

Introduction to Java

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere**.

Java is –

- **Object Oriented** – In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** – Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** – Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** – With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** – Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** – Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** – Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** – With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** – Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** – With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** – Java is designed for the distributed environment of the internet.
- **Dynamic** – Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java

James Gosling initiated Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called 'Oak' after an oak tree that stood outside Gosling's office, also went by the name 'Green' and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May, 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Tools You Will Need

For performing the examples discussed in this course, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

You will also need the following softwares –

- Linux 7.1 or Windows XP/7/8/10 operating system
- Java JDK 8
- IDE such as Netbeans or Eclipse

This course will provide the necessary skills to create GUI, networking, and web applications using Java.

What is Next?

The next chapter will guide you to how you can obtain Java and its documentation. Finally, it instructs you on how to install Java and prepare an environment to develop Java applications.

Environment Setup

In this chapter, we will discuss on the different aspects of setting up a congenial environment for Java.

Local Environment Setup

If you are still willing to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Following are the steps to set up the environment.

Java SE is freely available from the link [Download Java](https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html) ([https://https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html](https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html)). You can download a version based on your operating system.

Follow the instructions to download Java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you will need to set environment variables to point to correct installation directories –

Setting Up the Path for Windows

Assuming you have installed Java in *c:\Program Files\java\jdk* directory –

- Right-click on 'My Computer' and select 'Properties'.
- Click the 'Environment variables' button under the 'Advanced' tab.
- Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable `PATH` should be set to point to where the Java binaries have been installed. Refer to your shell documentation, if you have trouble doing this.

Example, if you use *bash* as your shell, then you would add the following line to the end of your `.bashrc`: `export PATH = /path/to/java:$PATH`

Popular Java Editors

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following –

- **Notepad** – On Windows machine, you can use any simple text editor like Notepad (Recommended for this course), TextPad.
- **Netbeans** – A Java IDE that is open-source and free which can be downloaded from <https://www.netbeans.org/index.html>.
- **Eclipse** – A Java IDE developed by the eclipse open-source community and can be downloaded from <https://www.eclipse.org/>.

What is Next?

Next chapter will teach you how to write and run your first Java program and some of the important basic syntaxes in Java needed for developing applications.

Basic Syntax

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- **Methods** – A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** – Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

Example

```
public class MyFirstJavaProgram {  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output  
     */  
  
    public static void main(String []args) {  
        System.out.println("Hello World"); // prints Hello World  
    }  
}
```

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps –

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption: The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

Output

```
C:\> javac MyFirstJavaProgram.java  
C:\> java MyFirstJavaProgram  
Hello World
```

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** – Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names** – For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
Example: `class MyFirstJavaClass`
- **Method Names** – All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
Example: `public void myMethodName()`
- **Program File Name** – Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).
Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as 'MyFirstJavaProgram.java'
- **public static void main(String args[])** – Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

In Java, there are several points to remember about identifiers. They are as follows –

- All identifiers should begin with a letter (A to Z or a to z), currency character (\$) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: age, \$salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers –

- **Access Modifiers** – default, public, protected, private
- **Non-access Modifiers** – final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

Java Variables

Following are the types of variables in Java –

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static Variables)

Java Arrays

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap. We will look into how to declare, construct, and initialize in the upcoming chapters.

Java Enums

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.

Example

```
class FreshJuice {
    enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
    FreshJuiceSize size;
}

public class FreshJuiceTest {

    public static void main(String args[]) {
        FreshJuice juice = new FreshJuice();
        juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
        System.out.println("Size:" + juice.size);
    }
}
```

The above example will produce the following result:

Output

Size:MEDIUM

Note: Enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Java Keywords

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

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

Comments in Java

Java supports single-line and multi-line comments very similar to C and C++. All characters available inside any comment are ignored by Java compiler.

Example

```
public class MyFirstJavaProgram {  
  
    /* This is my first java program.  
     * This will print 'Hello World' as the output  
     * This is an example of multi-line comments.  
     */  
  
    public static void main(String []args) {  
        // This is an example of single line comment  
        /* This is also an example of single line comment. */  
        System.out.println("Hello World");  
    }  
}
```

Output

Hello World

Using Blank Lines

A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

Inheritance

In Java, classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

This concept allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the **superclass** and the derived class is called the **subclass**.

