NEPAL BANEPA POLYTECHNIC INSTITUTE

Java Programming

Department of Information Technology

1/1/2019

Contents

Chapter 1
Introduction

What is Object Oriented Programming?

The fundamental idea behind object-oriented programming is to combine both *states* (also called *data* or *fields* or *attributes*) and *behaviors* (also called *functions* or *methods*) that operate on that data into a single unit. This unit is called an *object*. The data is hidden, so it is safe from accidental alteration. An object's methods typically provide the only way to access its data. In order to access the data in an object, we should know exactly what methods interact with it. No other methods can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

An object-oriented program typically consists of a number of objects, which communicate with each other by calling one another's methods. This is called *sending a message* to the object. This kind of relation is provided with the help of communication between two objects and this communication is done through information called *message*.

In addition, object-oriented programming supports *encapsulation*, *abstraction*, *inheritance*, and *polymorphism* to write programs efficiently. Some examples of object-oriented programming language include Java, *Simula*, *Smalltalk*, *C++*, *Python*, *C#*, *Visual Basic .NET*.

Classes and Objects

The concept of *classes* and *objects* is the heart of the OOP. A class is a framework that specifies what data and what methods will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. An object is a software bundle of related data and methods created from a class. For example we can think of the class **Students**. This class has some properties like name, address, marks, grade, and so on. Similarly it can have the methods like findPercentage(), findGPA(), etc. Now here we can see that from this class, we can create any number of objects of the type Students. In programming we say that an object is an instance of the class.

Abstraction

Abstraction is the essence of OOP. Abstraction means the representation of the essential features without providing the internal details and complexities. In OOP, abstraction is achieved by the help of class, where data and methods are combined to extract the essential features only. For example, if we represent chair as an object it is sufficient for us to understand that the purpose of chair is to sit. So we can hide the details of construction of chair and things required to build the chair.

OOP Principles

The three basic concepts underlying OOP are: *encapsulation*, *inheritance*, and *polymorphism*.

Encapsulation

Encapsulation is the process of combining the data and methods into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privilege to the data inside the class. So the term ***data hiding*** is possible due to the concept of encapsulation, since the data are hidden from the outside world.

Inheritance

Inheritance is the process of acquiring certain attributes and behaviors from parents. For examples, cars, trucks, buses, and motorcycles inherit all characteristics of vehicles. Object-oriented programming allows classes to inherit commonly used data and functions from other classes. If we derive a class (called super class) from another class (called sub class), some of the data and methods can be inherited so that we can reuse the already written and tested code in our program, simplifying our program.

Polymorphism

Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. However the behavior depends upon the attribute the name holds at particular moment. For example if we have the behavior called communicate for the objects vertebrates, then the communicate behavior applied to objects say dogs is quite different from the communicate behavior to objects human. So here the same name communicate is used in multiple process one for human communication, what we simply call talking. The other is used fro communication among dogs, we may say barking, definitely not talking.

Package

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming.

Java

Java is general purpose, object oriented, high-level programming language. It is used to develop different programs that run in network, desktops, servers, embedded systems etc. Hence, we can say that java has applications in every sector in all environments.

History of Java

- In 1991, a research group working as part of Sun Microsystems's "Green" project headed by James Gosling was developing software to control consumer electronic devices. The goal was to develop a programming language that could be used to develop software for consumer electronic devices like TVs, VCRs, toasters, and the like. As a result the team announced a new language named "Oak".
- In 1992, the team demonstrated the application of their new language to control a list of home applications using a hand-held device with a tiny touch-sensitive screen.
- In 1994, the members of the Green project developed a World Wide Web (WWW) browser completely in Java that could run Java applets. The browser was originally called WebRunner, but is now known as HotJava.
- In 1995, Oak was renamed "Java" due to some legal snags. Many popular companies including Netscape and Microsoft announced their support to Java.
- In 1997, Sun released the Servlet API, which revolutionized server-side Web development.
- In 1999, Sun released the first version of the Java 2, Enterprise Edition (J2EE) specification that included Java Server Pages (JSP) and Enterprise JavaBeans (EJB) in a highly distributed enterprise middleware.

Java Buzzwords

Simple

Java syntax is similar to C and C++. However, most of the poorly used and confusing parts form C++ are omitted or managed. For example pointer for handling memory in C or C++ is not there in Java and it handles memory automatically, and similarly, structs and typedef constructs are removed. So we say that Java is simple to learn and simple to implement.

Object Oriented

Java is a true object-oriented programming language that supports encapsulation, inheritance, and polymorphism. Almost everything in Java is an object. Since it is object oriented, everything but the primitive data types is represented as class and its instances and members.

Distributed

Java is designed as a distributed language for creating applications on networks. So, application development for running in multiple machines to run concurrently and interact with each other using java is very easy and efficient.

Robust

Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time code checking. It handles all memory management problems and also incorporates strong garbage collection facility.

Secure

Since Java was intended to work in the distributed environment, security was the issue that was taken care. For example, applets running in a Web browser cannot access a Web client's underlying file system or any other Web site other than that from which it originated and all Java programs run in a virtual machine that protects the underlying operating system from harm. Similarly, the series of test and restriction of memory manipulation also secure the application. Also, the absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

Architecture Neutral

Since Java runs inside of a Virtual Machine (a program written in operating system specific code that provides a common front-end to all operating systems), the program does not depend on the underlying operating system or hardware.

Portable

The phrase write once, run anywhere is the major concept for the portability that Java adheres by compiling Java code into byte-code, which then is interpreted by Virtual Machine to operating system–specific instructions. Because of this, any Java code that you write and compile into byte-code will run on any operating system for which there exists a compatible Virtual Machine.

Compiled and Interpreted

Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two-stage system. First, Java compiler translates source code into what is known as **bytecode** instructions. Bytecodes are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly executed by the machine that is running the Java program.

High Performance

Though we say Java programs are interpreted, we cannot say that java does not have high performance. The performance gain is due to the JIT (Just In Time) compilation that cache the recurring process to speed up the interpretation.

Multithreaded

Java allows multithreading such that the applications have the capability to run multiple things at the same time for e.g. writing a report and checking the spellings at a time.

Dynamic and Extensible

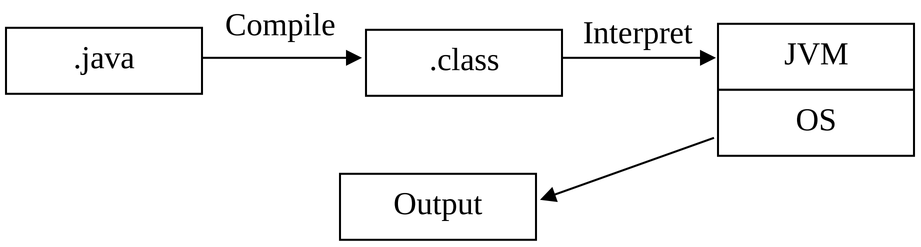
Java is dynamic since it is capable of dynamically linking in new class libraries, methods or functions, and objects.

Java programs support **native methods**. Native methods are the functions written in other languages such as C and C++. This feature enables the programmers to use the efficient functions available in languages other than Java. Native methods are linked dynamically at runtime.

Java Virtual Machine

When java compiles a code then it converts the program into the special architectural neutral executable program called *bytecode* that can only be run by using the special program that acts as a machine called **JVM** (**Java Virtual Machine**). JVM is not an actual machine but the run time system for Java.

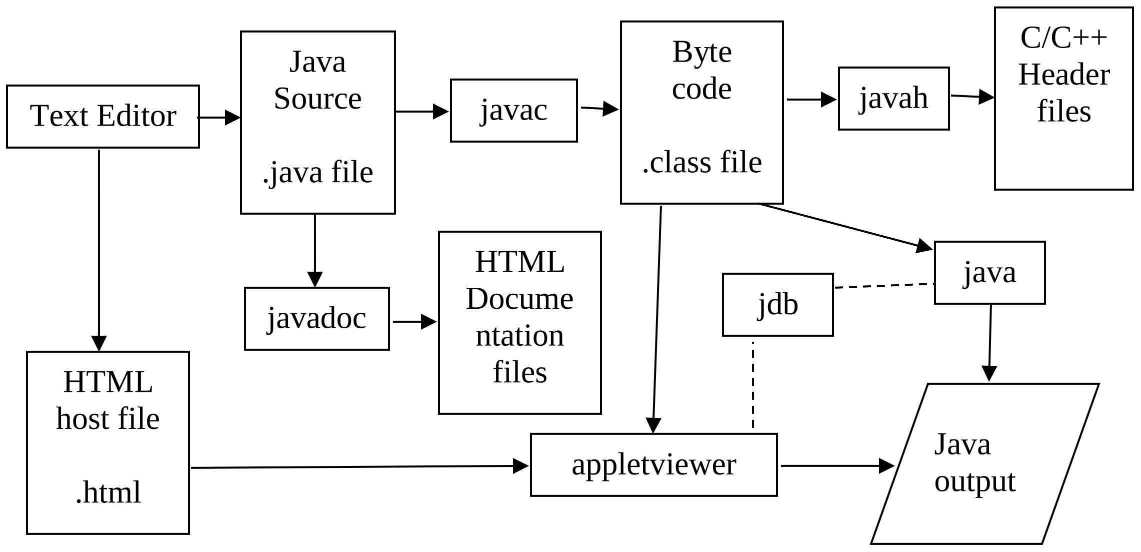
How Java Works



The java compiler produces an intermediate code known as *bytecode* for the Java Virtual Machine (JVM). This bytecode is not hardware specific. To generate hardware specific code (known as machine code), JVM interprets the bytecode by acting as an intermediary between bytecode and machine code.

Building Java Applications and Applets

There are two types of Java programs namely, java application and the java applet. The former runs within the system as a standalone program without using help of other tools, whereas the latter requires html file to include the applet and run it. The figure below is the pictorial representation of creating and running the both type of programs as well as some other functions.



Writing Your First Java Program

Here is the simple program in java and this can be written in any available text editors like notepad, WordPad, etc.

```
public class Welcome
{
    public static void main(String[]args)
    {
        System.out.println("NBPI");
    }
}
```

Compiling Your First Java Program

Write the above code and save it as a filename ***Welcome.java*** (remember Capital 'W'). Open a command prompt and navigate to the folder where you saved your ***`Welcome.java`*** file. You are ready to compile here. To compile the java file use the tool ***'javac'*** that comes along with Java 2 SDK as shown below:

javac Welcome.java

When this is done there are two possibilities: either there is no message and the prompt appears or there are messages that show the errors. If there are errors, your program will not compile and you must correct them to run the program.

Running Your First Java Program

After the successful compilation of ***Welcome.java***, ***`Welcome.class`*** file is created. This is the file that contains the byte-code. To run ***`Welcome`***, type the following from a command prompt in the folder that holds ***`Welcome. class`***.

java Welcome

The result should be displayed to the screen as ***NBPI***.

Understanding Your First Java Program

Line 1: ***public class Welcome***

A new class called ***`Welcome`*** is being defined. You can create new classes by using the keyword `class` followed by a name for the class.

Lines 2, 7 and 4, 6: ***the brace pair { and }***

Define the body of the `Welcome class by pair 2 and 7 i.e.` everything that is included between lines 2 and 7 is part of the Welcome class. Similarly body of the main method is defined by lines 4 and 6.

Line 3: ***public static void main(String[] args)***

It defines a method, or function, that is a member of the ***Welcome*** class called ***main***. ***main*** is the entry point for java applications that is run using a tool java. Let's closely look at the sub parts in the method:

`public`

The keyword `public` is known as an access modifier; an access modifier defines who can and cannot see this method (or variable.) There are three possible values for access modifiers: `public`, `protected`, and `private`, each having their own restrictions, which we will cover later. In this case, it is saying that this function, `main`, is publicly available to anyone who wants to call it. This is essential for the `main` function; otherwise the Java Runtime Engine would not be able to access the method, and hence could not launch our application.

`static`

The `static` keyword tells the compiler that there is one and only one method (or variable) to be used for all instances of the class. This functionality will be explained more later, but for now remember that `main` functions have to be static in a class.

`void`

The term `void` refers to the return type. Method can return values; for example, integers, floating-point values, characters, strings, and more. If a function does not return any value, it is said to return `void`. In this declaration it is saying that the `main` function does not return a value.

`main`

The word `main` is the name of the function. As previously mentioned, `main` is a special function that must be defined when writing a Java application. It is the entry-point into your class that the Java Runtime Engine processes when it starts; it will control the flow of your program.

(String[] args)
The portion enclosed in parentheses next to the function name is the parameter list passed to the method. You can pass any type of data for that method to work with. In this case, the `main` method accepts an array (a collection or set of items) of String objects that represent the command-line arguments the user entered when he launched the application. See below for explanation:

java Welcome one two three four
The command line then would have four items sent to it: `one`, `two`, `three`, and `four`. The variable arguments would contain an array of these four strings as:
args[0] = "one"
args[1] = "two"
args[2] = "three"
args[3] = "four"

Line 5: ***System.out.println("NBPI");***
Let's break apart this statement:

`System`
`System` is actually a class that the Java language provides for you. This class is used to access the standard input (keyboard), standard output (monitor), and standard error (usually a monitor unless you have a separate error output defined).
`out`
The `System` class's `out` property (represents output device) is defined as follows:

public static final PrintStream out

All the keywords used have usual meaning. The `final` keyword says that this variable cannot change in value. The `PrintSteam` class contains a collection of `print`, `println`, and `write` methods that know how to print different types of variables This is how `println` can print out so many different types of values.

`println`
This is the method of ***PrintStream*** class that prints the content passed as a parameter adding new line in it.

Chapter 2
Fundamental Programming Structures

Introduction

Java program is a collection of different atomic elements like whitespaces, identifiers, literals, comments, operators, separators, keywords, data types, statements etc. In this chapter we will discuss the basic concepts of Java programming.

Whitespace

In java indentations are not obligatory, so writing 100 lines of code and 10 lines of compact code is same. We can write a program continuously without separating it from the previous statement by a new line or other forms of white space. Whitespace is mandatory only if we are representing different tokens and that do not use operators and separators. For example **int   x,y,   z;** is identical to **int x,y,z;**. But **intx,y,z;** is erroneous because **int** is a special keyword and it must be separated in some way by whitespace or may be by some separators if permitted.

Identifiers

Identifiers are the words that describe variable's name, method names, class names etc. An identifier can be any string of letters lower and upper characters, dollar sign ($), underscore (_), and digits with the exception that it cannot start with digits. Remember java is case sensitive and identifier **radius** is different from **Radius**. For example,
Hello, hi, hi123, $wer, _dsf, sd_ew, etc. are valid identifiers.
1hi, hello Mr, Mr.X, hello-dude, etc. are not valid identifiers.
Note: We cannot use keywords as identifiers.

Literals

Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values. Java language specifies five major types of literals namely **integer**, **floating point**, **character**, **string**, and **boolean**. For example,
75 is an integer literal
56.89 is a floating point literal
'y' is a character literal
"hello" is a string literal
true is a boolean literal.

Comments

Comments are sentences that are not treated as a part of the program. In java there are three ways of putting comments on the program codes. They are:

- Single line comments begins with // and end at the end of the line. For example,
  // This is a simple Java program
- Multi line comment is enclosed in /* …… */. For example,

/* This is a simple Java
program */

- Documentation comment is enclosed in /** …….. */. This comment is used to produce documentation using **javadoc** tool. For example,
  /** This is a simple Java
  program */

Separators

Separators are symbols used to indicate where groups of code are divided and arranged. There are different separators in java with different intention. The separators we have in java are:

( ) **Parenthesis:** It is used to enclose list of parameters in method definition and invocation. It is used to define precedence in an expression. It is used to enclose type cast.

