider the pimpMyRide method if it were written as shown below.
```
        void pimpMyRide(String color, String type) {
           color = color;               //??? Which color is which
           type = type + " " + type;    //??? Which type is which
           System.out.println(getDescription());
        }
```
The parameters passed into pimpMyRide now conflict with the instance variable names. This code will compile, but it won't run correctly, as will be discussed. When

parameters passed into a constructor or method have the same name as instance variables, this is called *shadowing a field.* Shadowing helps clarify how the parameter will be used to set or modify an instance variable. To fix the problem, the *this* keyword helps to disambiguate what is the object's instance variable and what is just the parameter.

```
    void pimpMyRide(String color, String type) {
  this.color = color;
  this.type = type + " " + this.type;
  System.out.println(getDescription());
 }
```

Static methods do not have access to *this* because when you enter a static method, you are not in an object instance. "this" only applies to instances. In the above case, *this* helps provide clarity, but if alternative parameter names are used, you can avoid having to use *this*. In another case, the *this* keyword is the only way to accomplish the task. Consider the Car example with two constructors as specified below.

```
 Car(){
  carCount++;
  serialNumber = carCount;
}
Car(String c, String t){
  carCount++;
  serialNumber = carCount;
  color = c;
  type = t;
}
```

Do you notice some similarities between the two constructors? What happens if the way serial numbers are handled is changed? In this case, you would have to modify two constructors where duplicate code is used to deal with serial numbers. This is not a very good reuse design! How can the common code be isolated and reused in these constructors? Constructors can call other constructors in the same class using *this*. Using *this* is similar to calling a method from within your constructor: you must match the appropriate argument list. To call another constructor, use *this* on the **first line** of another constructor.

```
 Car(){
  carCount++;
  serialNumber = carCount;
}

Car(String c, String t){
  this();
  color = c;
  type = t;
}
```

By having the second constructor call the first one, you don't need to repeat code. Any change made to the first constructor will also affect the second constructor.
Setting up one constructor to call another is called constructor chaining.
Using the *this* keyword is awkward at first, but there are only two uses for it:

- Referring to the current instance (e.g., this.color or) this.getDescription()
- Referring to another constructor (e.g., this(<arg>) from inside a constructor.)

6.11 Summary

Constructors are methods that are used to set the values of object variables, but unlike setter methods they don’t need to be invoked. The *Default* constructor is a no-arguments constructor that is provided by Java if you define a class without explicitly defining any constructors. The Default constructor allows you to create objects of classes that have no specifically designed constructors. We can also pass arguments to constructors in the same way as to ordinary methods. Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. Constructor Overloading is an extension of method overloading, A constructor is overloaded when the same constructor with different number and types of arguments initializes an object with valid initial values. In Java, the destruction of objects takes place automatically. Java uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage." Within an instance method or a constructor, the keyword *this* is a reference to the current object. In other words, *this* refers to the current instance. You can use *this* to refer to any instance variable or instance method of the object from within an instance method or a constructor.

6.12 Some Practice Questions

1. What are constructors? Explain different properties of constructors.
2. What happens when you don’t define a constructor in a class?
3. Explain method overloading and constructor overloading in detail?

6.13 Suggested Readings

- The Complete Reference by Herbert Scheild, Mc-Graw Hill
- Programming with Java by E.Balagurusamy, Mc-Graw Hill
- Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
- Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
- Object Oriented Programming in Java by G.T. Thampi
- Java Programming by C. Xavier

Lesson No. 7 **Author : Kanwal Preet Singh**
Converted into SLM by: Dr. Vishal Singh

Passing Objects and Access Specifiers

7.1 Objectives

After reading this lesson you will be able to understand:

- How to pass objects as arguments
- How to return objects
- Recursion
- Access Specifiers
- Static variables and methods

7.2 Introduction

We can pass objects as arguments to methods just as other variables. Similarly a method can return s to the object to the calling method. In this lesson, we will see how to pass objects as arguments and how to return objects to the calling method. We will also discuss the concept of recursion in which a method calls itself again and again. There are four access specifiers used in Java – public, default, protected and private which restrict the accessibility of the variables to different parts of the program. In the end we will discuss static members which are those members that belong to the class rather than to the object.