Interfaces

In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

An interface defines the methods, a deriving class (subclass) should use. But the implementation of the methods is totally up to the subclass.

What is Next?

The next section explains about Objects and classes in Java programming. At the end of the session, you will be able to get a clear picture as to what are objects and what are classes in Java.

Object and Classes

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts –

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Passing

In this chapter, we will look into the concepts - Classes and Objects.

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior. If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

Example

```
public class Dog {  
    String breed;  
    int age;  
    String color;  
  
    void barking() {  
    }  
  
    void hungry() {  
    }  
  
    void sleeping() {  
    }  
}
```

A class can contain any of the following variable types.

- **Local variables** – Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** – Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** – Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor –
Example

```
public class Puppy {  
    public Puppy() {  
    }  
  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
    }  
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Note: We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class –

- **Declaration** – A variable declaration with a variable name with an object type.
- **Instantiation** – The 'new' keyword is used to create the object.
- **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object –
Example

```
public class Puppy {  
    public Puppy(String name) {  
        // This constructor has one parameter, name.  
        System.out.println("Passed Name is:" + name );  
    }  
  
    public static void main(String []args) {  
        // Following statement would create an object myPuppy  
        Puppy myPuppy = new Puppy( "tommy" );  
    }  
}
```

If we compile and run the above program, then it will produce the following result:

Output

Passed Name is:tommy

Accessing Instance Variables and Methods

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path –

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example

This example explains how to access instance variables and methods of a class.

```
public class Puppy {
    int puppyAge;

    public Puppy(String name) {
        // This constructor has one parameter, name.
        System.out.println("Name chosen is:" + name );
    }

    public void setAge( int age ) {
        puppyAge = age;
    }

    public int getAge( ) {
        System.out.println("Puppy's age is:" + puppyAge );
        return puppyAge;
    }

    public static void main(String []args) {
        /* Object creation */
        Puppy myPuppy = new Puppy( "tommy" );

        /* Call class method to set puppy's age */
        myPuppy.setAge( 2 );

        /* Call another class method to get puppy's age */
        myPuppy.getAge( );

        /* You can access instance variable as follows as well */
        System.out.println("Variable Value:" + myPuppy.puppyAge );
    }
}
```

If we compile and run the above program, then it will produce the following result:

Output

```
Name chosen is:tommy
Puppy's age is:2
Variable Value:2
```

Source File Declaration Rules

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non-public classes.
- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example: the class name is *public class Employee* then the source file should be as *Employee.java*.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.

Apart from the abovementioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import Statements

In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory `java_installation/java/io` –

```
import java.io.*;
```

A Simple Case Study

For our case study, we will be creating two classes. They are *Employee* and *EmployeeTest*. First open notepad and add the following code. Remember this is the *Employee* class and the class is a public class. Now, save this source file with the name *Employee.java*.

The *Employee* class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

Example

```
import java.io.*;
public class Employee {

    String name;
    int age;
    String designation;
    double salary;

    // This is the constructor of the class Employee
    public Employee(String name) {
        this.name = name;
    }

    // Assign the age of the Employee to the variable age.
    public void empAge(int empAge) {
        age = empAge;
    }

    /* Assign the designation to the variable designation.*/
    public void empDesignation(String empDesig) {
        designation = empDesig;
    }

    /* Assign the salary to the variable salary.*/
    public void empSalary(double empSalary) {
        salary = empSalary;
    }

    /* Print the Employee details */
    public void printEmployee() {
        System.out.println("Name:" + name );
        System.out.println("Age:" + age );
        System.out.println("Designation:" + designation );
        System.out.println("Salary:" + salary);
    }
}
```

As mentioned previously in this course, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.

Following is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file.

```
import java.io.*;
public class EmployeeTest {

    public static void main(String args[]) {
        /* Create two objects using constructor */
        Employee empOne = new Employee("James Smith");
        Employee empTwo = new Employee("Mary Anne");

        // Invoking methods for each object created
        empOne.empAge(26);
        empOne.empDesignation("Senior Software Engineer");
        empOne.empSalary(1000);
        empOne.printEmployee();

        empTwo.empAge(21);
        empTwo.empDesignation("Software Engineer");
        empTwo.empSalary(500);
        empTwo.printEmployee();
    }
}
```

Now, compile both the classes and then run *EmployeeTest* to see the result as follows –

Output

```
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

What is Next?