[ ] **Brackets:** It is used to declare an array and also used to dereference array values.

{ } **Braces:** It is used to put values of automatically initialized array and to define the block of code for classes, methods, and local scopes.

; **Semicolon:** It is used to terminate the statements.

, **Comma:** It is used to separate consecutive identifiers in variable declaration.

. **Period:** It is used to separate package name from sub-packages and classes and is also used to reference variables and methods of an object.

Keywords

Keywords are the words that have distinct and special meaning in Java language and treated in special way by java compiler. So we cannot use keywords as identifiers. Apart from the below listed keywords **true, false** and **null** are reserved words and cannot be specified as an identifier though they are not keywords. The java keywords are listed below:

Abstract	Continue	For	new	switch
Assert	Default	Goto	package	synchronized
Boolean	Do	If	private	this
Break	Double	Implements	protected	throw
Byte	Else	Import	public	throws
Case	Enum	Instanceof	return	transient
Catch	Extends	Int	short	try
Char	Final	Interface	static	void
Class	Finally	Long	strictfp	volatile
Const	Float	Native	super	while

Primitive Data Types

Java is strongly typed language so every variable in java must be declared of specific type explicitly so that the compiler understands what kind of variable we are considering. A data type determines the values it may contain, plus the operations that may be performed on it. A data type also defines the amount of memory that will be used when defining the data type and the valid range of values it can represent. There are eight primitive data types in java. The eight types constitutes four groups namely **integers**, **floating points**, **characters** and **boolean**.

Integer Data Types

Integers represent the numbers without the fractional part i.e. whole numbers. In Java four data types represent integers: **byte**, **short**, **int**, and **long**. The following table shows the memory required for each type and the valid range of values it can represent.

Integer Type	**Memory Required**	**Range of Values**
Byte	1 byte (8 bits)	–128 to +127 (–2^7 to 2^7–1)
Short	2 bytes (16 bits)	–32,768 to 32,767 (–2^{15} to 2^{15}–1)
Int	4 bytes (32 bits)	–2,147,483,648 to 2,147,483, 647 (–2^{31} to 2^{31}–1)
Long	8 bytes (64 bits)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (–2^{63} to 2^{63}–1)

You choose the particular type of integers according to your need. For example if you need age of man in years to be represented, byte is the suitable one since we do not expect age of the man to be more than 127.
We declare Integer Types as:

int *x;* ***byte*** *num;* ***short*** *snum;* ***long*** *lnum;*

We can initialize the above types while declaring or afterwards. In any case literals are used for initialization. For example:

```
int second = 5, minute;        /*here two variables are declared where
                               one is initialized
minute = 30;                    and the other is not. The other variable
                               minute is initialized after declaration
                               statement*/
```

Default value for these types, if not initialized is 0 for all types with usual literal conventions. There are 3 types of literal representations for integer types they are decimal literals like 10 and -34, octal literals with leading zero like 012 (decimal 10), and hexadecimal literals with leading zero x like 0x1A (decimal 26). Here if we use literal for long integer we append the letter L at the normal literal for e.g. ***long*** *howlong* = 2147483648L.

Floating-Point Types

The types of numbers with the fractional parts are referred to as floating-point types. A common way in mathematics, as well as in computer science, to represent floating-point numbers is to provide an exponential representation of a number. This is defined as listing the significant digits with one digit written before the decimal point, and then defining the power of 10 to multiply by the number to generate its real value. So, 123 million could be written as 1.23E+8. We have two types of floating-point data types **float** and **double**. The following table shows the memory required for each type and the valid range of values it can represent.

Floating - point Type	**Memory Required**	**Range of Values**
Float	4 bytes (32 bits)	+/– 3.40282347E+38
double	8 bytes (64 bits)	+/– 1.7976931346231570E+308

The choice of the above type depends upon the precision we need. As an instance if you are representing currency, float will be ok, but if you are calculating the amount of light to apply to a surface rendering in graphics application, a double becomes more appropriate.
We declare Floating-Point Types as:
float *floatval*; **double** *doubval*;
We can initialize the above types while declaring or afterwards as defined previously for integer types.
Default value for these types, if not initialized is 0.0 for both types with usual literal conventions. Here if we use literal for float we append the letter f or F at the number. For example, **float** *floatliteral* = 12.3456f, *floatlit2* = 1.234e4F. Similarly we can append d or D for double type (for double type d or D may be omitted). There are three special values for floating type numbers **Double.POSITIVE_INFINITY**, **Double.NEGATIVE_ INFINITY** and **Double.NaN** (also corresponds to **Float** class) to represent positive infinity, negative infinity and not a number respectively.

Character Type

The char data type is a single 16-bit Unicode character to represent text. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive). We declare character type as: **char** *mychar*; Characters can be represented in their Unicode form as: \uXXXX, Where XXXX is a number in the range of 0000 to FFFF (hexadecimal), or as their English equivalents delimited by single quotes: like ‘a’, ‘x’, etc. If we need group of characters we normally use String (discussed later) that is not a primitive type. The Java programming language also supports a few special **escape sequences**. They are as shown in table below:

Character	**Meaning**	**Unicode Equivalent**
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000a
\r	Carriage Return	\u000d
\”	Double Quote	\u0022
\’	Single Quote	\u0027
\\	Back Slash	\005c

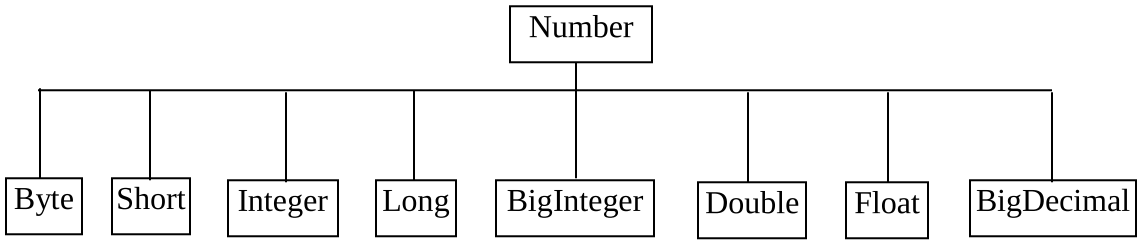
Example of character declaration and initialization:
char *mychar* = 'x';
char *notmychar* = '\t';
Note: We can perform arithmetic operations on char type variables.

Boolean Data Type

The last primitive data type in the Java programming language is the ***`boolean`*** data type. A `boolean` data type has one of two values: ***`true`*** or ***`false`***. These are not strings, but reserved words in the Java programming language.
The declaration and initialization of Boolean variable is as: ***boolean*** *boolval* = ***false****;*
Note: your compiler may generate error message if you do not initialize Boolean variable.

Numbers

We can use a number of classes in Java to work with numbers. The following figure shows the classes that can be used to work on with numbers. At the top is the abstract class Number.



These classes are called wrapper class. Whenever we need to have an object for primitive type data we can use wrapper classes so that we can take advantage of the capabilities that can be provided by classes like using different methods for converting values to other types like primitive types, Strings, etc and using defined variables like **MIN_VALUE** and **MAX_VALUE**.
The two other classes in the figure **BigInteger** and **BigDecimal** (in **java.math** package) provide the capability of defining arbitrary precision numeric values. For example,

```
import java.math.*;
public class Test
{


        public static void main(String[] args)
        {

                BigInteger bi1, bi2, bi3;
                bi1 = new BigInteger("123");
                bi2 = new BigInteger("50");
                bi3 = bi1.add(bi2);
                String str = "Result of addition is " + bi3;
                System.out.println( str );
        }
}
```

Variables

When we declare a data type it defines the storage capacity and usage for a region in memory. To access such memory in Java we assign a meaningful valid identifier and that identifier is called a variable. The act of creating a variable is called declaring a variable, and it has the following form:

datatype variableName1 [= value1, varaibleName2[= value2], …..] ; {here variable names may be chained by using comma as a separator and entities inside [and] are optional}.

For example:

char *c*;
int *numberOfStudents*;
int *a=5,b;*

The convention adopted by the Java world in naming variables is to start with a lowercase letter, and if the name has multiple words, capitalize the first letter of every word. For example: `myVaraible, hisChair, redHat. If we are naming class, the convention is to start with capital letter like MyClass.`
`In java there are four types of variables as given below:`

Instance Variables (Non-Static Fields)

Objects store their state in fields. Objects actually store their individual state in the fields that are declared as non-static. These types of fields are also known as instance variables. For e.g. if we have class name Students, then studentName can be instance variable.

Class Variables (Static Fields)

When a variable is declared using static modifier, then the resulting variable is called class variable. This variable represents the whole class and is common to all objects of the class in which the variable is declared as static. For example, numberOfStudents in class Student can be static since it represents the whole class.

Local Variables

There is no keyword that defines the local variable. However, the place where the variable is declared makes the variable local and the place is inside the method or block inside the method. Local variables are only visible to the method or block in which they are declared; they are not accessible from outside the method or block.

Parameters

The variable defined within the parenthesis in a method is the parameter. As an example **args** in **public static void main(String[]args)** is the parameter to this method.

Constants

To declare a variable to be a constant, or not changing, Java uses the keyword **final** to denote that a variable is in fact a constant. For example, we know that the value of PI is constant so we can declare PI with the keyword `final` and ensure that its value cannot change as ***final double*** *PI* = 3.14285;
If you try to compile a statement that modifies a constant value, such as PI with
PI = *PI*2*;
The compiler generates an error saying that you cannot assign a value to a final variable. By convention, programmers usually capitalize all the letters in a constant's name so that it can be found with only a quick look of the source code. Pi, therefore, was declared as **final double** *PI* = 3.14285;

Conversion between Numeric Types

Automatic Type Promotion in Expressions

When using numeric operands in expressions, the type does not necessarily have to be the same. Whenever two incompatible operands are operated then there will be automatic type promotion as follows:

- If any of the operands is **double**, the entire expression is promoted to **double**.
- If one operand is **float**, the entire expression is promoted to **float**.
- If one operand is **long**, the whole expression is promoted to **long**.
- All **byte** and **short** values are promoted to **int**.

`This type of conversion` is safe and does not result in the loss of information.

Conversion through Assignment

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

The rule for conversion between primitive numeric types is that a type can be assigned to a compatible type or a wider type, but not a narrower type. So, the following conversions are permissible by the compiler:

- `byte – short, int, long, float, double`
- `short – int, long, float, double`
- `int — long, float, double`
- `long — float, double`
- `float – double`
- `char – int`

Example: **int** *i* = 10; **long** *l* = *i*; // legal **double** *d* = *i*; // legal **short** *s* = *i*; // Illegal

Casting Incompatible Types

To create a *narrowing conversion* between two incompatible types, we must use cast. Casting tells the compiler to convert the data to the specified type even though it might lose data. Casting is performed by prefixing the variable or value by the desired data type enclosed in parentheses:
datatype *variable* = (***datatype***)*value*;

For example: **int** *i* = 10; short *s* = (**short**)*i*;
long *l* = 100;   **byte** *b* = (**byte**)*l*;
This type of casting must be carefully done since it may result in loss of data.

Operators

An operator is a symbol that is used to perform some action on one, two or three operands. The operators that perform on one operand are called **unary operators**, that perform on two operands are called **binary operators**, and that perform on three operands are called **ternary operators**. The java operators and their corresponding associativity are as given in table below where high precedence operators appear before low precedence operators and operators within the same group have same precedence.

Operator	**Description**	**Associativity**

`[]`
`.`
`()`
	Array element reference
	Member selection
	Method call
		Left to Right

`++` `--`	Post increment Post decrement	Left to Right

`~`
`!`
`-`
`+`
`++`
`--`
`(type)`
`New`
	Bitwise NOT
	Logical NOT
	Unary minus
	Unary plus
	Pre increment
	Pre decrement
	Casting
	Object and array creation
		Right to Left

`*`
`/`
`%`
	Multiplication
	Division
	Modulus
		Left to Right

`-` `+`	Addition Subtraction	Left to Right

`<<`
`>>`
`>>>`
	Bitwise shift left
	Bitwise shift right
	Bitwise shift right zero fill
		Left to Right

`<`
`<=`
`>`
`>=`
`Instanceof`
	Less than
	Less than or equal to
	Greater than
	Greater than or equal to
	Type comparison
		Left to Right

`==`
`!=`
	Equal to
	Not equal to
		Left to Right

`&`	Bitwise AND	Left to Right
`^`	Bitwise Exclusive OR	Left to Right
`|`	Bitwise Inclusive OR	Left to Right

`&&`	Logical AND	Left to Right
`||`	Logical OR	Left to Right
`? :`	Conditional Operator	Right to Left

`=`
`op = {*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=}`
	Simple assignment
	Shorthand assignment
		Right to Left

The [], ( ) and . operators

- **[]** operator is used for accessing the array elements at particular position. For e.g. if we have ***array1[5]*** then we are accessing 6th element of an array called ***array1***.
- **( )** operator is used for *calling methods*. For e.g. if static method of Math class called min, then we can have ***Math.min(7, 5)***. Similarly **( )** operator is used as *cast operator* for type casting as discussed previously. For e.g. if we have a variable x of double type and we want to cast it to the int type we must cast the variable x as ***(int)x***.
- **.** operator is used for accessing members of a class. For e.g. ***Math.PI***

The ++ and -- operators

The above two operators are unary operators. The ++ operator is called increment operator and the -- operator is called decrement operator. The increment operator increases its operand by 1 and the decrement operator decreases its operand by 1. For example, the statement:

x = x + 1;

can be rewritten like this by use of the increment operator:

x++; or ++x;

Similarly, the statement:

x = x – 1;

is equivalent to

x--; Or --x;

The increment/decrement operators can be applied before (prefix) or after (postfix) the operand. The code `x++;` and `++x;` will both end in x being incremented by one. The only difference is that the prefix version (`++x`) evaluates to the incremented value, whereas the postfix version (`x++`) evaluates to the original value. If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference. For example,

```
class IncDec {
    public static void main(String args[]) {
        int a = 1, b = 2, c, d;
        c = ++b;
        d = a++;
        c++;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
```

```
            System.out.println("d = " + d);
        }
}
```

Output:

```
a = 2
b = 3
c = 4
d = 1
```

The new Operator

The **new** operator is used to instantiate the class; that is, used to create an object of the class. This operator is also used to create an array (more on this later).

Arithmetic Operators

There are five arithmetic operators in Java. All the arithmetic operators are given below:

- **/** operator for division. This operator performs integer division (rounded) on integer types i.e. if both the operands are integers.
- ***** operator for multiplication
- **+** operator for addition (string concatenation and unary plus also)
- **-** operator for subtraction (unary minus also)
- **%** operator for obtaining remainder after integer division (also called **modulus** operator).

Examples:

7.5/5 = 1.5 7/5 = 1 2+3 = 5 3-2 = 1 5%3 = 2.

Assignment Operators

There are different assignment operators in java as provided in table above. These operators are given below:

- = operator is used to assign value of an expression to another expression. For e.g. if we want to assign value 5 to the variable five, then we do ***five = 5;***, similarly, ***str = "hello";*** assigns string value "hello" to the variable str.
- **Arithmetic assignment operators (+=, -=, *=, /=, %=):** += operator is used to assign variable a value obtained by adding the previous value of the variable and value of the other expression. For example res += res1; is equivalent to the statement res = res + res1; where res and res1 are two expressions. Other assignment operators are similarly interpreted.