7.3 Passing Objects as Arguments

So far we have only been passing simple types as parameters to methods. However, we can also pass objects to methods. Consider the following example:

```
// Objects may be passed to methods.
class Test
 {
   int a, b;
   Test(int i, int j)
    {
      a = i;
      b = j;
    }
  // return true if o is equal to the invoking object
  boolean equals(Test o)
   {
     if(o.a == a && o.b == b)
       return true;
     else
       return false;
   }
  }
class PassOb
 {
  public static void main(String args[])
   {
     Test ob1 = new Test(105, 20);
     Test ob2 = new Test(105, 20);
     Test ob3 = new Test(-10, -10);

     System.out.println("ob1 == ob2: " + ob1.equals(ob2));
     System.out.println("ob1 == ob3: " + ob1.equals(ob3));
   }
}
```

The output is:

```
ob1 == ob2:  true
ob1 == ob3:  false
```

In the above example, we pass an object of class Test to the method equals(). The method compares the class variables of the object that invoked it with object that has been passed to the method and returns true if they are equal.

There are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument use to call it. The second an argument can be passed is call-by-reference. In this method a reference to an argument is passed to the parameter. Inside the subroutine this reference is used to assist the actual argument specified in the call i.e. changes made to the parameter will affect the argument used to call the subroutine. Java uses both methods, depending upon what is passed.

In Java, when you pass a simple type to a method, it is passed by value. When you pass an object to a method, it is passed by a reference. When we create a variable of a class type, we are only creating a reference to an object. Thus, when we pass this reference to a method the parameter that receives it will refer to the same object as that referred to by the argument which means that objects are passed to methods by use of call-by-reference. This will be clear from the following example:

```
// Simple Types are passed by value. Objects are passed by reference.
class Test
 {
   int a, b;
   Test(int i, int j)
     {
      a = i;
      b = j;
     }
    //Pass by value
   void meth(int i, int j)
     {
       i *= 2;
       j /= 2;
     }
   // pass an object
   void meth(Test o)
    {
      o.a *=  2;
      o.b /= 2;
    }
  }
class CallByRef
  {
   public static void main(String args[])
    {
       int  a=5, b=10;
       Test ob = new Test(5, 10);
      System.out.println("a and b before call: " + a + " " + b);
      ob.meth(a, b);
      System.out.println("a and b after call: " + a + " " + b);

      System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);
      ob.meth(ob);
    System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
   }
 }
```
The output is:

```
a and b before call: 5 10
a and b after call: 5 10
ob.a and ob.b before call: 5 10
ob.a and ob.b after call: 10 5
```

It is clear from the above example that when we pass a simple type to a method it is passed by value and when we pass an object to a method it is passed by reference. In fact when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed referred to n object, the copy of that value will still refer to the same object that its corresponding argument does.

7.4 Returning objects

A method can return any type of data including objects that we create. In the following program, the incrByTen() method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test
 {
  int a;
  Test(int i)
   {
    a = i;
   }
  Test incrByTen()
   {
    Test temp = new Test(a+10);
    return temp;
   }
  }
class RetOb
 {
  public static void main(String args[])
   {
    Test ob1 = new Test(2);
    Test ob2;
    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);
    ob2 = ob2.incrByTen();
    System.out.println("ob2.a after second increase: " + ob2.a);
   }
  }
```
7.5 Recursion

The term recursion generally refers to the technique of repeatedly splitting a task into "the same task on a smaller scale". In practice, it often means making a method that calls itself. A method called in this way is often called a recursive method. Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. for example, 3 factorial is 1×2×3, or 6. Here is how a factorial can be computed by use of a recursive method.

```
class Factorial
  {
    int fact(int n)
     {
       int result;
       if ( n ==1) return 1;
       result = fact (n-1) * n;
       return result;
     }
  }
class Recursion
  {
   public static void main (String args[])
     {
       Factorial f =new Factorial();
       System.out.println("Factorial of 3 is " + f.fact(3));
       System.out.println("Factorial of 4 is " + f.fact(4));
       System.out.println("Factorial of 5 is " + f.fact(5));
     }
  }
```
The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```
If you are unfamiliar with recursive methods, then the operation of **fact()** may seem a bit confusing. Here is how it works. When **fact()** is called with an argument of 1, the function returns 1; otherwise it returns the product of **fact(n-1)*n.** to evaluate this expression, **fact()** is called with **n-1**. this process repeats until **n** equals 1 and the calls to the method begin returning.

To better understand how the **fact()** method works, let's go through a short example. When you compute the factorial of 3, the first call to **fact()** will cause a second call to be made with an argument of 2. this invocation will cause **fact()** to be called a third time with an argument of 2. This call will return 1, which is then be called a third time with an argument of 1. This call will return1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact()** and multiply by 3 (the original value of **n).** This yields the answer, 6. You might find it interesting to insert **println()** statements into **fact()** which will show at what level each call is and what the intermediate answers are. The following diagram shows how it works:

```
fact(3)
  fact(2)
    fact(1)
    return 1
  return 2*1 = 2
return 3*2 = 6
```

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. A recursive call does not make a new copy of the method. Only the arguments are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the call inside the method. Recursive methods could be said to "telescope" out and back.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables, it is possible that the stack could be exhausted. If this occurs, the java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild.

Our factorial implementation exhibits the two main components that are required for every recursive function. The *base case* returns a value without making any subsequent recursive calls. It does this for one or more special input values for which the function can be evaluated without recursion. For fact(), the base case is N = 1. The *reduction step* is the central part of a recursive function. It relates the function at one (or more) inputs to the function evaluated at one (or more) other inputs. For fact(), the reduction step is N * fact(N-1). All recursive functions must have these two components. Furthermore, the sequence of parameter values must *converge* to the base case. For fact(), the value of N decreases by one for each call, so the sequence of parameter values converges to the base case N = 1.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way.

7.6 Java Access Specifiers

access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. To take advantage of encapsulation, you should minimize access whenever possible. Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. A member has package or default accessibility when no accessibility modifier is specified. The following Access Modifiers are provided in Java:

- **private**
- **protected**
- **default**
- **public**

public access modifier

Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier

The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accessed by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier

The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

default access modifier

Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

Class Access Control Modifiers

1. A class can have either the **public** or the **default** access control level.
2. You make a class public by using the public access control modifier.
3. A class whose declaration bears no access control modifier has default access.

Understand the effects of public and private access

```
class Test
 {
  int a; // default access
  public int b; // public access
```

```
   private int c; // private access
   void setc(int i)
    {
      c = i;
    }

   int getc()
    {
      return c;
    }
 }
class AccessTest
 {
   public static void main(String args[])
    {
      Test ob = new Test();
      ob.a = 10;
      ob.b = 20;
      // ob.c = 100; Not Possible. Can't access private member outside its class
      ob.setc(100);
      System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
    }
  }
```

We will come back to the access specifiers when we have done packages. We can understand default and protected access specifiers only when we have full understanding of packages.

7.7 Static Variables

Adding a couple of additional fields to the car example should help to clarify the class variable concept. First, each car has a serialNumber (called a vehicle identification number or VIN). This is pretty simple. Based on what you have already learned, simply add a serialNumber instance variable to the Car class.

```
public class Car
 {
    String color;
    String type;
    int serialNumber;
    ...  // The rest of the class goes here. This symbol (...) is used
         // throughout this text to indicate that not all the code is
         // is shown but only the code that is pertinent at the time.
 }
```

This serialNumber, however, must be a unique integer for each car created. In this case, the serialNumber should be unique for each car object created from the Car class. To make sure each car has a unique serialNumber, you can use the Car class and a class variable to keep track of the total number of cars created.

```
public class Car
 {
   String color;
   String type;
   int serialNumber;
   static int carCount;
      ...
  }
```

Here the modifier "**static**" was added to the carCount variable in this class. The carCount variable is related to the class, not any single Car object. To access this variable, you use the name of the class and the class variable name. For example, the line of code below sets the carCount to 1.

```
    Car.carCount = 1;
```

Oddly, you can use either the class name (Car) or any object reference of that type to access class variables, so the following code would do the same thing:

```
    Car myCar = new Car("black", "Ranger");
    myCar.carCount = 1;
```

This makes it look like carCount is an instance variable, doesn't it? Therefore, it is considered clearer and preferred, to use the class name when accessing class/static variables.

Note that the carCount variable is created and initialized not when an object is instantiated but when the class is first loaded into the JVM by the class loader! That is because the carCount data is associated with the class (Car) and not any single instance of the class (myCar). No matter how many Cars get created, there is still just one carCount. Recall that what is needed is to set the serialNumber for every Car and that number must be unique. The carCount class variable can be used to achieve this goal. However, to pull this off, more work on the constructors for the Car class is needed.

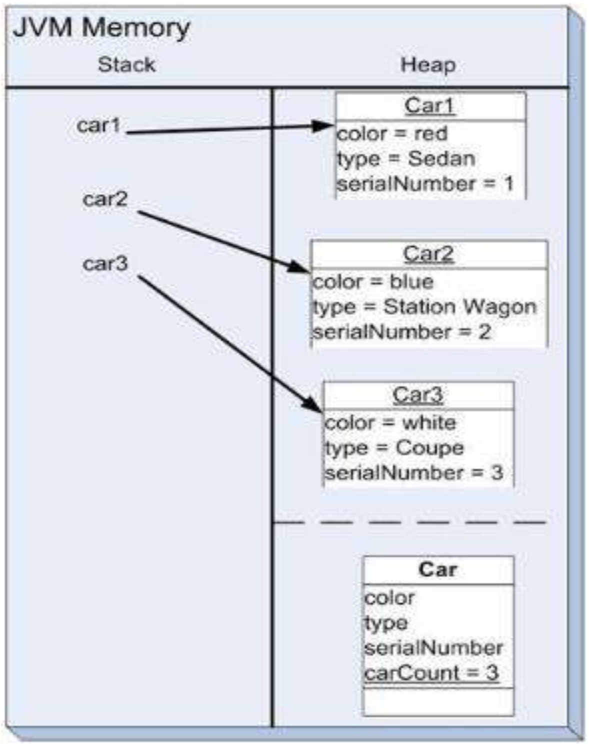
```
 Car()
  {
    carCount++;
    serialNumber = carCount;
  }
 Car(String c, String t)
  {
      color = c;
      type = t;
      carCount++;
      serialNumber = carCount;
  }
```

This code increments the number of Cars (carCount) every time a Car is created and then assigns carCount to the car object's serialNumber. Now each Car has a unique serial number, created from the class's carCount. When each car is created, the constructor increases the value of carCount by one. The carCount variable represents the total number of cars that have been created so far. Now take a look at the code below to visualize what is happening.

```
Car car1, car2, car3;        //create 3 object references.
car1 = new Car("red","Sedan");
car2 = new Car("blue", "Station Wagon");
car3 = new Car("white", "Coupe");
```

When the declaration of a Car reference is made (Car car1), the class loader must load the Car class into the JVM. At that time, carCount is initialized and available. Every object type used in an application has a respective Class object stored in a special place on the JVM's heap. This Class object contains all of the details about the object type:

- What properties it has
- What the properties' types are
- What methods the class has and what arguments the methods take
- What code is executed when a method is called and so forth. Static data is stored in the Class object on the JVM's heap.



All three cars have their own independent color, type, and serialNumber, but they share one car Count variable.

Class variables are global (only one exists in memory per class). All instances share the one carCount, which is obtained through the class. Each instance has its own color, type and serialNumber. All code in the JVM can reference static information of a class depending on the access modifier.

7.8 Static Method

The static keyword can also be applied to a method. A static method, like a static variable, is associated with the class, not the objects (instances). Static methods are also called class methods (versus nonstatic methods, which are instance methods). Methods you have seen so far are instance methods. Below, another instance method, drive(), is added to the Car class.

```
public class Car
{
  String color;
  String type;

  void drive()
   {
     System.out.println("Put 'er in gear and drive it like you stole it");
   }
    ...
 }
```

To call an instance (nonstatic) method, you must have an object reference.

```
    Car c = new Car();
    c.drive();          //Correct
    Car.drive();  //Wrong (what car are you driving?)
```

To define a static method, simply add the *static* keyword as a modifier to the method as shown below.

```
public class Car
 {
    ...
    static void resetCarCount()
     {
        carCount = 0;
     }
   ...
 }
```

To call a static method, you only need the name of the class.

```
    Car c = new Car();
    c.resetCarCount();    //Legal but confusing
    Car.resetCarCount();  //Proper way to code
```

As shown above, just as with class variables, you can call on a static method using any object of that type. However, this makes it look like resetCarCount() is an instance method, doesn't it? Again, it is considered clearer and preferred, to use the class name when accessing static methods.

Static methods do not have access to object data. Looking at the example below, what color are you changing if you called resetCarCount?

```
public class Car
 {
    ...
    static void resetCarCount()
      {
        carCount = 0;
        color = "blue"; //Wrong - which instance's color are you changing?
      }
   ...
 }
```

Now you might be asking yourself, "What's the point of static methods?" If so, that's a good and fair question. Static methods essentially have two purposes.

1) They are used to access (update or fetch) class variable data. Although this can be done with any instance of the class, it is considered more appropriate to use class methods for this purpose. In some cases, you may not have an instance of an object created before the data is needed. You don't want to have to create an object just to be able to access class variables.
2) Static methods provide functionality without the need for an object/instance. For example, mathematical formulas are great reasons to have static methods. Should you have to create an instance of some object to compute sine, cosine or tangent? Examine the Math class to see some excellent uses of static methods. There is no need to create an instance of Math to compute the absolute value of a number!

7.9 Static Initialization Block

Classes can also have initialization blocks. More precisely, they can have *static initialization* blocks. Like a normal initialization block, a static initialization block is a normal block of code enclosed in braces {} preceded by the *static* keyword.