In the next session, we will discuss the basic data types in Java and how they can be used when developing Java applications.

Constructors

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Syntax

Following is the syntax of a constructor –

```
class ClassName {  
    ClassName() {  
    }  
}
```

Java allows two types of constructors namely –

- No argument Constructors
- Parameterized Constructors

No argument Constructors

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

Example

```
public class MyClass {  
    int num;  
    MyClass() {  
        num = 100;  
    }  
}
```

You would call constructor to initialize objects as follows

```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();  
        System.out.println(t1.num + " " + t2.num);  
    }  
}
```

This would produce the following result

100 100

Parameterized Constructors

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor –

```
// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass(int i ) {
        x = i;
    }
}
```

You would call constructor to initialize objects as follows –

```
public class ConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}
```

This would produce the following result:

```
10 20
```

Basic Datatypes

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java –

- Primitive Data Types
- Reference/Object Data Types

Primitive Data Types

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive) ($2^7 - 1$)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^{15})
- Maximum value is 32,767 (inclusive) ($2^{15} - 1$)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is -2,147,483,648 (-2^{31})
- Maximum value is 2,147,483,647 (inclusive) ($2^{31} - 1$)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808 (-2^{63})
- Maximum value is 9,223,372,036,854,775,807 (inclusive) ($2^{63} - 1$)

- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

char

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

Reference Datatypes

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example –

```
byte a = 68;
char a = 'A';
```

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example –

```
int decimal = 100;  
int octal = 0144;  
int hexa = 0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are –

```
"Hello World"  
"two\nlines"  
"\\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example –

```
char a = '\u0001';  
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well –

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

What is Next?

This chapter explained the various data types. The next topic explains different variable types and their usage. This will give you a good understanding on how they can be used in the Java classes, interfaces, etc.

Variable Types

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration:

```
data type variable [ = value][, variable [ = value] ...] ;
```

Here *data type* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java –

```
int a, b, c;           // Declares three ints, a, b, and c.
int a = 10, b = 10;    // Example of initialization
byte B = 22;           // initializes a byte type variable B.
double pi = 3.14159;   // declares and assigns a value of PI.
char a = 'A';          // the char variable a is initialized with value 'a'
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java –

- Local variables
- Instance variables
- Class/Static variables

Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
public class Test {
    public void pupAge() {
        int age = 0;
        age = age + 7;
        System.out.println("Puppy age is:" + age);
    }
    public static void main(String args[]) {
        Test test = new Test();
        test.pupAge();
    }
}
```

This will produce the following result:

Output

Puppy age is:7

Example

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test {  
    public void pupAge() {  
        int age;  
        age = age + 7;  
        System.out.println("Puppy age is:" + age);  
    }  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

This will produce the following error while compiling it –

Output

```
Test.java:4:variable number might not have been initialized  
age = age + 7;  
      ^  
1 error
```

Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

Example

```
import java.io.*;
public class Employee {

    // this instance variable is visible for any child class.
    public String name;

    // salary variable is visible in Employee class only.
    private double salary;

    // The name variable is assigned in the constructor.
    public Employee (String empName) {
        name = empName;
    }

    // The salary variable is assigned a value.
    public void setSalary(double empSal) {
        salary = empSal;
    }

    // This method prints the employee details.
    public void printEmp() {
        System.out.println("name:" + name );
        System.out.println("salary:" + salary);
    }

    public static void main(String args[]) {
        Employee empOne = new Employee("Ransika");
        empOne.setSalary(1000);
        empOne.printEmp();
    }
}
```

This will produce the following result:

Output

```
name:Ransika
salary:1000.0
```

Class/Static Variables

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.

- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

Example

```
import java.io.*;
public class Employee {

    // salary variable is a private static variable
    private static double salary;