Relational and Equality Operators

Relational and equality operators are very important in any programming. They are used for comparing the operands. The outcome of using these operators is a **boolean** value. Java has various such operators as given below. The first two are equality operators and the other operators are relational operators.

- **==** operator is used to verify whether two operands are equal.
- **!=** operator is used to verify whether two operands are not equal

- **>** operator is used to verify whether the first operand is greater than the second operand or not.
- **>=** operator is used to verify whether the first operand is greater than or equals to the second operand or not.
- **<** operator is used to verify whether the first operand is less than the second operand or not
- **<=** operator is used to verify whether the first operand is less than or equals to the second operand or not.
- **instanceof** operator is used for type comparison.

Example piece of code:

```
int a = 23;
int b = 23;
int c = 15;
if(a == b) System.out.println("a == b");
if(a != c) System.out.println("a != c");
if(a > c) System.out.println("a > c");
if(a < b) System.out.println("a < b");
if(a <= b) System.out.println("a <= b");
```

Output:

```
a ==  b
a != c
a > c
a <=b
```

Using instanceof:

Instanceof operator compares the object to the specified type. For e.g. if we have classes say *Parent* and *Child*, where Child class extends Parent and implements *MyInterface*. The following piece of code is interpreted as given below in output.

```
Parent obj1 = new Parent();
Parent obj2 = new Child();
System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
```

Output:

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

Note: ***null*** is not an instance of anything.

Logical Operators

The logical operators operate on Boolean operand(s) and results the value as true or false. See appendix for truth tables. Logical operators in java are as given below:

- **!** **logical NOT** operator is used to negate the boolean value i.e. if the value of the Boolean variable is **true** then its negation will be **false**..
- **&&** **logical AND** operator outputs the result true if both the operands on which the operator is operating are true, false otherwise.
- || **logical OR** operator outputs the result false if both the operands on which the operator is operating are false, true otherwise.

Usage Example:

if(((x= =6)&&(y!=4))||(!z)) { //code to do something.} here the expression inside if is true when we have either x is equal to 6 and y is not equal to 4 or z is false.

Bitwise Operators

The purpose of bitwise operators is to operate on the given integral operand (**long**, **int**, **short**, **char**, and **byte**) in low level i.e. bit level to represent the data. See appendix for truth tables. The following are the bitwise operators in java.

- ~ operator is used to negate the individual bits of the operand. For example,
- & operator does the ANDing of individual bits and produces output. The ANDing of two bit results in 1 if both the bits are 1, 0 otherwise.
- | operator does the ORing of individual bits and produces output. The ORing of two bits results in 0 if both the bits are 0, 1 otherwise.
- ^ operator does the XORing of individual bits and produces output. The XORing of two bit results in 1 if exactly one of the bits is 1 and the other is 0, 0 otherwise.
- << The left shift operator shifts all of the bits in a value to the left a specified number of times. The general form is: **value << num**, where *num* specifies the number of positions to left-shift the value in *value*. The top (leftmost) bits due to left shift are lost and the rightmost bits shifted are filled in with 0's.
- >> The right shift operator shifts all of the bits in a value to the right a specified number of times. Its general form is: **value >> num**, where *num* specifies the number of positions to right-shift the value in *value*. The top (leftmost) bits due to right shift are filled in with previous content of the top bit. This is called ***sign extension*** that serves to present the sign of the number shifted.
- >>> Unsigned right shift (shift right and zero fill) operator shifts all of the bits in the first operand to right a specified number of times given as a second operand. The leftmost bits due to shifting are 0.
- **Bitwise assignment operators (&=, ^=, |=, <<=, >>=, >>>=):** &= operator is used to assign variable a value obtained by ANDing the individual bits of the previous value of the variable and value of the other expression. For example res &= res1; is equivalent to the statement res = res & res1;, where res and res1 are two expressions. Other assignment

operators are similarly interpreted.

Examples: In the examples below, we assume 8 bit (*byte*) values.

- ~2 = ~(00000010) = 11111101 = -3.
- ~-2 = ~(11111110) = 00000001 = 1.
- 5&2 = 00000101&00000010 = 00000000 = 0.
- -5&-2 = 11111011&11111110 = 11111010 = -6.
- 5|2 = 0000 0101 | 0000 0010 = 00000111 = 7.
- -5|-2 = 11111011|11111110 = 11111111 = -1.
- 5^2 = 00000101 ^ 0000 0010 = 00000111 = 7.
- -5^-2 = 11111011^11111110 = 00000101 = 5.
- 35>>2 = 0010 0011 >> 2 = 0000 1000 = 8.
- -5>>1 = 11111011>>1 = 11111101 = -3.
- 35>>>2 = 0010 0011 >> 2 = 0000 1000 = 8.
- -5>>>1 = 11111111111111111111111111111011 >>> 1 = 0111111111111111 1111111111111101 = 2147483645 (this operator is only meaningful for 32 and 64 bit values).
- 35<<2 = 00100011 << 2 = 10001100 = 140.
- -5<<1 = 1111 1011<<1 = 1111 0110 = -10.

Conditional Operator (?:)

This operator is ternary operator i.e. it requires three operands. The syntax for this operator is ***condition ? statement1:statement2***. Here we interpret like: if condition is true then statement 1 is executed else statement 2 is executed. For e.g. ***int min = (x <= y) ? x : y;*** means if x <= y then min = x, otherwise min = y.

Associativity

If we come up with the situation where operators are from the same precedence group then the order of evaluation is provided by associativity. For example, if we have `3 * 5 % 3`, what is the result `(3 * 5) % 3` or `3 * (5 % 3)`? First expression is `15 % 3`, which is `0`. The second expression is `3 * 2`, which is `6`. Here, since * and the % operators have the same precedence, precedence does not give the answer. So we take the advantage of associativity and calculate the expression `3 * 5 % 3` as `15 % 3` since * and % are left to right associative. To come out of this kind of confusing situation, we can use parenthesis to define precedence.

Expressions

Expressions are the logical constructs consisting the combination of different constructs such that it results in single value. Some examples are:

24 (a literal)
this (this object reference)
Double.MAX_VALUE (field access)
Math.sqrt(16) (a method call)
new String("hello") (object creation)
new arr[2] (array creation)

arr[1] (array access)
x+y (expression with operators)
(x*200+3) (Expression in parenthesis).

Statements

Statement is set of constructs that make an executable portion of program. In java, most of the statements are ended with semicolon (;). Different types of statements in java are:

- **Expression Statements:** Statement that consists of expression and ended with semicolon. For example,

Assignment expressions	e.g. a = 3;
Any use of ++ or --	e.g. x++;
Method invocations	e.g. Math.pow(2,3);
Object creation statement	String str = new String("Oops!");

- **Declarative Statements:** Statements declaring the variables. For e.g. int x = 101;
- **Control Flow Statements:** These statements define the order of execution. For example, **if**, **switch**, **for**, **while**, **do-while**, **break**, **continue** and **return**.(detail later)

Blocks

Block is a compound statement, or a group of statements enclosed within braces ({ … }). The following is the example of block.

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

Control Flow Statements

The three types of control flow statements in Java are: conditional statements, iteration statements, and jump statements.

Conditional Statements

Conditional statements allow us to decide a statement of code or a block of code to execute depending upon the given condition. These statements are also called **selection statements** or **decision making statements**. Java supports two conditional statements: **if** and **switch**.

The if Statements

This type of statements are used when we have to select a statement or a block of statements if certain condition is satisfied and take some other action if the condition is not satisfied. There are different forms of if statements as provided below:

The *if only* Statement

Consider the situation when we have to perform an action only if certain condition is matched and do nothing if the condition is not matched. This situation can be best described by if only statements. The syntax for if only statement is

```
if( condition )
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```

If the condition is true then statements in block are executed else nothing is done. For example,

```
if(amount >= 1000)
        discount = amount * 0.05;
```

The *if-else* Statement

When there is a need of executing a statement (or block) if a condition matches and another statement (or block) is to be executed otherwise, then we use if – else statement. It has the general syntax as given below:

```
if (condition)

{
        Statement 1
        Statenebt 2
        ……………
        Statement n

}
else
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```

In this situation, if the condition is satisfied the first block is executed, otherwise the second block is executed. For example,

```
if(amount >= 1000)
        discount = amount * 0.05;

else

        discount = amount * 0.03;
```

The if else statement resembles with the conditional operator (?:). Conditional operator is a ternary operator (demands 3 operands), and is used in certain situations, replacing if-else statement. Its general form is:

exp1 **?** *exp2* **:** *exp3;*

If **exp1** is true, **exp2** is executed and whole expression is assigned with value of **exp2**. If the exp1 is false, then **exp2** is executed, and whole expression is assigned with **exp2** value. For example,

c = a > b ? a+b : a-b;

is equivalent to

```
if(a>b)
        c=a+b;
else

c=a-b;
```

The if-else-if ladder Statement

If we have the situation where there are different actions that are executed depending upon different condition with same type of instance then if-else-if is useful. It has general syntax as given below:

```
if (condition 1)

{
        Statement 1
        Statenebt 2
        ……………
        Statement n

}
else if (condition 2)
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
…………………………
else if ( condition n)
{
        Statement 1
        Statenebt 2
        …………
        Statement n
}
else
{
        Statement 1
        Statenebt 2
        …………
        Statement n
}
```

Here if condition1 is true first block of statements is executed, otherwise if condition2 is true second block of statements is executed, for true condition n, nth block of statements is executed and final block of statements is executed if no condition matches. For example,

```
if(amount >= 5000)
        discount = amount * 0.1;
else if (amount >= 4000)
        discount = amount * 0.07;
else if (amount >= 3000)
        discount = amount * 0.05;
else
          discount = amount * 0.03;
```

The *switch* statement

It is a multiple branch selection statement, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statement(s) associated with that constant are executed. Its syntax is:

```
switch                                                    (expression)
{
case constant1:
        statement(s)
        break;
case constant2:
        statement(s)
        break;
   ...
case constantn:
        statement(s)
        break;
default:
        statement(s)
}
```

The expression must evaluate to an integer type. The value of expression is tested against the constants present in the case labels. When a match is found, the statement sequence, if present, associated with that case is executed until the break statement or the end of the switch statement is reached. The statement following default is executed if no matches are found. The default is optional, and if it is not present, no action takes place if all matches fail. For example,

```
import java.util.*;
public class SwitchTest
{
	public static void main(String[]args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a day:");
		int day = sc.nextInt();
		switch(day)
		{
			case 1:
				System.out.print("Sunday");
				break;
			case 2:
				System.out.print("Monday");
				break;
			case 3:
				System.out.print("Tuesday");
				break;
			case 4:
				System.out.print("Wednesday");
				break;
			case 5:
				System.out.print("Thrusday");
				break;
			case 6:
				System.out.print("Friday");
				break;
			case 7:
				System.out.print("Saturday");
				break;
			default:
				System.out.print("Wrong day!");
		}
	}
}
```

Things to remember with switch:

- A switch statement can only be used to test for equality of an expression. We cannot check for other comparisons like <, <=, >, >=
- switch expression must evaluate to an integral value
- No two case constants can be same
- Omission of a break statement causes execution to go to next case label
- The default label is executed when no case constants matches the expression value

Iteration Statements

When there is a need of executing a task a specific number of times until a termination condition is met, we use iteration statements. These statements are also called **looping statements** or **repetitive statements**. Java provides three ways of writing iterative statements. They are `while` *statement*, `do-while` *statement*, and `for` *statement*.

The *while* Statement

The `while` statement executes the statements as long as the given condition remains true. The general syntax of while statement is:

```
while(condition)
{
        Statement 1
        Statenebt 2
        …………
        Statement n
}
```

Here the statements within the brace are executed as long as the condition is `true`. When the `condition` becomes `false`, the `while` loop stops executing these statements and exits out of the loop. For example,

```
int i=1;
while( i <= 10 )
{
        System.out.println( "i=" + i );
        i++;
}
```

Remember: Do not forget to increment i, otherwise loop will never terminate i.e. code goes into infinite loop. So when using loop, remember to guarantee its termination.

The *do while* Statement

The do while loop always executes its body at least once. So, to guarantee, at least one time, execution of statements do while loop is used. The basic syntax of do while loop is:

```
do
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
} while(condition);
```
Here the statements inside the braces are executed at least once, because its condition is at the bottom of the loop. For example,
```
int i=1;
do
{
        System.out.println( "i=" + i );
        i++;
} while( i <= 10 );
```

The *for* Statement

When we have the fixed number of the iteration known then we use for loop (normally, while loop is also possible). The basic syntax of `for` statement is:
```
for(initialization; condition; increment/decrement )
{
        Statement 1
        Statenebt 2
        ……………
        Statement n
}
```
First before loop starts, the initialization statement is executed that initializes the variable or variables in the loop. Second the condition is evaluated, if it is true, the body of the `for` loop will be executed. Finally, the increment/decrement statement will be executed, and the condition will be evaluated, this continues until the condition is false. For example,
```
for(int  i=1;i <= 10; i++)
        System.out.println( "i=" + i );
```
Remember: Variable declaration in initialization part of the loop is permissible and generally is a good practice. If you have already declared and initialized the variable you could leave the initialization section blank as shown below
```
int i = 1;
for(;i <= 10; i++)
        System.out.println( "i=" + i );
```
We can also include more than one statement in the initialization and increment/decrement portions of the for loop. For example,
```
int a, b;
for (a = 1, b = 4; a < b; a++, b--)
{
        System.out.println(“a = “ + a);
        System.out.println(“b = “ + b);
}
```

Note: The three expressions of the `for` loop are optional; an infinite loop can be created as follows:

```
for ( ; ; )
{
        Statement 1
        Statenebt 2
        …………
        Statement n
 }
```

Comparison of Three Loops

Depending on the position of the test expression (condition) in the loop, a loop may be classified either as the **entry-controlled loop** or **exit-controlled loop**.

In the entry-controlled loop, the test expressions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. Examples are *while* and *for* loops.

In case of an exit-controlled loop, the test expression is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time and conditionally after that. Example is *do – while* loop.

Nested Control Flow Statements

We can nest any control flow statement (conditional and iteration statement) within another. When one loop is nested inside another loop, the inner loop is first terminated and again restarted when the first loop starts for the next incremented value. The outer loop is terminated last. For example,

```
for(int i = 1; i<=2;i++)
{
        System.out.println(i);
        for(int j=1; j<=3;j++)
                System.out.println(j);
}
```

Jump Statements

Java, for the transfer of control from one part to another, supports three jump statements: **break**, **continue**, and **return**.

The *break* Statement

The use of break statement causes the immediate termination of the loop (all type of loops) and switch from the point of break statement and the passage program control to the statements following the loop. In case of nested loops if break statement is in inner loop only the inner loop is terminated. For example,

```
for( int i=1; i<= 10; i++ )
{
        if( i == 5 )
                break;
        System.out.print( " i=" + i );
}
```

The *continue* Statement

Continue statement causes the execution of the current iteration of the loop to cease, and then continue at the next iteration of the loop. For example,

```
for( int i=1; i<= 10; i++ )
{
        if( i == 5 || i == 7 )
                continue;
        System.out.print(i );
}
```

The *return* Statement

It is used to transfer the program control back to the caller of the method. There are two forms of return statement. First with general form ***return expression;*** returns the value whereas the second with the form ***return;*** returns no value but only the control to the caller. For example,

```
class ReturnTest
{
        public static int add(int a, int b)
        {
                return a + b;
        }
        public static void main(String[]args)
        {
                 int a = Integer.parseInt(args[0]);
                 int b = Integer.parseInt(args[1]);
                 int c = add(a, b);
                System.out.print("Sum = " + c);
        }
}
```

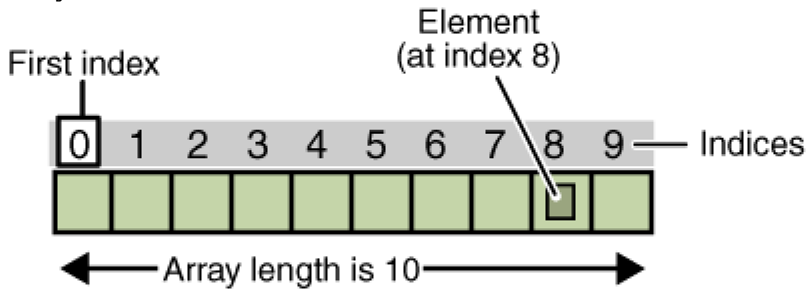
Arrays

An array is a group of contiguous liked-typed variables that are referred to by a common name. The length of an array is established when the array is created. After creation, its length is fixed. Each item in an array is called an *element*, and each element is accessed by its numerical *index*.