```
public class Car
 {
      ...
      static
       {
         carCount = 1;
       }
      ...
  }
```

A class can have any number of static initialization blocks and they can appear anywhere in the class. They are often grouped together for better maintenance. The static initialization blocks are called in the order they appear in the code. Static initialization code blocks get executed once when the class is loaded. Just like static methods, they can only initialize static variables of the class.

7.10 Summary

We can pass objects as arguments to methods just as other variables. Similarly a method can return s to the object to the calling method. The term recursion generally refers to the technique of repeatedly splitting a task into "the same task on a smaller scale". In practice, it often means making a method that calls itself. A method called in this way is often called a recursive method. The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. There are four access specifiers used in Java – public, default, protected and private. Static members are those members that belong to the class rather than to the object. There exists only one copy of static variables for all the class objects and static methods can access only the static variables and other static methods.

7.11 Short Answer Type Questions

1. Explain the concept of recursion?
2. Write a program to print Fibonacci series using recursion?
3. Why do we use static variables?

7.12 Long Answer Type Questions

1. Explain in detail with the help of an example how to pass objects as arguments and how to return objects to the calling method in java.
2. Explain in detail the different access specifiers used in Java.

7.13 Suggested Readings

- The Complete Reference by Herbert Scheild, Mc-Graw Hill
- Programming with Java by E.Balagurusamy, Mc-Graw Hill
- Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
- Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
- Object Oriented Programming in Java by G.T. Thampi
- Java Programming by C. Xavier

Lesson No. 8 **Author : Kanwal Preet Singh**
Converted into SLM by: Dr. Vishal Singh

Nested and Inner Classes

8.1 Objectives

After reading this lesson you will be able to understand:

- Nested Classes
- Static Nested Classes
- Inner Classes
- Strings
- Java String Functions

8.2 Introduction

We can define a class within another class. Such classes are called nested classes. Nested classes are divided into two categories: static and non-static. There are several compelling reasons for using nested classes and they are discussed in this chapter. The other topics that we will discuss in this lesson are strings and inheritance. String is a special class built into the Java language defined in the java.lang package. The String class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class.

8.3 Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass
 {
   ...
   class NestedClass
   {
      ...
   }
}
```

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass
 {
   ...
   static class StaticNestedClass
    {
       ...
    }
   class InnerClass
    {
       ...
    }
 }
```

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world. More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

8.4 Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.

Note: A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.
Static nested classes are accessed using the enclosing class name:
OuterClass.StaticNestedClass
For example, to create an object for the static nested class, use this syntax:
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

The definition of a static nested class looks just like the definition of any other class, except that it is nested inside another class and it has the modifier static as part of its declaration. A static nested class is part of the static structure of the containing class. It can be used inside that class to create objects in the usual way. If it has not been declared private, then it can also be used outside the containing class, but when it is used outside the class, its name must indicate its membership in the containing class. This is similar to other static components of a class: A static nested class is part of the class itself in the same way that static member variables are parts of the class itself.

For example, suppose a class named WireFrameModel represents a set of lines in three-dimensional space. (Such models are used to represent three-dimensional objects in graphics programs.) Suppose that the WireFrameModel class contains a static nested class, Line, that represents a single line. Then, outside of the class WireFrameModel, the Line class would be referred to as WireFrameModel.Line. Of course, this just follows the normal naming convention for static members of a class. The definition of the WireFrameModel class with its nested Line class would look, in outline, like this:

```
public class WireFrameModel
 {
   . . . // other members of the WireFrameModel class

   static public class Line
    {
       // Represents a line from the point (x1,y1,z1)
       // to the point (x2,y2,z2) in 3-dimensional space.
      double x1, y1, z1;
      double x2, y2, z2;
    }  // end class Line
     . . . // other members of the WireFrameModel class
  } // end WireFrameModel
```

Inside the WireFrameModel class, a Line object would be created with the constructor "new Line()". Outside the class, "new WireFrameModel.Line()" would be used. A static nested class has full access to the static members of the containing class, even to the private members. Similarly, the containing class has full access to the members of the nested class. This can be another motivation for declaring a nested class, since it lets you give one class access to the private members of another class without making those members generally available to other classes.

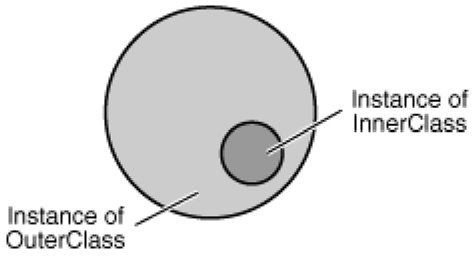
When you compile the above class definition, two class files will be created. Even though the definition of Line is nested inside WireFrameModel, the compiled Line class is stored in a separate file. The name of the class file for Line will be WireFrameModel$Line.class.

8.5 Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself. Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass
{
   ...
   class InnerClass
    {
       ...
    }
}
```

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance. The next figure illustrates this idea.