    // DEPARTMENT is a constant
    public static final String DEPARTMENT = "Development ";

    public static void main(String args[]) {
        salary = 1000;
        System.out.println(DEPARTMENT + "average salary:" + salary);
    }
}
```

This will produce the following result:

Output

Development average salary:1000

Note: If the variables are accessed from an outside class, the constant should be accessed as `Employee.DEPARTMENT`

What is Next?

You already have used access modifiers (public & private) in this chapter. The next chapter will explain Access Modifiers and Non-Access Modifiers in detail.

Modifier Types

Modifiers are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following –

- Java Access Modifiers
- Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following example.

Example

```
public class className {  
    // ...  
}  
  
private boolean myFlag;  
static final double weeks = 9.5;  
protected static final int BOXWIDTH = 42;  
  
public static void main(String[] arguments) {  
    // body of method  
}
```

Access Control Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are –

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionalities.

- The *static* modifier for creating class methods and variables.
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

What is Next?

In the next section, we will be discussing about Basic Operators used in Java Language. The chapter will give you an overview of how these operators can be used during application development.

Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators –

Assume integer variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators –
Assume integer variable A holds 60 and variable B holds 13 then –

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~(bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators –

Assume Boolean variables A holds true and variable B holds false, then –

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
(logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A B) is true

! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true
-----------------	--	-------------------

The Assignment Operators

Following are the assignment operators supported by Java language –

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator.	C = 2 is same as C = C 2

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –
variable x = (expression) ? value if true:value if false

Following is an example –

```
public class Test {  
  
    public static void main(String args[]) {  
        int a, b;  
        a = 10;  
        b = (a == 1) ? 20:30;  
        System.out.println( "Value of b is:" + b );  
  
        b = (a == 10) ? 20:30;  
        System.out.println( "Value of b is:" + b );  
    }  
}
```

This will produce the following result:

Output

```
Value of b is:30  
Value of b is:20
```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as –
(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example

```
public class Test {  
    public static void main(String args[]) {  
  
        String name = "James";  
  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

This will produce the following result:

Output

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example –

Example

```
class Vehicle {}

public class Car extends Vehicle {
    public static void main(String args[]) {

        Vehicle a = new Car();
        boolean result = a instanceof Car;
        System.out.println( result );
    }
}
```

This will produce the following result:

Output

true

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3 * 2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	expression++ expression--	Left to right
Unary	++expression --expression +expression -expression ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right

Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= ^= = <<= >>= >>>=	Right to left

What is Next?

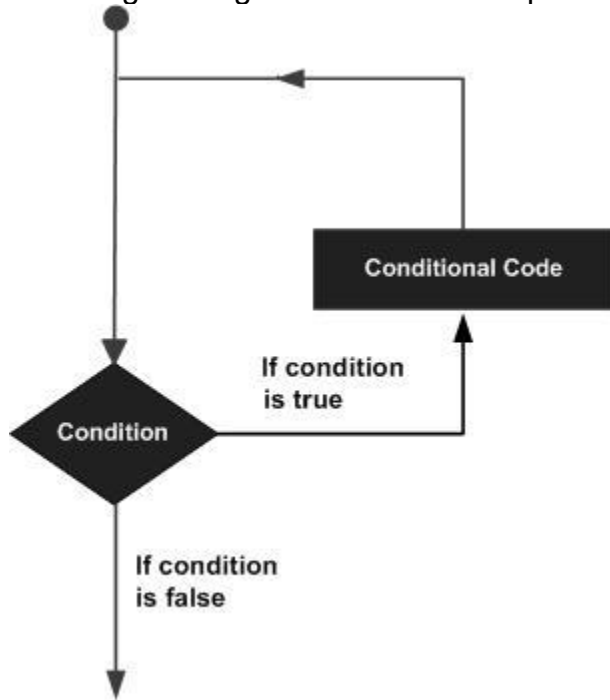
The next chapter will explain about loop control in Java programming. The chapter will describe various types of loops and how these loops can be used in Java program development and for what purposes they are being used.

Loop Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages –



Java programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

No.	Loop & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop Like a while statement, except that it tests the condition at the end of the loop body.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Java supports the following control statements. Click the following links to check their detail.

No.	Control Statement & Description
1	break statement Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Enhanced for loop in Java

As of Java 5, the enhanced for loop was introduced. This is mainly used to traverse collection of elements including arrays.