Array offers a convenient means of grouping related information since array is the collection of similar type of data elements. Arrays of any type can be created and may have one or more dimensions.

One-Dimensional Arrays

In one-dimensional arrays, a list of items can be given one variable name using only one subscript. This type of array is also called *single-subscripted* variable. To access an element in this array we use single index (subscript) value and in Java this index value starts with 0. Hence, the 9th element is accessed at index 8. The figure below shows an array of 10 elements.



Declaration

The general form of a one-dimensional array declaration is
type[] arrayName; **or** *type arrayName [];*
For example,
int intArray []; **or** *int [] intArray;*
We can declare any type of array i.e. it may be the array of objects like
String [] srtingArray;
or it may be array of primitive types as previously shown.
We can declare several arrays at the same time as follows:
int[]nums1, nums2, nums3; **or** *int nums1[], nums2[], nums3[];*

Creation

After declaring an array, we need to create it in the physical memory. Java allows us to create arrays using **new** operator only, as shown below:
arrayName = new type[size];
For example,
intArray = new int[10];   //array of int with 10 elements
Note: It is also possible to combine the two steps – declaration and creation – into one as shown below:
int[]intArray = new int[10];

Initialization

When an array is created we can initialize it after the creation or within the declaration itself. Within the declaration, we can initialize it as shown follows:
int [] intArray = {10, 20, 31, 45, 56};
This is initialization at the declaration time and at this situation the size of the array is 5. Remember all the elements are enclosed within the braces separated by commas.
Initialization after the creation is done like this:
int [] intArray = new int[5];
intArray[0] = 10;
intArray[1] = 20;
intArray[2] = 31;
intArray[3] = 45;
intArray[4] = 56;

Accessing Array Elements

Each array element is accessed by its numerical index. For example,

```
int[]intArray = {10, 20, 31, 45, 56};
System.out.println(intArray[0]);
System.out.println(intArray[2]);
System.out.println(intArray[3]);
```

Output:

```
10
31
45
```

Array Length

In Java, all arrays store the allocated size in a variable named **length**. We can access the length of the array **intArray** using **intArray.length** and the value of **intArray.length** in the above example is 5.

Example

```
class Test
{
	public static void main(String[]args)
	{
		int[]nums = {8, 4, 15, 20, 7, 29};
		int sum = 0;
		for(int i = 0; i < nums.length; i++)
			sum += nums[i];
		System.out.print("Sum = " + sum);
	}
}
```

Output:

```
Sum = 83
```

Question:

Modify the above program such that we can provide inputs to array without assigning the values within array declaration. (**Hint:** You can either use command line input or run time input).

Copying Arrays

To copy the array we can take advantage of the method defined in the **System** class called **arraycopy** with the signature:

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

Here **src** is source array, **srcPos** is position from where we are copying, **dest** is destination array, **destPos** is position at the destination array to start copying and length is the number of element to be copied.

Example Piece of Code:

```
		int [] sourceArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
```

```
            int [] destArray = new int[5];
            System.arraycopy(sourceArray, 3, destArray, 0, 5);
            for(int i = 0; i < destArray.length; i++)
                        System.out.print(destArray[i]);
```
Output:

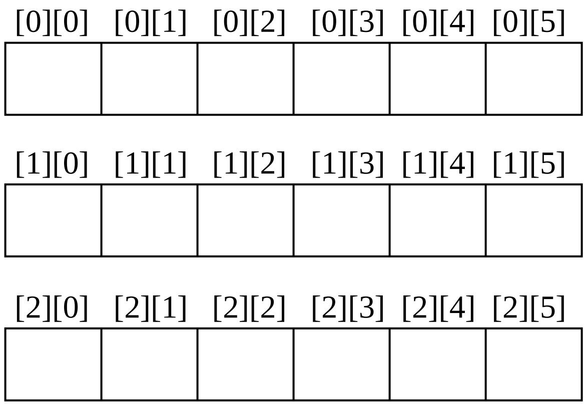
```
45678
```

Multi-Dimensional Arrays

Multidimensional arrays are actually arrays of arrays. The declaration, creation and initialization schemes are similar to the one-dimensional arrays but with subtle differences. For multidimensional arrays, we specify each additional index using another set of square brackets. For example, we can create, initialize and access two-dimensional array as follows:

```
int array2D [][] = new int [3][6];      //array creation
```

This code creates an array of dimension 3 by 6 as shown below.



We can initialize two-dimensional arrays as follows:

```
char charArray2D[2][3] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’};
```

Or equivalently,

```
char charArray2D[][] = {{‘a’, ‘b’, ‘c’}, {‘d’, ‘e’, ‘f’}};
```

To access the array we use the same subscript form but there is slight difference in that the two-dimensional array starts from inner subscripts to outer one. For example, charArray2D[0][0].

Example Piece of Code:

```
char charArray2D[][] = {{‘a’, ‘b’, ‘c’}, {‘d’, ‘e’, ‘f’}};
for(int i = 0; i < 2; i++)   //for inner subscripts
{
        for(int j = 0; j < 3; j++)   //for outer subscripts
                System.out.print(charArray2D[i][j]);
        System.out.println();
}
```

Output:

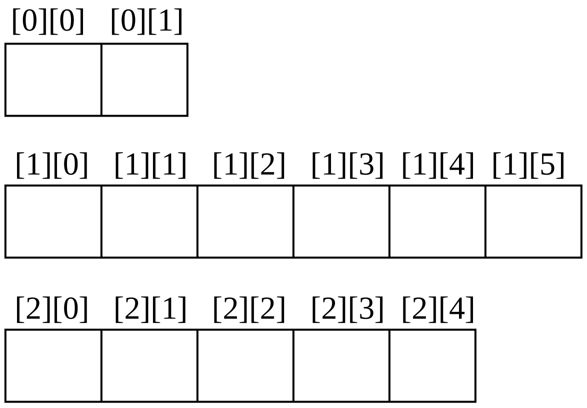
```
abc
```

def

When multidimensional array is created it is not necessary to define memory for all dimensions. The definition for the first (leftmost) dimension is sufficient. For example,

```
String str2D [][] = new String[3][];
str2D[0] = new String[2];
str2D[1] = new String[6];
str2D[2] = new String[5];
```

This code creates the following array.



Chapter 3
Classes and Objects
Defining Classes

As we have discussed earlier, a class is a framework that specifies what data and what methods will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. A class is also called user defined data type or programmers defined data type because we can define new data types according to our needs. The general syntax of the class definition is given below:

```
[Access][Modifiers] class className [extends superClass]
                 [implements interface1[, interface2][, ……]]{
      //body
}
```

<u>Components</u>:

- **Access:** public, private, and others discussed later.
- **Modifiers:** static, final and others discussed later.
- **Class name:** The class name, with the initial letter capitalized by convention.
- **Superclass:** The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class (subclass) can only extend one parent (superclass).
- **Interfaces:** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body generally contains variables and methods, surrounded by braces, {}.

Adding Variables:

Data is encapsulated in a class by placing data fields inside the body of the class definition. The general form of variable declaration is:

```
[Access][Modifiers] data-type variableName;
```

<u>Components</u>:

- **Access:** public, private, and others discusses later.
- **Modifiers:** static, final and others discussed later.
- **Data type:** type of the variable. It may be primitive or user defined.
- **Variable name:** name of the variable

Adding Methods:

To manipulate data contained in the class we add methods inside the body of the class definition. The general form is:

```
[Access][Modifiers] return-type methodName(ArgumentList)
                                             [exceptions lists]{
      //body
}
```

<u>Components</u>:

- **Access:** public, private, and others discussed later.
- **Modifiers:** static, final and others discusses later
- **Return-type:** the data type of the value returned by the method or void if the method does not return a value.
- **Method name:** valid identifier with convention of starting with small letter, first word as a verb and the later words start with capital letter.
- **The parameter list in parenthesis:** a comma-separated list of input parameters, preceded by their data types, enclosed by parentheses. If there are no parameters, you must use empty parentheses.
- **Exception list:** discussed later.
- **The method body, enclosed between braces:** the method's code, including the declaration of local variables, goes here.

Example:

```
class Rectangle {
      private int length;   //variable declaration
      private int breadth;  //variable declaration
      public void setData(int l, int b) {    //method definition
            length = l;
            breadth = b;
      }
      public int findArea() {     // method definition
            return length * breadth;
      }
      public int findPerimeter() {     // method definition
            return 2 * (length + breadth);
      }
}
```

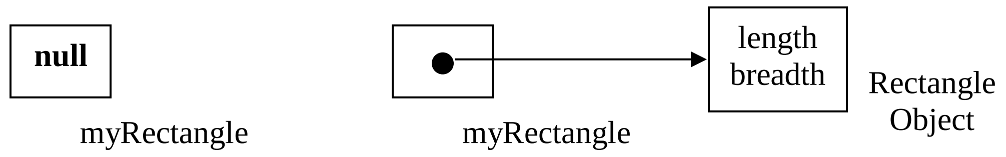
Creating Objects

Objects are created using the **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object. The main idea of using **new** is to create the memory that is required to hold an object of the particular type in run time i.e. to dynamically allocate the memory at the run time.

To create a usable object we must finish two steps: *declaring* the variable of its type and *instantiating* the object. The following example shows the creation of an object of type Rectangle.

```
Rectangle myRectangle; //declaring the variable of type Rectangle
myRectangle = new Rectangle(); //Instantiating an object
```

The execution of first statement creates the variable that holds reference of the class Rectangle using the name myRectangle. It points nowhere (i.e. null) as shown below:



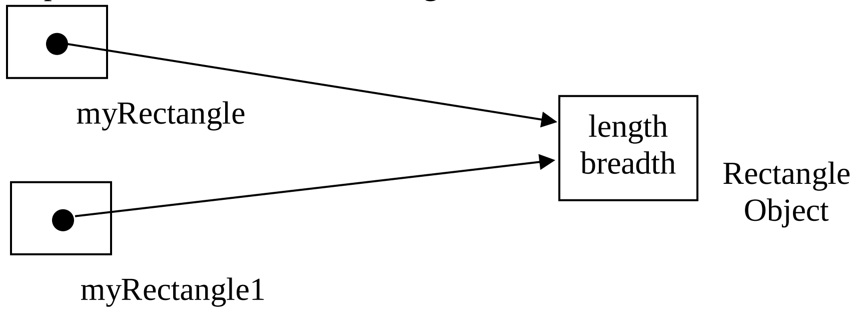
When the second statement is executed then the actual assignment of object reference to the variable is done as shown in above figure (right). The above two steps are equivalently written as:

```
Rectangle myRectangle = new Rectangle();
```

We can create more than one reference for same object and using different name we can manipulate same object. For example,

```
Rectangle myRectangle = new Rectangle();
Rectangle myRectangle1 = myRectangle;
```

The pictorial representation of above code segment is as below:



Though myRectangle and myRectangle1 reference the same object they are not linked to each other so if we assign some other object to one of the name, previous link is removed from that name. For example, after some lines of code from above if we write

```
myRectangle = null;
```

The name myRectangle now does not point to the rectangle object.

Using Class Members

When an object of the class is created then the members are accessed using the '.' dot operator as shown below:

```
myRectangle.setData(5, 2);
myRectangle.findArea();
myRectangle.findPerimeter();
```

Remember: Not all the members can be accessed from outside the class using the dot '.' operator. Here, we cannot access length and breadth from outside the class. (Why?)

Complete Example

```
class Rectangle {
     private int length;
     private int breadth;
     public void setData(int l, int b) {
          length = l;
          breadth = b;
     }
     public int findArea() {
          return length * breadth;
     }
```

```
      public int findPerimeter() {
            return 2 * (length + breadth);
      }
}
class MainRectangle {
      public static void main(String[]args) {
            Rectangle myRectangle = new Rectangle();
            Rectangle myRectangle1 = new Rectangle();
            myRectangle.setData(5, 2);
            myRectangle1.setData(9, 4);
            System.out.println("First rectangle:");
            System.out.println("Area = " +
myRectangle.findArea());
            System.out.println("Peremeter = " +
myRectangle.findPerimeter());
            System.out.println("Second rectangle:");
            System.out.println("Area = " +
myRectangle1.findArea());
            System.out.println("Peremeter = " +
myRectangle1.findPerimeter());
      }
}
```

Output:

```
First rectangle:
Area = 10
Perimeter = 14
Second rectangle:
Area = 36
Perimeter = 26
```

Setters and Getters

These are also called *set* and *get* methods. We use setters and getters to provide the interface to modify and use fields (private) within the class respectively. The program below shows the use of set and gets methods.

```
class Rectangle {
      private int length;
      private int breadth;
      public int getLength() { return length;} //getter
      public int getBreadth() {return breadth;} //getter
      public void setBreadth(int b) {breadth = b;} //setter
      public void setLength(int l) {length = l;} //setter
      public int findArea() {return length*breadth;}
      public int findPerimeter() {return 2*(length+breadth);}
}

class MainRectangle {
      public static void main(String [] args) {
            Rectangle rect1 = new Rectangle();
            rect1.setLength (10);
            rect1.setBreadth(5);
            System.out.println("Length: "+ rect1.getLength());
            System.out.println("Breadth: "+ rect1.getBreadth());
            System.out.println("Area: " + rect1.findArea());
            System.out.println("Perimeter: " +
rect1.findPerimeter());

      }
}
```

Output:

```
Length: 10
Breadth: 5
Area: 50
Perimeter: 30
```

Remember: If a method changes the value of variable(s), it is called **mutator method**. For example, set method in a class. The method that does not change but access value(s) is called **accessor** or **query method**. For example get method in a class.

Constructors

Initializing the values of the variables of the object is very time consuming process. Though we have set methods that can be used to assign values to the variables it is tedious too. So to remedy this complexity we can take advantage of constructor for providing values to the instance variables.

The constructor initializes the variables of the object immediately after its creation and it will be simpler for us if we initialize variables while creating the object. Constructor has same name as of class and looks like a method except that it has no return type.