An Instance of InnerClass Exists Within an Instance of OuterClass. To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

In Java, a nested class is any class whose definition is inside the definition of another class. Nested classes can be either named or anonymous. A named nested class, like most other things that occur in classes, can be either static or non-static.

Non-static nested classes are referred to as inner classes. Inner classes are not, in practice, very different from static nested classes, but a non-static nested class is actually associated with an object rather than to the class in which it is nested. This can take some getting used to.

Any non-static member of a class is not really part of the class itself (although its source code is contained in the class definition). This is true for inner classes, just as it is for any other non-static part of a class. The non-static members of a class specify what will be contained in objects that are created from that class. The same is true -- at least logically -- for inner classes. It's as if each object that belongs to the containing class has its own copy of the nested class. This copy has access to all the instance methods and instance variables of the object, even to those that are declared private. The two copies of the inner class in two different objects differ because the instance variables and methods they refer to are in different objects. In fact, the rule for deciding whether a nested class should be static or non-static is simple: If the nested class needs to use any instance variable or instance method from the containing class, make the nested class non-static. Otherwise, it might as well be static.

From outside the containing class, a non-static nested class has to be referred to using a name of the form variableName.NestedClassName, where variableName is a variable that refers to the object that contains the class. This is actually rather rare, however. A non-static nested class is generally used only inside the class in which it is nested, and there it can be referred to by its simple name.

In order to create an object that belongs to an inner class, you must first have an object that belongs to the containing class. (When working inside the class, the object "this" is used implicitly.) The inner class object is permanently associated with the containing class object, and it has complete access to the members of the containing class object. Looking at an example will help, and will hopefully convince you that inner classes are really very natural. Consider a class that represents poker games. This class might include a nested class to represent the players of the game. This structure of the PokerGame class could be:

```
public class PokerGame {  // Represents a game of poker.
   private class Player {  // Represents one of the players in this game.
      .
      .
      .
   } // end class Player
   private Deck deck;      // A deck of cards for playing the game.
   private int pot;        // The amount of money that has been bet.
   .
   .
   .
} // end class PokerGame
```

If game is a variable of type PokerGame, then, conceptually, game contains its own copy of the Player class. In an instance method of a PokerGame object, a new Player object would be created by saying "new Player()", just as for any other class. (A Player object could be created outside the PokerGame class with an expression such as "game.new Player()". Again, however, this is very rare.) The Player object will have access to the deck and pot instance variables in the PokerGame object. Each PokerGame object has its own deck and pot and Players. Players of that poker game use the deck and pot for that game; players of another poker game use the other game's deck and pot. That's the effect of making the Player class non-static. This is the most natural way for players to behave. A Player object represents a player of one particular poker game. If Player were a static nested class, on the other hand, it would represent the general idea of a poker player, independent of a particular poker game.

8.6 Strings

String is a special class built into the Java language defined in the java.lang package. The String class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class. Strings are immutable; that is, they cannot be modified once created. For example:

String str = "This is string literal";

On the right hand side a String object is created represented by the string literal. Its object reference is assigned to the str variable. The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. For example:

String str = "First part" + " second part";

// --- Is the same as:

String str = "First part second part";

Integers will also be converted to String after the (+) operator:

String str = "Age=" + 25;

Each Java object has the String toString() inherited from the Object class. This method provides a way to convert objects into Strings. Most classes override the default behavior to provide more specific (and more useful) data in the returned String. The String class provides a nice set of methods for string manipulation. Since String objects are immutable, all methods return a new String object. For example:

name = name.trim();

The trim() method returns a copy of the string with leading and trailing whitespace removed. Note that the following would do nothing useful:

```
  name.trim();   // wrong!
```

This would create a new trimmed string and then throw it away.

If 2 or more Strings have the same set of characters in the same sequence then they share the same reference in memory. Below illustrates this phenomenon.

```
String str1 = “My name is bob”;
String str2 = “My name is bob”;
String str3 = “My name ”+ “is bob”; //Compile time expression
String name = “bob”;
String str4 = “My name is” + name;
String str5 = new String(“My name is bob”);
```

In the above code all the String references str1, str2 and str3 denote the same String object, initialized with the character string: “My name is bob”. But the Strings str4 and str5 denote new String objects.