Syntax

Following is the syntax of enhanced for loop –

```
for(declaration:expression) {  
    // Statements  
}
```

- **Declaration** – The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression** – This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example

```
public class Test {  
    public static void main(String args[]) {  
        int [] numbers = {10, 20, 30, 40, 50};  
        for(int x:numbers ) System.out.print( x + ",");  
        System.out.print("\n");  
        String [] names = {"James", "Larry", "Tom", "Lacy"};  
        for( String name:names ) System.out.print( name + ",");  
    }  
}
```

This will produce the following result:

Output

```
10, 20, 30, 40, 50,  
James, Larry, Tom, Lacy,
```

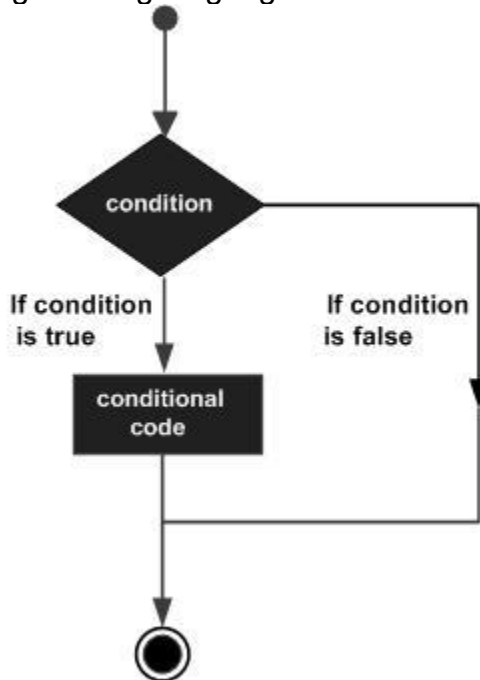
What is Next?

In the following chapter, we will be learning about decision making statements in Java programming.

Decision Making

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision-making structure found in most of the programming languages –



Java programming language provides following types of decision making statements. Click the following links to check their detail.

No.	Statement & Description
1	<u>if statement</u> An if statement consists of a boolean expression followed by one or more statements.
2	<u>if...else statement</u> An if statement can be followed by an optional else statement , which executes when the boolean expression is false.
3	<u>nested if statement</u> You can use one if or else if statement inside another if or else if statement(s).
4	<u>switch statement</u> A switch statement allows a variable to be tested for equality against a list of values.

The ? : Operator

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form –

```
Exp1 ? Exp2:Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon. To determine the value of the whole expression, initially exp1 is evaluated.

- If the value of exp1 is true, then the value of Exp2 will be the value of the whole expression.
- If the value of exp1 is false, then Exp3 is evaluated and its value becomes the value of the entire expression.

What is Next?

In the next chapter, we will discuss about Number class (in the java.lang package) and its subclasses in Java Language.

We will be looking into some of the situations where you will use instantiations of these classes rather than the primitive data types, as well as classes such as formatting, mathematical functions that you need to know about when working with Numbers.

Numbers Class

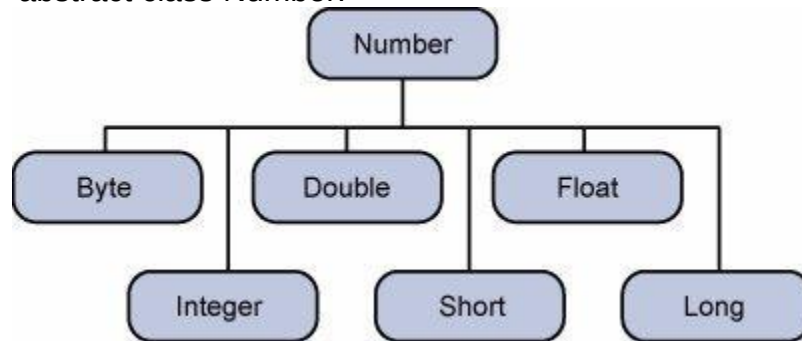
Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double, etc.

Example

```
int i = 5000;  
float gpa = 13.65;  
double mask = 0xaf;
```

However, in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides **wrapper classes**.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



The object of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into object is called **boxing**, and this is taken care by the compiler. Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.

And the Wrapper object will be converted back to a primitive data type, and this process is called unboxing. The **Number** class is part of the java.lang package.

Following is an example of boxing and unboxing –