We have already seen the use of the constructor in above programs as Rectangle() [default constructor]. Though we have used the constructor we have not defined it and if we do not define the constructor the default constructor is called automatically. And if we define constructor ourselves then there will be no default constructor so in case of our need of default constructor we have to create by ourselves. Since constructor looks like method we can also provide arguments (parameters) to the constructor. The below program is the modification of the above program using constructors.

```
class Rectangle {
      private int length;
      private int breadth;
      public Rectangle() { // default constructor
      }
      public Rectangle(int l, int b) { //parameterized
constructor
            length = l;
            breadth = b;
```

```
        }
        public void setLength(int l) {
            length = l;
        }
        public void setBreadth(int b) {
            breadth = b;
        }
        int findArea() {
            return length*breadth;
        }
        int findPerimeter() {
            return 2*(length+breadth);
        }
}

class MainRectangle {
        public static void main(String [] args) {
            Rectangle rect1 = new Rectangle(10, 5);
            Rectangle rect2 = new Rectangle();
            rect2.setLength (7);
            rect2.setBreadth(3);
            System.out.println("First rectangle");
            System.out.println("Area:" + rect1.findArea());
            System.out.println("Perimeter:" +
rect1.findPerimeter());
            System.out.println("\nSecond rectangle");
            System.out.println("Area:" + rect2.findArea());
            System.out.println("Perimeter:" +
rect2.findPerimeter());

        }
}
```

Output:

```
First rectangle
Area: 50
Perimeter: 30

Second rectangle
Area: 21
Perimeter: 20
```

Overloading Methods

In Java, we can define different methods with the same name but with different parameters (either parameter type or number of parameters). This is one of the examples of **polymorphism**. If we define many methods with same name but with difference in parameters then this process is called **method overloading**. In this case the return type of the method does not matter and can be same as previously defined. For example,

```
class Overloading {
      public static void main(String[]args)    {
            System.out.println(sum(5, 7));
            System.out.println(sum(5, 7, 2));
            System.out.println(sum(5.2f, 7));
            System.out.println(sum(5.3, 7.4));
      }
      static int sum(int a, int b){return a+b;}
      static int sum(int a, int b, int c) {return a + b + c;}
      static float sum(float a, int b){return a+b;}
      static double sum(double a, double b){return a+b;}
}
```

Output:

```
12
14
12.2
12.7
```

Remember: As we have already seen, constructor is much like method. So it is, most of the times, necessary to overload the constructor and this is possible with all the rules as defined for other methods. For example,

```
class Rectangle {
      public Rectangle() {
      }
      public Rectangle(int l, int b) {
            length = l;
            breadth = b;
      }
      //other methods and variables
}
```

Parameters to the Methods and Return Type

Methods take parameters that are used to provide the function inside the method. The number of parameters in the method is up to the nature of the method we are defining. For example, some method may not take any parameter, or it may take only one parameter, or two, and so on. In Java, method can be defined to take unknown number of parameters.

Java methods can take any type of data (primitive as well as objects) as parameters and return any type of value (primitive as well as objects). If the method takes parameter as primitive type then passing of argument from the caller is done as **call by value** and if the parameters are of object type the calling method is **call by reference**. The same process happens for the return type.

Call by value

Here, the actual value of the argument (actual parameter) is passed to the method. In this case, actual parameter is copied to the formal parameter of the method so that changes made to the parameter inside the method are not reflected to the actual parameter.

Call by reference

In this method, unlike the value, reference to an argument is passed to the formal parameter so that they refer to the same object location. In this process the modification to the parameter inside the method is reflected to the actual parameter.

Example:

```
class ParameterPass {
	public static void main(String[]args) {
		int a = 7;
		Rectangle r1 = new Rectangle(10,5);
		Rectangle.increment(a);   //method call with primitive
type parameter
		Rectangle.increment(r1);  //method call with object
type parameter
		System.out.println("a = " + a);
		r1.display();
	}
}

class Rectangle {
	private int length;
	private int breadth;
	Rectangle(int l, int b) {
		length = l;
		breadth = b;
	}
	static void increment(int a) {
		a++;
	}
	static void increment(Rectangle r1)	{
		r1.length++;
		r1.breadth++;
	}
	public void display() {
		System.out.println("Length = " + length);
		System.out.println("Breadth = " + breadth);
	}
}
```

Output:

```
a = 7
Length = 11
Breadth = 6
```

this Keyword

The keyword **this** can be used within any method to represent the current object i.e. **this** is the reference to the object on which method is getting invoked. For example,

```
class Rectangle {
      private int length;
      private int breadth;
      public Rectangle(int length, int breadth) {
            this.length = length;
            this.breadth = breadth;
      }
      .........
      .........
}
```

If we look at the constructor definition above we see the name **length** and **breadth** as both parameters and instance variables. Though normally in java declaring two variables within the same scope is illegal it is permissible to have same name as parameter of a method and instance variables. In this case, we use **this** keyword to identify the instance variables.

Access Control

We have different access specifiers namely **public**, **private**, and **protected**; there is also no modifier (**package-private**).

Access specifiers for a class (top level [not nested])

If we need our class to be visible to all other packages then we must declare it as public using "public" specifier. And if we need it to be visible in the same package only, we use no specifier. The use of other specifiers is not allowed. The term "visible" means you can use the accessible resources (call a method, read a field, whatever it is). Below we discuss both the cases:

```
package ku.kcm.bbis;
   public class II { //class body }    //public specifier
```

Class II will be visible to all packages everywhere.

```
package ku.kcm.bbis;
      class II {//class body}    //no specifier
```

Class II will be visible only to classes in the same package (means same directory) ku.kcm.bbis.

Access specifiers for members (methods, fields, and nested classes)

At member level we can use all the possible specifiers (private, public, and protected) and no specifier as well. Usage of the possible specifiers gives us four level of visibility. The below is the skeleton class describing all the possible specifiers.

```
package ku.kcm.bbis;
```

```
public class II {
        II() {//your codes}
        int rollNumber;
        private String name;
        protected String subject;
        public int getRollNumber(){return rollNumber;}
        .........
        .........
}
```

If the member has no specifier then it is visible to all the classes within the package. This is often called package access. So the *rollNumber* in the above skeleton is accessible within the classes inside *ku.kcm.bbis* package.

If the member has **public** specifier then it is visible to all the classes in all the packages. So the method *getRollNumber* in the above skeleton is accessible to all the packages.

If the member has **private** specifier then it is visible only inside the top level class and no where else. So the field *name* in the above skeleton is accessible to II class only.

If the member has **protected** specifier then it is visible only inside the package and in the subclass outside the package. So the field *subject* in the above skeleton is accessible to *ku.kcm.bbis* package and to all the classes inheriting class II. This specifier is used for the classes that can act as parent to open up visibility of their members to their child classes, but not to others.

Remember: If a class in a package is not public, then none of its members will be visible outside that package.

All about Static

The static modifier in java can be used in the following parts:

- Variables (class variables)
- Methods (class methods)
- Blocks – These are blocks within a class that are executed only once, usually for some initialization. They are like instance initializers, but execute once per class, not once per object.
- Classes – Classes nested in another class. (We will discuss it later)

Class variables

A class basically contains variables and a new copy of each variable is created when an object of that class is created or when that class is instantiated. These variables are called *instance variables*.

Let us assume that we want to define a variable that is common to all the objects and accessed without using a particular object if it is accessible. That is, the variable belongs to the class as a whole rather than the objects created from the class. For this, we have the solution that comes in form of the **static** keyword. When we use static modifier in variable declaration then such variables are called **class variables**. Here is the program that illustrates the use of the class variable.

```
class StaticVariable {
	public static void main(String []args) {
		Student st1 = new Student("John", 1);
		Student st2 = new Student("Peter", 2);
		Student st3 = new Student("Mike", 3);
		st1.display();
		st2.display();
		st3.display();
		Student.clas = "BBA"; //we can also use any object
with the same effect to access static variable. For example,
st1.clas = "BBA";
		st1.display();
		st2.display();
		st3.display();
	}
}
class Student{
	String name;
	int rollNo;
	static String clas = "BBIS"; //static variable
	Student(String n, int r){
		name = n;
		rollNo = r;
	}
	public void display() {
		System.out.println("Name:" + name);
		System.out.println("Roll Number:" + rollNo);
		System.out.println("Class:" + clas);
		System.out.println();
	}
}
```

Output:
Find out yourself.

Class Methods

If we need a class-wide method, then we declare a method using the static keyword so that it is not intended for an individual object. We have already seen many example of static method like main method, parseInt, sqrt etc. These methods are invoked with the class name like **ClassName.methodName(arguments);**

```
Integer.parseInt("22");    //static method of Integer class
```

However, we can use object to refer it (bad practice!!! makes no sense).
When dealing with the static methods you must remember these facts:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods cannot access instance variables or instance methods directly--they must use an object reference. Also, class methods cannot use **this** and **super** in any way.

Example:
```
class StaticMethod {
      public static void main(String []args) {
            Add a1 = new Add(5, 6);
            System.out.println("Sum:" + Add.add());
      }
}

class Add {
      private static int a, b;
      public Add(int aa, int bb) {
            a = aa;
            b = bb;
      }
      public static int add() {
            return a + b;
      }
}
```
Output:
```
Sum = 11
```

Static Blocks

If we need to do computation in order to initialize static variables, we can declare a static block, which gets executed exactly once, when a class is first loaded. Static blocks are also called **static initialization blocks**. A *static initialization block* is a normal block of code enclosed in braces, `{ }`, and preceded by the `static` keyword. Here is an example:

```
class StaticBlock {
      static int a = 3;
      static int b;
      static {
            System.out.println("Static block initialized");
            b = a * 4;
      }
      public static void main(String[]args) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
      }
}
```
Output:
```
Staic block initialized
a = 3
b = 12
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.

Note: There is an alternative to static blocks —you can write a private static method:

```
class Whatever {
   public static varType myVar = initializeClassVariable();
      private static varType initializeClassVariable() {
            //initialization code goes here
      }
}
```

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

Initializing Instance Members

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.

Initializer blocks for instance variables look just like static initializer blocks, but without the `static` keyword:

```
{
      // whatever code is needed for initialization goes here
}
```

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors. For example,

```
class InstanceBlock {
      int a;
      int b;
      {
            a = 3;
            b = a * 4;
      }
      void display() {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
      }
      public static void main(String[]args) {
            InstanceBlock ib = new InstanceBlock();
            ib.display();
      }
}
```

Output:
Discover yourself.

We can also use *final method* to initialize instance variables. Here is an example of using a final method for initializing an instance variable:

```
class InstanceInitialize {
      int a = initialize();
      protected final int initialize() {
            return 5;
```

```
	}
	void display() {
		System.out.println("a = " + a);
	}
	public static void main(String[]args) {
		InstanceInitialize ii = new InstanceInitialize ();
		ii.display();
	}
}
```

Output:
```
a = 5
```

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

Nesting of Methods

A method can be called by using only its name by another method of the same class. This is known as nesting of methods. For example,

```
class Nesting {
	private int m, n;
	public Nesting(int x, int y) {
		m = x;
		n = y;
	}
	public int largest() {
		if(m > n)
			return(m);
		else
			return(n);
	}
	public void display() {
		int large = largest(); //calling largest method
		System.out.println("Largest value = " + large);
	}
}
class NestingTest {
	public static void main(String[]args) {
		Nesting nest = new Nesting(50, 40);
		nest.display();
	}
}
```

Output:
```
Largest value = 50
```

A method can call any number of methods. It is also possible for a called method to call another method, which in tern call another method and so on.

All about final

The final modifier in java can be used in the following ways:

- Using final with variables
- Using final with inheritance (will be discussed in the next chapter)

Using final with variables

A variable (primitive or user defined) can be declared as final to prevent its content from being modified. If we try to change the content of the final variable the compile time error occurs. Here are some examples of final variables.

```
final static String CLAS = "BBIS";
final Student S1 = new Student("John", 1);
final int VAR1 = 20;
```

It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis.

Java also supports the blank final variables i.e. final variables without initialize value at declaration. When you initialize final variable later then you cannot change it later.

If all variables of a class are final, then the class is immutable and values in the variables of the objects never change after they are initialized. For example, the class below is immutable.

```
class MyClass {
     final String name; //blank final variable
     MyClass (String s) {
      name = s; //initialization of final variable
     }
     .........
     .........
}
```

Nested and Inner Classes

It is possible to define a class within another class. Such classes are known as **nested classes**. They are same as any other class except that they are defined inside the body of another class. For example,

```
class Outer {
      int outer_x = 100;
      void test() {
            Inner in = new Inner();
            in.display();
      }
      class Inner {
            void display() {
                  System.out.println(outer_x);
            }
      }
}
```

There are two types of nested classes: **static** and **non-static**. An **inner class** is a non-static nested class and it can be defined outside a method and inside a method. Some of the reasons behind using nested classes are listed below:

- **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.

Static Nested Classes

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. However, it can access static fields and methods of the class directly. Instances of the static nested class can be created in any method of the outer class.

If the nested class is static, then they are not associated with instance of the outer class. Hence, we can create the instance of the static nested class in the classes other than outer and inner without creating the instance of outer class. For example,

```
class Outer {
      static int outer_x = 100;
      static class Inner {
            void display() {
                  System.out.println(outer_x);
            }
      }
}
class NestedClass {
      public static void main(String[]args) {
            Outer.Inner in = new Outer.Inner(); //creating
instance of static nested class
            in.display();
      }

}
```

Inner Classes outside Methods

An inner class defined outside a method belongs to the class and has class scope (like other members). Like static nested classes, instances of the inner class can be created in any method of the outer class.

We can create the instance of the nested class in other classes than outer and inner by creating the instance of outer class. For example,

```
class Outer {
      int outer_x = 100;
      class Inner {
            void display() {
                  System.out.println(outer_x);
            }
      }
}
class NestedClass {
      public static void main(String[]args) {
            Outer out = new Outer();           //creating instance
            Outer.Inner in = out.new Inner(); // of inner class
            in.display();
      }
}
```

Alternatively, we can created objects as follows:

```
Outer.Inner in = new Outer().new Inner();
```

Inner Classes inside Methods

An inner class defined inside a method belongs to the method and has local (method) scope (like automatic variables). That method can create instances of that class, but other methods of the outer class cannot see this inner class. Inner classes defined inside a method have the following limitations:

- They cannot be declared with an access modifier
- They cannot be declared static
- Inner classes with local scope can only access variables of the enclosing method that are declared final, referring to a local variable of the method or an argument passed into the enclosing method

Inner classes can be defined inside a method without name (anonymous inner class), have following properties:

- Created with no name
- Defined inside a method
- Have no constructor
- Declared and constructed in the same statement
- Useful for event handling

Garbage Collection

We use **new** keyword to dynamically allocate the memory. Since the memory is created it must be destroyed to free the space if the object is not referenced by any variable. To do this task java takes a smarter approach that destroys the memory space used by the object whenever object is no longer in use. The process of freeing up memory used by objects that are not used is called garbage collection and this process is automatically done by java. It is not true that whenever the object is not used java runs the garbage collector but the garbage collector is run occasionally. Normally, we do not have to take care about this thing in our programming. If you want to call the garbage collector by yourself then you can use the static method **gc** of the **System** class as:

```
System.gc();
```

Remember, the above request runs the garbage collector. It is not true that it forces the JVM to perform garbage collection.

Finalizer

When we need to perform some actions at the time of object destruction then we can use the special method called **finalize** to do those actions. For e.g. if you are using some non java resources like file handle in object's constructor and we want to free the handle before object destruction then we can use this concept. The **finalize** method is called just before the garbage collection, has no parameter and has return type **void**. The main issue with **`finalize`** is that the garbage collector is not guaranteed to execute at a specified time or it may never execute before a program terminates. So the call of finalize is also not guaranteed. For this reason, it is advisable to avoid finalize method. The below code segments tells us the definition of finalizer.