//String Equality

```
public class StringsDemo1
 {
   public static void main(String[] args)
     {
         String str1 = "My name is bob";
         String str2 = "My name is bob";
         String str3 = "My name " + "is bob"; //Compile time expression
         String name = "bob";
         String str4 = "My name is " + name;
         String str5 = new String("My name is bob");
         System.out.println("str1 == str2 : " + (str1 == str2));
         System.out.println("str2 == str3 : " + (str2 == str3));
         System.out.println("str3 == str1 : " + (str3 == str1));
         System.out.println("str4 == str5 : " + (str4 == str5));
         System.out.println("str1 == str4 : " + (str1 == str4));
         System.out.println("str1 == str5 : " + (str1 == str5));
         System.out.println("str1.equals(str2) : " + str1.equals(str2));
         System.out.println("str2.equals(str3) : " + str2.equals(str3));
         System.out.println("str3.equals(str1) : " + str3.equals(str1));
         System.out.println("str4.equals(str5) : " + str4.equals(str5));
         System.out.println("str1.equals(str4) : " + str1.equals(str4));
         System.out.println("str1.equals(str5) : " + str1.equals(str5));
     }
   }
```

Output is:

```
str1 == str2 : true
str2 == str3 : true
str3 == str1 : true
str4 == str5 : false
str1 == str4 : false
str1 == str5 : false
str1.equals(str2) : true
str2.equals(str3) : true
str3.equals(str1) : true
str4.equals(str5) : true
str1.equals(str4) : true
str1.equals(str5) : true
```

The == operator is used when we have to compare the String object references. If two String variables point to the same object in memory, the comparison returns true. Otherwise, the comparison returns false. Note that the '==' operator does not compare the content of the text present in the String objects. It only compares the references the 2 Strings are pointing to. The equals method is used when we need to compare the content of the text present in the String objects. This method returns true when two String objects hold the same content. The following section briefly discusses the different functions of the String class that can be used on the objects of String class.

8.7 Java String Functions

The following program explains the usage of the some of the basic String methods like ;

1. **compareTo**(String anotherString): Compares two strings lexicographically. It compares char values similar to the equals method. The compareTo method returns a negative integer if the first String object precedes the second string. It returns zero if the 2 strings being compared are equal. It returns a positive integer if the first String object follows the second string.
2. **charAt**(int index): Returns the character at the specified index.
3. **getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin): Copies characters from this string into the destination character array.
4. **length**(): Returns the length of this string.
5. **equals**(Object anObject): Compares this string to the specified object.
6. **equalsIgnoreCase**(String anotherString): Compares this String to another String, ignoring case considerations.
7. **toUpperCase**(): Converts all of the characters in this String to upper case using the rules of the default locale.
8. **toLowerCase**(): Converts all of the characters in this String to upper case using the rules of the default locale.
9. **concat**(String str): Concatenates the specified string to the end of this string.
10. **indexOf**(int ch): Returns the index within this string of the first occurrence of the specified character.
11. **indexOf**(int ch, int fromIndex): Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
12. **indexOf**(String str): Returns the index within this string of the first occurrence of the specified substring.
13. **indexOf**(String str, int fromIndex): Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
14. **lastIndexOf**(int ch): Returns the index within this string of the last occurrence of the specified character.
15. **lastIndexOf**(int ch, int fromIndex): Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
16. **lastIndexOf**(String str): Returns the index within this string of the rightmost occurrence of the specified substring.
17. **lastIndexOf**(String str, int fromIndex): Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
18. **substring**(int beginIndex): Returns a new string that is a substring of this string.

19. **substring**(int beginIndex, int endIndex): Returns a new string that is a substring of this string.
20. **replace**(char oldChar, char newChar): Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
21. **trim**(): Returns a copy of the string, with leading and trailing whitespace omitted.
22. **toString**(): This object (which is already a string!) is itself returned.