```
public class Test {  
  
    public static void main(String args[]) {  
        Integer x = 5; // boxes int to an Integer object  
        x = x + 10;    // unboxes the Integer to a int  
        System.out.println(x);  
    }  
}
```

This will produce the following result:

Output

15

When x is assigned an integer value, the compiler boxes the integer because x is integer object. Later, x is unboxed so that they can be added as an integer.

Number Methods

Following is the list of the instance methods that all the subclasses of the Number class implements –

No.	Method & Description
1	<u>xxxValue()</u> Converts the value of <i>this</i> Number object to the xxx data type and returns it.
2	<u>compareTo()</u> Compares <i>this</i> Number object to the argument.
3	<u>equals()</u> Determines whether <i>this</i> number object is equal to the argument.
4	<u>valueOf()</u> Returns an Integer object holding the value of the specified primitive.
5	<u>toString()</u> Returns a String object representing the value of a specified int or Integer.
6	<u>parseInt()</u> This method is used to get the primitive data type of a certain String.
7	<u>abs()</u> Returns the absolute value of the argument.
8	<u>ceil()</u> Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
9	<u>floor()</u> Returns the largest integer that is less than or equal to the argument. Returned as a double.
10	<u>rint()</u> Returns the integer that is closest in value to the argument. Returned as a double.
11	<u>round()</u> Returns the closest long or int, as indicated by the method's return type to the argument.
12	<u>min()</u> Returns the smaller of the two arguments.
13	<u>max()</u> Returns the larger of the two arguments.

14	<u>exp()</u> Returns the base of the natural logarithms, e, to the power of the argument.
15	<u>log()</u> Returns the natural logarithm of the argument.
16	<u>pow()</u> Returns the value of the first argument raised to the power of the second argument.
17	<u>sqrt()</u> Returns the square root of the argument.
18	<u>sin()</u> Returns the sine of the specified double value.
19	<u>cos()</u> Returns the cosine of the specified double value.
20	<u>tan()</u> Returns the tangent of the specified double value.
21	<u>asin()</u> Returns the arcsine of the specified double value.
22	<u>acos()</u> Returns the arccosine of the specified double value.
23	<u>atan()</u> Returns the arctangent of the specified double value.
24	<u>atan2()</u> Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.
25	<u>toDegrees()</u> Converts the argument to degrees.
26	<u>toRadians()</u> Converts the argument to radians.
27	<u>random()</u> Returns a random number.

What is Next?

In the next section, we will be going through the Character class in Java. You will be learning how to use object Characters and primitive data type char in Java.

Character Class

Normally, when we work with characters, we use primitive data types `char`.

Example

```
char ch = 'a';

// Unicode for uppercase Greek omega character
char uniChar = '\u039A';

// an array of chars
char[] charArray = { 'a', 'b', 'c', 'd', 'e' };
```

However in development, we come across situations where we need to use objects instead of primitive data types. In order to achieve this, Java provides wrapper class **Character** for primitive data type `char`.

The `Character` class offers a number of useful class (i.e., static) methods for manipulating characters. You can create a `Character` object with the `Character` constructor –

```
Character ch = new Character('a');
```

The Java compiler will also create a `Character` object for you under some circumstances. For example, if you pass a primitive `char` into a method that expects an object, the compiler automatically converts the `char` to a `Character` for you. This feature is called autoboxing or unboxing, if the conversion goes the other way.

Example

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch = 'a';

// Here primitive 'x' is boxed for method test,
// return is unboxed to char 'c'
char c = test('x');
```

Escape Sequences

A character preceded by a backslash (`\`) is an escape sequence and has a special meaning to the compiler.

The newline character (`\n`) has been used frequently in this course in `System.out.println()` statements to advance to the next line after the string is printed.

Following table shows the Java escape sequences –

Escape Sequence	Description
<code>\t</code>	Inserts a tab in the text at this point.
<code>\b</code>	Inserts a backspace in the text at this point.