```
protected void finalize() {
      System.out.printf( "object destroyed”);
}
```

The access of the finalize method is protected so that code outside the class cannot call this method.

Example:

```
class GarbageCollection {
      protected void finalize() {
            System.out.println("Object destroyed!");
      }
      public static void main(String[]args) {
            GarbageCollection g1 = new GarbageCollection();
            g1 = null;
            System.gc();
      }
}
```

Output:

```
Object destroyed!
```

Chapter 4
Inheritance, Interfaces, and Packages

Inheritance

Inheritance is the mechanism of deriving a new class from existing class. The existing class is known as the **superclass** or **base class** or **parent class** and the new class is called **subclass** or **derived class** or **child class** or **extended class**. The subclass inherits some of the members (fields, methods, and nested classes) from the superclass and can add its own properties as well. A subclass does not inherit `private` members of its parent class. Constructors are not members, so they are not inherited by subclass, but the constructor of the superclass can be invoked from the subclass by using **super** keyword.

Inheritance uses the concept of code **reusability**. Once a super class is written and debugged, we can reuse the properties in this class in other classes by using the concept of inheritance. Reusing existing code saves time and money and increases program's reliability.

An important result of reusability is the ease of distributing classes. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Note: In Java, every class is implicitly a subclass of `Object class`.

Remember: In Java, a superclass reference variable can refer to a subclass object.

Types of Inheritance

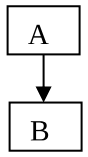
A class can inherit properties from one or more classes and from one or more levels. On the basis of this concept, inheritance may take following different forms:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance

Single Inheritance

In single inheritance, a class is derived from only one existing class. The general form and figure are given below:

```
class A {
      members of A
}

class B  extends A {
      own members of B
}
```



The keyword **extends** signifies that the properties of the superclass are extended to the subclass. The example below shows single inheritance.

```
class Room {
      int length;
      int breadth;
      int area() {
            return length * breadth;
      }
}

class BedRoom extends Room {
      int height;
      void setData(int x, int y, int z) {
            length = x;
            breadth = y;
            height = z;
      }
      int volume() {
            return length * breadth * height;
      }
}

class SingleInheritance {
      public static void main(String[]args) {
            BedRoom br = new BedRoom();
            br.setData(5, 3, 4);
            System.out.println("Area = " + br.area());
            System.out.println("Volume = " + br.volume());
      }
}
```

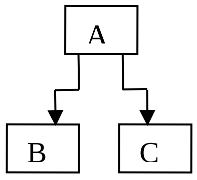
Multiple Inheritance

In this type, a class derives from more than one existing classes. Java does not support multiple inheritance. That is, a class cannot have more than one immediate superclass. However, Java provides an alternate approach known as **interfaces** to support the concept of multiple inheritance. We will discuss this type of inheritance when we discuss interfaces.

Hierarchical Inheritance

In this type, two or more classes inherit the properties of one existing class. The general form and figure are given below:

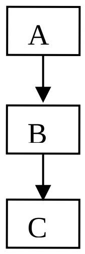
```
class A {
      members of A
}

class B extends A {
      own members of B
}

class C extends A {
      own members of C
}
```



Multilevel Inheritance

The mechanism of deriving a class from another subclass class is known as multilevel inheritance. The process can be extended to an arbitrary number of levels. The general form and figure are given below:

```
class A {
     members of A
}
class B extends A {
     own members of B
}
class C  extends B  {
     own members of C
}
```



Method Overriding

The process of redefining the inherited method of the super class in the derived class is called method **overriding**. For example if we have a method called worksFor() in the super class called *Father*, then the child class named *Son* can redefine the method so that it can represent where the son works. The signature and the return type of the method must be identical in both the super class and the subclass. The overridden method in the subclass should have its access modifier same or less restrictive than that of super class. For example, if the overridden method in super class is protected, then it must be either **protected** or **public** in the subclass but not **private**. The example below shows method overriding.

```
class Father {
     .........
     .........
     void worksFor(){System.out.println("father's office");}
}
class Son extends Father {
     .........
     .........
     void worksFor(){System.out.println("son's office");}
}
```

Remember: Method overloading and method overriding are the examples of polymorphism.

Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. This is the example of *run-time polymorphism*.

When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
class A {
      void callme() {
            System.out.println("Inside A's callme method");
      }
}
class B extends A {
      void callme() {
            System.out.println("Inside B's callme method");
      }
}
class C extends B {
      void callme() {
            System.out.println("Inside C's callme method");
      }
}
class Dispatch {
      public static void main(String[]args) {
            A a = new A();
            B b = new B();
            C c = new C();
            A r;
            r = a;
            r.callme();
            r = b;
            r.callme();
            r = c;
            r.callme();
      }
}
```

Output:

```
Inside A’s callme method
Inside B’s callme method
Inside C’s callme method
```

Using super

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. We can use **super** keyword in the following two ways:

- Using super to call superclass constructor
- Using super to access a member of the superclass that has been hidden by a member of a subclass

Using super to call superclass constructor

A subclass can call a constructor defined by its superclass by the use of following form of

super:

```
super(parameter-list);
```

Here, parameter-list specifies any parameters needed by the constructor in the superclass. **super** always refers to the superclass immediately above the calling class. This is true even in multilevel inheritance. Also, **super** must always be the first statement executed inside a subclass constructor.

Example:

```
class Room {
      int length;
      int breadth;
      Room(int x, int y) { //superclass constructor
            length = x;
            breadth = y;
      }
      int area() {
            return length * breadth;
      }
}

class BedRoom extends Room {
      int height;
      BedRoom(int x, int y, int z) {
            super(x, y); //using super
            height = z;
      }
      int volume() {
            return length * breadth * height;
      }
}

class UseSuper {
      public static void main(String[]args) {
            BedRoom br = new BedRoom(5, 3, 4);
            System.out.println("Area = " + br.area());
            System.out.println("Volume = " + br.volume());
      }
}
```

Using super to access a member of the superclass that has been hidden by a member of a subclass

This form is applicable if member names of a subclass hide members by the same name in the superclass. The usage in this case has the following general form:

```
super.member
```

For example,

```
class A {
      int i;
}
```

```
class B extends A {
      int i;
      B(int a, int b) {
            super.i = a;
            i = b;
      }
      void show() {
            System.out.println("i in superclass: " + super.i);
            System.out.println("i in subclass: " + i);
      }
}
class UseSuper {
      public static void main(String[]args) {
            B subObj = new B(1, 2);
            subObj.show();
      }
}
```

Remember: super always refers to the superclass immediately above the calling class. This is true even in multilevel inheritance.

Execution of Constructors in Multilevel Inheritance

In the multilevel inheritance, constructors are called in order of derivation, from superclass to subclass. For example,

```
class A {
      A() {
            System.out.println("Inside A's constructor");
      }
}
class B extends A {
      B() {
            System.out.println("Inside B's constructor");
      }
}
class C extends B {
      C() {
            System.out.println("Inside C's constructor");
      }
}
class CallingCons {
      public static void main(String[]args) {
            C c = new C();
      }
}
```

Output:

```
Inside A’s constructor
```

Inside B’s constructor
Inside C’s constructor

Abstract Classes and Methods

If we declare a class as abstract, then it is necessary for you to subclass it so as to instantiate the object of that class i.e. we cannot instantiate object of abstract class without creating its subclass. The keyword **abstract** is used to create a class as an abstract class and it can contain abstract methods. Likewise, if you need your method to be always overridden before it can be used, you can declare the method as an abstract method using **abstract** keyword and without the method definition i.e. end it with semicolon. Remember, if you declare some method as an abstract method, you must declare your class as abstract. See the example skeleton below:

```
public abstract class A {
      .........
      .........
      public abstract void mymethod1();  //no definition
      void myMethod2(){……definition here}
}
```

Here we cannot instantiate object of A, if we need to instantiate we must inherit class A to some other class such that it overrides the abstract methods in class A. If the derived class does not define all the abstract methods of the super class then the derived class again should be abstract.

Using final with Inheritance

We can use the **final** keyword in inheritance in the following two ways.

- Using final to prevent overriding
- Using final to prevent inheritance

Using final to prevent overriding

To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. For example,

```
class A {
      final void meth() {
            System.out.println(“This is the final method”);
      }
}

class B extends A {
      void meth() { //Error! Can’t override
            System.out.println(“Illegal”);
      }
}
```

Using final to prevent inheritance

If we want to prevent a class from being inherited, we precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. Here is an example of a final class:

```
final class A {
      .........
      .........
}

class B expends A { //Error
      .........
      .........
}
```

"Is a" versus "Has a"

Just think of three classes A, B and C, where B extends A and C has the instance of class B in it. In this situation we say that “B” is a “A” since B is subclass and the A is parent class. Whereas, in the later case we say “C” has a “B” since there is some object of B in class C.

Chapter 5
Interface

In the Java programming, an *interface* is a reference type, similar to a class that can contain *only* constants, method declarations, and nested types. There are no method bodies. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

To use an interface, we write a class that *implements* the interface. When an instantiable class implements an interface, it should either provide method body for each of the methods declared in the interface or the class should be declared abstract.

Defining an Interface

The syntax for defining an interface is very similar to that for defining a class. To define an interface, we use the keyword **interface** instead of **class**. The general form of an interface definition is:

```
[access] interface InterfaceName [extends IName1[,IName2[,…]] {
      constant declarations;
      method declarations;
}
```

Components:

- **access**: It is either **public** or not used. When it is declared as public, the interface is available to any other code. When it is declared with no access specifier, the interface is only available to other members of the package in which it is declared.
- **InterfaceName**: It is the name of the interface, and can be any valid identifier.
- **constant declarations**: These are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value.
- **method declarations**: Method declarations will contain only a list of methods without body statements. They end with semicolon after the parameter list. These methods are implicitly **abstract**.

Remember: All the members in an interface are public implicitly so we can omit the public keyword there.

Example:
```
interface Area {
      float PI = 3.1415f;
      float compute(float x, float y);
}
```

Extending Interfaces

Like classes, interfaces can also be extended. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. We cab also extend several interfaces into a single interface. When an interface extends two or more interfaces they are separated by commas as follows:

```
interface ItemConstants {
      int code = 1001;
      String name = "Fan";
}
interface ItemMethods {
      void display();
}
interface Item extends ItemConstants, ItemMethods {
      ………
      ………
}
```

Implementing Interfaces

Interfaces are used as superclass whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. We use **implements** keyword to do this. Remember that a class can implement any number of interfaces. The general form is:

```
[access][modifier]class ClassName implements IName1 [,IName2[,…]]
{
      //definition of the class
      //all the methods in the interfaces must also be defined
}
```

Example:

```
interface Area {
      float PI = 3.1415f;
      float findArea(float x);
}

class Circle implements Area {
      public float findArea(float x) {
            return PI * x * x;
      }
}

class InterfaceTest {
      public static void main(String[]args) {
            Circle c1 = new Circle();
            float area = c1.findArea(3.2f);
            System.out.println("Area = " + area);
      }
}
```

Note: If a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface. In the above example we can write,

```
Area c1 = new Circle();
```

Uses of Interface for Achieving Multiple Inheritance

Interfaces can be used for multiple inheritance. In Java a class can extend only one class and if the situation comes where we need to gather the properties of two kinds of objects then Java cannot help us doing so by using classes. In this kind of situation interface can be used to achieve multiple inheritance. Note that we can extend only one class but can implement any number of interfaces. The method signature in the class must match the method signature of the interface. For example,

```
class Test {
      int rollNumber;
      float part1, part2;
      void setData(int r, float p1, float p2) {
            rollNumber = r;
            part1 = p1;
            part2 = p2;
      }
      void showData() {
            System.out.println("Roll Number: " + rollNumber);
            System.out.println("Marks Obtained");
            System.out.println("Part1 = " + part1);
            System.out.println("Part2 = " + part2);
      }
}

interface Sports {
      float sportWt = 6.0f;
      void showSportWt();
}

class Results extends Test implements Sports {
      float total;
      public void showSportWt() {
            System.out.println("Sports Wt = " + sportWt);
      }
      void display() {
            total = part1 + part2 + sportWt;
            showData();
            showSportWt();
            System.out.println("Total score = " + total);
      }
}

class MultipleInheritance {
      public static void main(String[]args) {
            Results s1 = new Results();
            s1.setData(12, 27.5f, 56.0f);
            s1.display();
      }
}
```

Interface versus Abstract Class

- An abstract class can contain non-static and non-final fields where as an interface has static final fields by default.

- An abstract class can contain at least one abstract method whereas an interface has all methods abstract by default. That is, an abstract class can have some implemented methods but an interface does not.
- A class can extend at most one abstract class whereas it can implement any number of interfaces.
- Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way. By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

Casting Rules for Classes and Interfaces

- parentVariable = childVariable : Allowed, No casting needed.
- childVariable = (ChildClass)parentVariable : Allowed, checking is done at run time to see whether the parent is referring to the correct child or not. If the parent is actually pointing to some unrelated subclass, an exception ClassCastException will be raised.
- AVariable = BVariable (A and B are unrelated): not allowed in any circumstances.
- You can cast only within an inheritance hierarchy.
- Use `instanceof` to check before casting from a superclass to a subclass.

Example:

```
class Room {
      int length;
      int breadth;
      Room(int x, int y) {
            length = x;
            breadth = y;
      }
      int area() {
            return length * breadth;
      }
}

class BedRoom extends Room {
      int height;
      BedRoom(int x, int y, int z) {
            super(x, y);
            height = z;
      }
      int volume() {
            return length * breadth * height;
      }
}

class MultilevelInheritance {
      public static void main(String[]args) {
            Room r1 = new BedRoom(5, 3, 4);
            BedRoom br1 = (BedRoom)r1; //casting
            System.out.println("Area = " + br1.area());
            System.out.println("Volume = " + br1.volume());
      }
}
```

Packages

A package is a grouping of related types providing access protection and name space management. Note that types refer to classes, interfaces, enumerations, and annotation types. For example, **java.io** is a package that contains the related types for providing input output facilities. A package name must match directory name where the files are stored. For example, **java.lang** package must have its class files in the directory java/lang.

Creating a Package

To create a package, name the package (we will see the naming convention later) and write a package statement (there is only one package statement per file) with the package name at the top of the source file that is to be bundled in the package. For example, if we want to create a package name **ku.kcm.bbis** then the first line in the file that contains classes, enumerations, etc must have **package ku.kcm.bbis;**

Naming Convention

To get rid of the name collision especially class names, it is necessary for us to put our classes in the package so that when distributing the classes we have no danger of ending up with the problem of having same name (since we can have same name outside the package). However, in these days of global market, we may distribute our package in any part of the world and to avoid the name collision the package name must be unique. So to have unique name we follow the following conventions:

- Package names are written in all lowercase to avoid conflict with the names of classes or interfaces.
- Companies use their reversed Internet domain name to begin their package names—for example, **com.example.orion** for a package named **orion** created by a programmer at example.com.
- Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name (for example, **com.company.region.project**).
- Packages in the Java language itself begin with java. or javax.
- In some cases, the internet domain name may not be a valid package name. This can occur if the domain name contains a hyphen or other special character, if the package name begins with a digit or other character that is illegal to use as the beginning of a Java name, or if the package name contains a reserved Java keyword, such as "int". In this event, the suggested convention is to add an underscore. For example:

Legalizing Package Names	
Domain Name	**Package Name Prefix**

clipart-open.org	org.clipart_open
free.fonts.int	int_.fonts.free
poetry.7days.com	com._7days.poetry

Using Package

There are two ways of accessing classes in another package.
Add the full package name in front of every class name. For example:

```
java.util.Date today = new java.util.Date();
```

This process is very cumbersome so the simpler and common approach is to use import keyword. Using this approach you do not have to give the classes their full names. In this approach either you can import a specific class or the whole package. The import statement will be given at the top of the file just below the package statement. Here is the example for importing the package:

```
import java.util.*;
```

Now you can use

```
Date today = new Date();
```

Also you can do

```
import java.util.Date;
```

When importing a number of packages you need to pay attention to packages for a name conflict. For example, both the *java.util* and *java.sql* packages have a Date class. Suppose you write a program that imports both packages.

```
import java.util.*;
import java.sql.*;
```

If you now use the Date class, then you get a compile-time error:

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

The compiler cannot figure out which Date class you want. You can solve this problem by adding a specific import statement:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

What if you really need both Date classes? Then you need to use the full package name with every class name.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Locating classes in packages is an activity of the compiler. The bytecodes in class files always use full package names to refer to other classes.