The following program demonstrates the use of String methods:

```
// Program to demonstrate String methods.
public class StringsDemo2
 {
   public static void main(String[] args)
    {
             String str1 = "My name is bob";
             char str2[] = new char[str1.length()];
             String str3 = "bob";
             String str4 = "cob";
             String str5 = "BoB";
             String str6 = "bob";
             System.out.println("Length of the String str1 : " + str1.length());
             System.out.println("Character at position 3 is : "
                          + str1.charAt(3));
             str1.getChars(0, str1.length(), str2, 0);
             System.out.print("The String str2 is : ");
             for (int i = 0; i < str2.length; i++)
               {
                 System.out.print(str2[i]);
               }
             System.out.println();
             System.out.print("Comparision Test : ");
             if (str3.compareTo(str4) < 0)
              {
                 System.out.print(str3 + " < " + str4);
              }
             else if (str3.compareTo(str4) > 0)
              {
                  System.out.print(str3 + " > " + str4);
              }
              else
              {
                  System.out.print(str3 + " equals " + str4);
              }
             System.out.println();
```

```
        System.out.print("Equals Test");
        System.out.println("str3.equalsIgnoreCase(5) : "
                + str3.equalsIgnoreCase(str5));
        System.out.println("str3.equals(6) : " + str3.equals(str6));
        System.out.println("str1.equals(3) : " + str1.equals(str3));
        str5.toUpperCase(); //Strings are immutable
        System.out.println("str5 : " + str5);
        String temp = str5.toUpperCase();
        System.out.println("str5 Uppercase: " + temp);
        temp = str1.toLowerCase();
        System.out.println("str1 Lowercase: " + str1);
        System.out.println("str1.concat(str4): " + str1.concat(str4));
        String str7temp = "  \t\n Now for some Search and Replace Examples ";
        String str7 = str7temp.trim();
        System.out.println("str7 : " + str7);
        String newStr = str7.replace('s', 'T');
        System.out.println("newStr : " + newStr);
        System.out.println("indexof Operations on Strings");
        System.out.println("Index of p in " + str7 + " : " + str7.indexOf('p'));
        System.out.println("Index of for in " + str7 + " : " + str7.indexOf("for"));
        System.out.println("str7.indexOf(for, 30) : " + str7.indexOf("for", 30));
        System.out.println("str7.indexOf('p', 30) : "+ str7.indexOf('p', 30));
        System.out.println("str7.lastIndexOf('p') : "+ str7.lastIndexOf('p'));
        System.out.println("str7.lastIndexOf('p', 4) : " + str7.lastIndexOf('p',  4));
        System.out.print("SubString Operations on Strings");
        String str8 = "SubString Example";
        String sub5 = str8.substring(5); // "ring Example"
        String sub3_6 = str8.substring(3, 6); // "Str"
        System.out.println("str8 : " + str8);
        System.out.println("str8.substring(5) : " + sub5);
        System.out.println("str8.substring(3,6) : " + sub3_6);
    }
}
```

Output is:

```
Length of the String str1 : 14
Character at position 3 is : n
The String str2 is : My name is bob
Comparision Test : bob < cob
Equals Teststr3.equalsIgnoreCase(5) : true
str3.equals(6) : true
str1.equals(3) : false
str5 : BoB
```

```
str5 Uppercase: BOB
str1 Lowercase: My name is bob
str1.concat(str4): My name is bobcob
str7 : Now for some Search and Replace Examples
newStr : Now for Tome Search and Replace ExampleT
Indexof Operations on Strings
Index of p in Now for some Search and Replace Examples : 26
Index of for in Now for some Search and Replace Examples : 4
str7.indexOf(for, 30) : -1
str7.indexOf('p', 30) : 36
str7.lastIndexOf('p') : 36
str7.lastIndexOf('p', 4) : -1
SubString Operations on Stringsstr8 : SubString Example
str8.substring(5) : ring Example
str8.substring(3,6) : Str
```

8.8 Summary

The Java programming language allows you to define a class within another class. Such a class is called a *nested class.* Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*. String is a special class built into the Java language defined in the java.lang package. The String class represents character strings. String literals in Java programs, such as "abc", are implemented as instances of this class. There are different functions of the String class that can be used on the objects of String class.

8.9 Review Questions

1. Differentiate between static nested and inner classes?
2. Can we access inner class objects outside the class it is defined in? How?
3. What is inheritance? Explain with the help of an example. How can we access base class constructors? Explain with example.

8.10 Suggested Readings

- The Complete Reference by Herbert Scheild, Mc-Graw Hill
- Programming with Java by E.Balagurusamy, Mc-Graw Hill
- Java : A Beginner's Guide by Herbert Schildt, Mc-Graw Hill
- Introduction to Java Programming by Y.Daniel Cians, Prentice Hall
- Object Oriented Programming in Java by G.T. Thampi
- Java Programming by C. Xavier

Last Updated on July 2023

Mandatory Student Feedback Form

https://forms.gle/KS5CLhvpwrpgjwN98

Note: Students, kindly click this google form link, and fill this feedback form once.