Note: The Java compiler automatically imports three entire packages for each source file: (1) the package with no name, (2) the `java.lang` package, and (3) the current package (the package for the current file).

Finding Packages and CLASSPATH

The java run-time system knows about packages in two ways. First, by default, the java run-time system uses the current working directory as its starting point. Thus if our package is in the current directory, or a subdirectory of the current directory, it will be found. Second, we can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

Chapter 6
Essential Java Classes
Exceptions

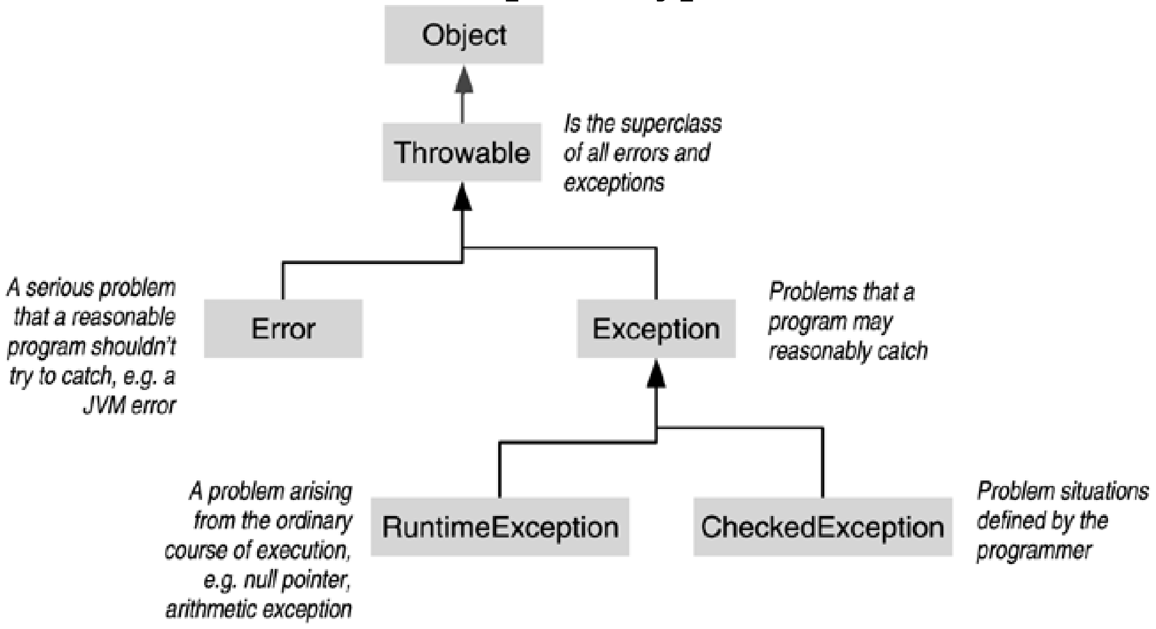
Exception is condition that is reached in the programs due to the abnormal behavior in the program while running. For example, while doing the arithmetic operation if we get the situation when some number is divided by zero then it is an error and this situation is called exception. When an error occurs within a method, the method creates an **exception object** that contains information about the error, including its type and state of the program when the error occurred. The object is then **thrown** in the method that caused the error. This is called **throwing an exception**. The method may choose to handle the exception itself or pass it on. The block of code that is used to handle the exception is called **exception handler**. The appropriate exception handler chosen is called **catch the exception**.

Exception Handling Fundamentals

In java, exception handling is managed by the use of five keywords: **try, catch, throw, throws,** and **finally**. Program structures that you want to monitor for exceptions are contained within a **try** block. If an error occurs within a try block, it is thrown. Your code can catch this exception using **catch** and handle it. System generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified by **throws** clause. Any code that absolutely must be executed before a method returns is put into a **finally** block. The general form of exception handling block is:

```
try {
      //block of code to monitor for errors
} catch(ExceptionType1 exOb) {
      //exception handler for exception type 1
} catch(ExceptionType2 exOb) {
      //exception handler for exception type 2
}
.........
.........
} catch(ExceptionTypeN exOb) {
      //exception handler for exception type N
}
finally {
      //block of codes for finally block
}
```

Exception Types


Object
Throwable
Is the superclass of all errors and exceptions
Error
A serious problem that a reasonable program shouldn't try to catch, e.g. a JVM error
Exception
Problems that a program may reasonably catch
RuntimeException
A problem arising from the ordinary course of execution, e.g. null pointer, arithmetic exception
CheckedException
Problem situations defined by the programmer

The figure above shows the exception class hierarchy. All the exception types are subclasses of the class called **Throwable**. **Throwable** class has two immediate subclasses called **Exception** and **Error**.

The branch with **Exception** class is used to deal with the exceptions that are meant to be caught by the user programs. Also, we will subclass this class to create our own custom exceptions. The Exception class has an important subclass called **RunTimeException** (called unchecked exception). These types of exceptions are automatically defined for the programs for e.g. division by zero, array index out of bound, etc. All the other exceptions are called checked exceptions and must be handled by the program; otherwise compile time error will be generated.

The other branch topped by the **Error** class is not expected to be caught under normal circumstances by the program. This type of exceptions generally represents exceptions related to run-time environment and JVM errors for e.g. stack overflow.

Uncaught Exceptions

The example below shows the effect when we don't handle exceptions.

Program 7.1

```
class UncaughtExcep7_1 {
      public static void main(String []args) {
            int divisor = 0;
            int quot = 33/divisor; //divide by zero
            System.out.println("This won'texecute");
      }
}
```

When you run this program (where exception is not caught) java sees the exception (see divide by zero) and processes the exception by default handler. The default handler displays the string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. The handler displays the following:

```
Exception in thread "main" java.lang.ArithmeticException : / by zero
        at UncaughtExcep7_1.main(UncaughtExcep7_1.java:4)
```

The stack trace will always show the sequence of method invocations that led up to the error. For example, if we modify the above program as below, it introduces the same error as follows:

```
class UncaughtExcep7_1 {
        public static void subroutine() {
                int divisor = 0;
                int quot = 33/divisor; //divide by zero
        }
        public static void main(String []args) {
                subroutine();
        }
}
```

```
Exception in thread "main" java.lang.ArithmeticException : / by zero
        at UncaughtExcep7_1.subroutine(UncaughtExcep7_1.java:4)
        at UncaughtExcep7_1.main(UncaughtExcep7_1.java:7)
```

Using try and catch

When you want to handle the exceptions by yourself so as to fix the error and prevent the automatic termination of the program you can enclose the code that you want to monitor inside the **try** block and use the **catch**. You can put as many **catch** clauses as you want to catch separate exception that can occur in your program. However, in case of multiple catch statements it is necessary that the exception subclasses must come before the superclasses, otherwise compile time error will occur. If you want to execute some codes in your program (when using try-catch block) even if the exception is thrown or no exception is caught, then you can use **finally** clause (normally you use **finally** if you want to free the allocated memory and other resources). Remember, **try** block requires at least one catch clause or **finally** clause where finally clause is optional. See below for example:

Program 7.2

```
class CatchMultiExcep7_2 {
        public static void main(String []args) {
                try {
                        int argnum = args.length;
                        System.out.println("Number of arguments = "+
argnum);
                        int quot = 33/argnum;
                        int c[] = {1, 2, 3};
                        c[22] = 34;
                } catch(ArithmeticException e) {
                        System.err.println("Divide by zero: "+ e);
```

```
          } /*catch(Exception e){} --- do not put this here this
will cause error because Exception class is superclass of the
Exception being caught below*/
          catch(ArrayIndexOutOfBoundsException e) {
               System.out.println("Exception: " + e);
          } finally {
               System.out.println("Inside finally");
          }
     }
}
```

Output

- On **java CatchMultiExcep7_2**

```
Number of arguments = 0
Divide by zero: java.lang.ArithmeticException: / by zero
Inside finally
```

- On **java CatchMultiExcep7_2 23 54**

```
Number of arguments = 2
Exception: java.lang.ArrayIndexOutOfBoundsException
Inside finally
```

Nested try

We can put the **try** statement inside another **try** statement. If the inner try statement does not have the catch handler for the particular exception then the program try to use the catch statement of the outer try statement. This process continues for any level of nesting. If no catch statement is matched then the java run-time system handles the exception.

Program 7.3

```
class NestedTry7_3 {
     public static void main(String []args) {
          try {
               int argnum = args.length;
               System.out.println("Number of arguments = "+ argnum);
               int quot = 33/argnum;
               try  {
                    if (argnum == 1)
                         argnum = 34/(argnum-argnum);
                    if (argnum == 2) {
                         int c[] = {1, 2, 3};
                         c[22] = 34;
                    }
               } catch(ArithmeticException e) {
                    System.err.println("Divide by zero: "+ e);
               }
          } catch(ArrayIndexOutOfBoundsException e) {
               System.out.println("Exception: " + e);
          }
     }
}
```

Output:

- On **java NestedTry7_3**

```
Number of arguments = 0
```

```
Exception in thread "main" java.lang.ArithmeticException : / by zero
        at NestedTry7_3.main(NestedTry7_3.java:6)
```

- On **java NestedTry7_3 a**

```
Number of arguments = 1
Divide by zero: java.lang.ArithmeticException: / by zero
```

- On **java NestedTry7_3 a b**

```
Number of arguments = 2
Exception: java.lang.ArrayIndexOutOfBoundsException
```

- On **java NestedTry7_3 a b c**

```
Number of arguments =3
```

Using Throw

If you want to throw an exception explicitly by your method in a program, **throw** statement can be used. The **throw** statement requires the **Throwable** object or object of any subclass of **Throwable** class. Its format is as given below:

```
	throw someThrowableInstance;
```

There are two ways to obtain the **Throwable** object, using parameter into a catch clause, or creating one with the **new** operator. The flow of execution stops immediately after the throw statement. The **try** block is inspected to see if it has a **catch** statement matching the exception. If the matching **catch** statement is found it is used otherwise default handler is used.

Program 7.4

```
class ThrowTest7_4 {
	static void throwDemo() {
		try {
			throw new NullPointerException("This is demo");
		} catch(NullPointerException e) {
			System.out.println("Caught inside throwDemo method");
			throw e;   //rethrow the exception
		}
	}
	public static void main(String []args) {
		try {
			throwDemo();
		} catch(NullPointerException e) {
			System.out.println("Rethrown: Caught inside Main 
method " + e);
		}
	}
}
```

Output:

```
Caught inside throwDemo method
Rethrown: Caught inside Main method java.lang.NullPointerException: This is demo
```

Using Throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that caller of the method can guard itself against that exception. This is necessary for all exceptions except those of types **Error** and **RunTimeException**, or their subclasses. The general form of using **throws** is as below:

```
[access] [modifier] type methodName(paramlist) throws commaseparatedexceptionslist {
        //method body
}
```

Program 7.5

```
class ThrowsTest7_5 {
      static void throwsDemo(int x) throws ClassNotFoundException,
InterruptedException {
            if (x < 100) {
                  System.out.println("I am Inside throwsDemo with
integer < 100" );
                        throw new ClassNotFoundException("Class not
found");
            } else {
                  System.out.println("I am Inside throwsDemo with
integer >= 100" );
                  throw new InterruptedException("Interrupted");
            }
      }
      public static void main(String []args) throws
InterruptedException {
            try {
                  throwsDemo(5);
            } catch(ClassNotFoundException e) {
                  System.out.println("Caught " + e);
            }
            try {
                  throwsDemo(500);
            } catch(ClassNotFoundException e) {
                  System.out.println("Caught " + e);
            }
      }
}
```

Output:

```
I am Inside throwsDemo with integer < 100
Caught java.lang.ClassNotFoundException: Class not found
I am Inside throwsDemo with integer >= 100
Exception in thread “main” java.lang.InterruptedException : Interrupted
        At ThrowsTest7_5.throwsDemo(ThrowsTest7_5.java:11)
        At ThrowsTest7_5.main(ThrowsTest7_5.java:19)
```

Some Exceptions

- Unchecked Exceptions
  - → **ArithmeticException:** Arithmetic error like divide by zero.

- → **ArrayIndexOutOfBoundsException:** Array index is out of bounds.
- → **ArrayStoreException:** Assignment to an array element of an incompatible type.
- → **ClassCastException:** Invalid class cast.
- → **IllegalArgumentException:** Illegal argument used to invoke a method.
- → **IndexOutOfBoundsException:** Some types of index are out of bounds.
- → **NegativeArraySizeException:** Array created with negative size.
- → **NullPointerException:** Invalid use of null reference.
- → **NumberFormatException:** Invalid conversion of a string to a numeric format.
- → **StringIndexOutOfBoundsException:** Attempt to index outside the bounds of a string.

- Checked Exceptions
  - → **ClassNotFoundException**: Class not found.
  - → **CloneNotSupportedException:** Attempt to clone (duplicate) an object that does not implement the **Cloneable** interface.
  - → **IllegalAccessException:** Access to a class is denied.
  - → **InstantiationException:** Attempt to create an object of an abstract class or interface.
  - → **InterruptedException:** One thread has been interrupted by another thread.
  - → **NoSuchFieldException:** A requested field does not exist.
  - → **NoSuchMethodException:** A requested method does not exist.

Chapter 7
Multithreading

A multithreaded program contains two or more parts in the program that can run concurrently. Each part of such program is called a thread, and each **thread** defines a separate path of execution. A multithreaded program can perform two or more tasks simultaneously. **Processes** are **heavyweight** tasks that require their own separate address space. Threads, on the other hand, are **lightweight**. They share the same address space and cooperatively share the same heavyweight process. Inter thread communication is inexpensive and **context switching** from one thread to the next is of low cost. Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

The Java Thread Model

In multithreading one thread can pause without stopping the other parts of the program and all other threads continue to run. Thread exists in several states: a thread can be **running**, it can be **ready to run** as soon as it gets the CPU time, a running thread can be **suspended**, a suspended thread can be **resumed**, a thread can be **blocked** when waiting for a resource, and can be **terminated**. The thread model also includes the following concepts.

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Priority is used to decide when to switch from one running thread to the next. This is called **Context Switch**.

Synchronization

Multithreading causes an asynchronous behavior to the program and if synchronization is needed (for e.g. the condition where two threads are using the same resource; then the thread should not be able to write if another thread is reading it) Java uses the **monitor**. A monitor can be thought of the placeholder that can handle only on thread. So if one thread is inside the monitor, then the other threads must wait until the thread is inside the monitor so that the shared resource is protected from being manipulated by more than one thread at a particular time.

Messaging

When the program is divided into separate threads, then there must be way of communication among the threads. In java this process is done through the call of various predefined methods (we will see these methods later).

Thread Class and Runnable Interface

To create a new thread, your program will either extend **Thread** class or implement the **Runnable** interface. The **Thread** class defines different methods for managing the threads. Some of them are:

getName:	Obtain a thread's name.
getPriority:	Obtain a thread's priority.
isAlive:	Determine if a thread is still running.

join:	Wait for a thread to terminate.
run:	Entry point for the thread.
sleep:	Suspend a thread for a period of time.
start:	Start a thread by calling its run method.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the **main thread**. Although the main thread is created automatically when your program is started, it can be controlled through a thread object. To do this, you must obtain a reference to it by calling a static method **currentThread()** in the Thread class.

Program

```
class MainThreadDemo {
     public static void main(String args[]) {
          Thread t = Thread.currentThread(); //static Thread
currentThread()
          System.out.println("Current thread: " + t);
          t.setName("My Thread"); // change the name of the
thread
          System.out.println("After name change: " + t);
          try {
                    for(int n = 5; n > 0; n--) {
                    System.out.println(n);
                    Thread.sleep(1000); //argument is given in
milliseconds
                    }
          } catch (InterruptedException e) {
               System.out.println("Main thread interrupted");
          }
     }
}
```

Output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Thread[main,5,main] *: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is* ***main****. Its priority is 5, which is the default value, and* ***main*** *is also the name of the group of threads to which this thread belongs.*

Creating a Thread

As already discussed we can create thread in two ways; one by extending **Thread** class and the other is implementing **Runnable** interface.

Implementing Runnable

To implement **Runnable**, a class need only implement a single method called **run()**. Inside run, you will define the code that constitutes the new thread.

Program

```
class NewThread implements Runnable {
      static int numThreads = 0;
      Thread t;
      NewThread(String name) {
            numThreads++;
            t = new Thread(this, name);
            System.out.println("Child thread("+numThreads+"): " +
t);
            t.start(); // Start the thread
      }
      public void run() {
            try {
                        for(int i = 5; i > 0; i--) {
                        System.out.println("Child Thread: " + i);
                        Thread.sleep(500);
                        }
            } catch (InterruptedException e) {
                  System.out.println("Child interrupted.");
            }
            System.out.println("Exiting child thread.");
      }
}
class ThreadDemo {
      public static void main(String args[]) {
            new NewThread("Second Thread"); // create a new thread
            new NewThread("Third Thread");
            new NewThread("Fourth Thread");
            try {
                        for(int i = 5; i > 0; i--) {
                        System.out.println("Main Thread: " + i);
                        Thread.sleep(1000);
                        }
            } catch(InterruptedException e) {
                  System.out.println("Main thread interrupted.");
            }
            System.out.println("Main thread exiting.");
      }
}
```

Output

```
Child thread(1): Thread[Second Thread,5,main]
Child thread(2): Thread[Third Thread,5,main]
Child thread(3): Thread[Fourth Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 5
Child Thread: 5
Child Thread: 4
Child Thread: 4
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 3
Child Thread: 3
Child Thread: 2
Child Thread: 2
Child Thread: 2
Main Thread: 3
Child Thread: 1
Child Thread: 1
Child Thread: 1
Exiting child thread.
Exiting child thread.
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread

The extending class must override the **run()** method, which is the entry point for the new thread. The following change on the program above is to be done.

class NewThread implements Runnable TO

class NewThread extends Thread

```
  NewThread(String name) {
    numThreads++;
    t = new Thread(this, name);
    System.out.println("Child thread("+numThreads+"): " + t);
    t.start();
}
```

TO

```
  NewThread(String name) {
    super(name);  //uses Thread(String str) constructor
    numThreads++;
    System.out.println("Child thread("+numThreads+"): " + this);
    start();
  }
```

Using isAlive() and join()

The method **isAlive()** is defined by Thread class, and its general form is:

final boolean isAlive()

It returns true if the thread upon which it is called is still running, for e.g.

```
NewThread ob1 = new NewThread("First");
System.out.println("Thread one is alive: "+ob1.t.isAlive());
```

Here ob1 is using the **Thread** object t (an instance variable for **NewThread**; see code above) in **NewThread** class that implements **Runnable** interface.

The **join()** method waits until the thread on which it is called terminates. The general form is:
final void join() throws InterruptedException
e.g. *ob1.t.join();*

Thread Priorities

To set a thread priority, use the **setPriority()** method. It is the member of **Thread** with general form:

final void setPriority(int level) //level can be 1 to 10 normal is 5

For e.g.

ob1.t.setPriority(4);

Similarly to obtain the priority of the thread we use the **getPriority()** method with the form:

final int getPriority()

For e.g.

int p = ob1.t.getPriority();

Synchronization

For synchronization, we simply call the method inside the class which contains a synchronized block. The general form is:

synchronized (object) {//statement to be synchronized}

Here the object is a reference to the object being synchronized. For e.g.

public void run(){synchronized(this){name = fname;}}

Another way of synchronization is using synchronized method. Here the method is defined by using **synchronized** keyword. For e.g.

public synchronized void syncMethod(){//codes here…..}

When a thread is invoking synchronized method for an object, all other threads invoking the same method for objects must wait until the running thread completes the action.
Remember: you cannot use **synchronized** keyword for a constructor (this is an error).

Interthread Communication

For interthreaded communication, we use **wait()**, **notify()** and **notifyAll()** methods.
wait() tells the calling thread to give up the monitor and go to sleep until some other threads enters the same monitor and calls notify().
notify() wakes up a thread that called wait() on the same object.
notifyAll() wakes up the threads that called wait() on the same object. One of the threads will be granted access.
These methods are declared as shown below:

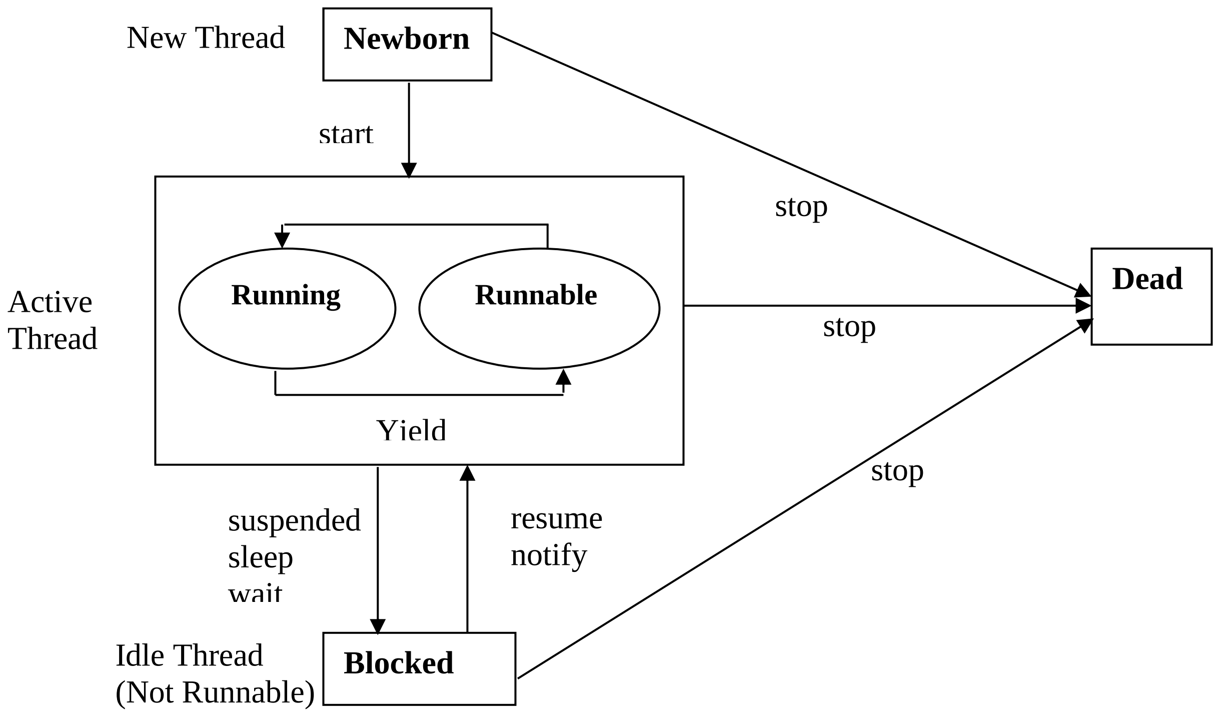
final void wait() throws InterruptedException
final void notify()
final void notifyAll()

Life Cycle of a Thread

During the lifetime of a thread, there are many states it can enter. These include:

- Newborn State
- Runnable State
- Running State

- Blocked State
- Dead State



Newborn State
When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it.

- Schedule it for running using the **start( )** method.
- Kill it using **stop( )** method.

If scheduled, it moves to the runnable state. If we attempt to use any other method at this stage, an exception will be thrown.

Runnable State
The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round-robin fashion, i.e., first-come, first-serve manner. This is called *time slicing*.

Running State
Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or a higher priority thread preempts it. A running thread may relinquish its control in one of the following situations:

- It has been suspended using **suspend( )** method.
- It has been made to sleep using **sleep( )** method.
- It has been told to wait until some event occurs. This is done using the **wait( )** method.

Blocked State

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.

Dead State

A running thread ends its life when it has completed executing its **run( )** method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon as it is born, or while it is running, or even when it is in blocked state.

Chapter 8
Input Output (java.io package)

The package **java.io** is used for using the classes suitable for taking input (from program, disks, objects, etc.) and producing output (to program, disks, objects, printers, etc.). In java, I/O operations are performed through **streams**. **Input streams** represent sources that you can read data from and **output streams** represent sources that you can write data to. These streams are subdivided into different types like **byte-oriented streams** and **character-oriented streams**. In this lesson we cover briefly about these I/O.

Byte Streams and Character Streams

Byte streams in Java are used for handling I/O of bytes. Byte streams are generally suitable for reading and writing binary data like audio, video, images etc. However the most basic way of reading and writing is using byte streams. Java has also defined the **character streams** that are used for input and outputs of character types.

Byte Stream Classes

There are two top-level abstract classes called **InputStream** and **OutputStream** for byte streams I/O. Each of these classes has many concrete subclasses that are used to handle different I/O like disks, memory buffers, files, etc. To use stream classes you must import **java.io** package in your program. The below is the list of some byte stream classes.

Stream Class	**Meaning**
BufferedInputStream	Input stream that reads from buffer
BufferedOutputStream	Output stream that writes to a buffer
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Filter input stream
FilterOutputStream	Filter output stream
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Character Stream Classes

The two top-level abstract classes **Reader** and **Writer** define the character streams. These classes handle Unicode character streams. Java uses concrete subclasses of the above two classes for character I/O handling. The below is the list of some character stream classes.

Stream Class	**Meaning**
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Using FileInputStream and FileOutputStream Classes

These two classes are used for reading and writing files. These classes (descendent of **InputStream** and **OutputStream** classes) create byte streams linked to files. We use the following commonly used constructors for file reading and writing purposes.

FileInputStream(String fileName) throws FileNotFoundException

creates a **`FileInputStream`** by opening a connection to an actual file, the **fileName** is the path in the file system. If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading then a **`FileNotFoundException`** is thrown.

FileOutputStream(String filename, boolean append) throws FileNotFoundException

creates an output file stream to write to the file with the specified name. If the second argument is true, then bytes will be written to the end of the file rather than the beginning. If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then a **FileNotFoundException** is thrown.

After we use the file we must close the file by calling close method defined in both the classes above. Its definition format is:
void close() throws IOException

Reading from the file can be done by using read method defined in **FileInputStream** class whose definition syntax is:

int read() throws IOException

The above method reads each byte from the file and returns the byte as an integer value. -1 is returned if end of file is obtained.

Writing to the file is done by using the write method defined in **FileOutputStream** class whose syntax is:

void write(int byteval) throws IOException

Program

```
import java.io.*;
public class FileReadWrite {
     public static void main(String[] args) throws IOException {
          FileInputStream in = null;
          FileOutputStream out = null;
          try {
                    in = new FileInputStream("BBIS.txt");
                    out = new FileOutputStream("BBIS1.txt");
                    int c;
                    while ((c = in.read()) != -1) {
                         out.write(c);
                    }
          } finally {
               if (in != null) {
                    in.close();
               }
               if (out != null) {
                    out.close();
               }
          }
     }
}
```

Using FileReader and FileWriter Classes

These two classes are used for reading and writing files. These classes (descendents of **Reader** and **Writer** classes) create character streams linked to files. **FileReader** is the direct descendent of **InputStreamReader** class that acts as the bridge between byte stream and character stream. Similarly, **FileWriter** is the direct descendent of **OutputStreamReader** class that acts as the bridge between byte stream and character stream. We use the following commonly used constructors for file reading and writing purposes.

FileReader(String fileName) throws FileNotFoundException creates a `FileReader` by opening a connection to an actual file, the ***fileNname*** is the path in the file system. If the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading then a `FileNotFoundException` is thrown.

FileWriter(String filename, boolean append) throws IOException

creates an output file stream to write to the file with the specified name. If the second argument is true, then character will be written to the end of the file rather than the beginning. If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then an **IOException** is thrown.

After we use the file we must close the file by calling close method defined in both the classes above. Its definition format is:

void close() throws IOException

Reading from the file can be done by using read method defined in **FileReader** class whose definition syntax is:

int read() throws IOException

The above method reads each character from the file and returns the character as an integer value. -1 is returned if end of file is obtained.

Writing to the file is done by using the write method defined in **FileWriter** class whose syntax is:

void write(int c) throws IOException

c is an integer representing the character to be written.

Program

Change the class name **FileInputStream** to **FileReader** and **FileOutputStream** to **FileWriter** in the above program and run.

Using Classes for Line Reading

A line is terminated by newline character (“\n”) or some system uses carriage return character followed by newline character (“\r\n”) or carriage return character (“\r”) only. To perform the line based input we can use **BufferedReader**.

BufferedReader

This class is inherited from **Reader** class. It reads the text from the character input stream using the character buffer that helps efficient reading of characters, arrays, and lines. The buffering causes the input streams to be stored in temporary memory location so that the costly operation “read” that would be otherwise done from the file by converting bytes to the character is efficiently done through the buffer. The normally used constructor for this class is:

BufferedReader(Reader in)

The example of using the constructor is as given below:
BufferedReader in = new BufferedReader(new FileReader("my.txt"));

Some methods related to this class are given below:

- *int read()throws IOException*: Reads a single character. This method overrides read method in the class **Reader**.
- *String readLine() throws IOException*: Reads a line of text and returns the string with line content without line termination character. Null is returned if end of stream is met.
- *void close() throws IOException*: Close the stream.
- *long skip(long n) throws IOException*: Skip **n** number of characters and returns the number of characters actually skipped.

Examples of reading console input using BufferedReader

Reading Characters

```
import java.io.*;
class BRRead {
	public static void main(String[]args) throws IOException {
		char c;
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		System.out.print("Enter a character:");
		c = (char)br.read();
		System.out.print("Input character is " + c);
	}
}
```

Reading Strings

```
import java.io.*;
class BRReadLines {
	public static void main(String[]args) throws IOException {
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		System.out.print("a = ");
		int a = Integer.parseInt(br.readLine());
		System.out.print("b = ");
		int b = Integer.parseInt(br.readLine());
		System.out.println("Sum = " + (a + b));
	}
}
```

Chapter 8
Concept of Applets

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.

- **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;

import java.awt.*;



public class HelloWorldApplet extends Applet {

   public void paint (Graphics g) {

      g.drawString ("Hello World", 25, 50);

   }

}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet

- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Following is an example that invokes the "Hello, World" applet −

```
<html>

  <title>The Hello, World Applet</title>

  <hr>

  <applet code = "HelloWorldApplet.class" width = "320" height = "120">

    If your browser was Java-enabled, a "Hello, World"

    message would appear here.
  </applet>
  <hr>
</html>
```

Note − You can refer to HTML Applet Tag to understand more about calling applet from HTML.

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with an </applet> tag.

If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.