\n	Inserts a newline in the text at this point.
\r	Inserts a carriage return in the text at this point.
\f	Inserts a form feed in the text at this point.
\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
\\	Inserts a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Example

If you want to put quotes within quotes, you must use the escape sequence, \", on the interior quotes –

```
public class Test {

    public static void main(String args[]) {
        System.out.println("She said \"Hello!\" to me.");
    }
}
```

This will produce the following result:

Output

She said "Hello!" to me.

Character Methods

Following is the list of the important instance methods that all the subclasses of the Character class implement –

No.	Method & Description
1	<u>isLetter()</u> Determines whether the specified char value is a letter.
2	<u>isDigit()</u> Determines whether the specified char value is a digit.
3	<u>isWhitespace()</u> Determines whether the specified char value is white space.
4	<u>isUpperCase()</u> Determines whether the specified char value is uppercase.
5	<u>isLowerCase()</u>

	Determines whether the specified char value is lowercase.
6	<u>toUpperCase()</u> Returns the uppercase form of the specified char value.
7	<u>toLowerCase()</u> Returns the lowercase form of the specified char value.
8	<u>toString()</u> Returns a String object representing the specified character value that is, a one-character string.

For a complete list of methods, please refer to the `java.lang.Character` API specification.

What is Next?

In the next section, we will be going through the `String` class in Java. You will be learning how to declare and use Strings efficiently as well as some of the important methods in the `String` class.

Strings Class

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings

The most direct way to create a string is to write: `String greeting = "Hello world!";`

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!".

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has 11 constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

Example

```
public class StringDemo {  
    public static void main(String args[]) {  
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };  
        String helloString = new String(helloArray);  
        System.out.println( helloString );  
    }  
}
```

This will produce the following result:

Output

hello.

Note: The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

String Length

Methods used to obtain information about an object are known as **accessor methods**. One accessor method that you can use with strings is the `length()` method, which returns the number of characters contained in the string object.

The following program is an example of **length()**, method String class.

```
public class StringDemo {  
    public static void main(String args[]) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        System.out.println( "String Length is:" + len );  
    }  
}
```

This will produce the following result:

Output

String Length is:17

Concatenating Strings

The String class includes a method for concatenating two strings –

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in –

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in –

```
"Hello," + " world" + "!"
```

which results in –

```
"Hello, world!"
```

Let us look at the following example –

```
public class StringDemo {  
    public static void main(String args[]) {  
        String string1 = "saw I was ";  
        System.out.println("Dot " + string1 + "Tod");  
    }  
}
```

This will produce the following result:

Output

```
Dot saw I was Tod
```

Creating Format Strings

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of –

```
System.out.printf("The value of the float variable is " +  
                  "%f, while the value of the integer " +  
                  "variable is %d, and the string " +  
                  "is %s", floatVar, intVar, stringVar);
```

You can write –

```
String fs;  
fs = String.format("The value of the float variable is " +  
                   "%f, while the value of the integer " +  
                   "variable is %d, and the string " +  
                   "is %s", floatVar, intVar, stringVar);  
System.out.println(fs);
```

String Methods

Here is the list of methods supported by String class –

No.	Method & Description
1	<u>char charAt(int index)</u> Returns the character at the specified index.
2	<u>int compareTo(Object o)</u> Compares this String to another Object.
3	<u>int compareTo(String anotherString)</u> Compares two strings lexicographically.
4	<u>int compareToIgnoreCase(String str)</u> Compares two strings lexicographically, ignoring case differences.
5	<u>String concat(String str)</u> Concatenates the specified string to the end of this string.
6	<u>boolean contentEquals(StringBuffer sb)</u> Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	<u>static String copyValueOf(char[] data)</u> Returns a String that represents the character sequence in the array specified.
8	<u>static String copyValueOf(char[] data, int offset, int count)</u> Returns a String that represents the character sequence in the array specified.
9	<u>boolean endsWith(String suffix)</u> Tests if this string ends with the specified suffix.
10	<u>boolean equals(Object anObject)</u> Compares this string to the specified object.
11	<u>boolean equalsIgnoreCase(String anotherString)</u> Compares this String to another String, ignoring case considerations.
12	<u>byte[] getBytes()</u> Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	<u>byte[] getBytes(String charsetName)</u> Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

14	<u>void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)</u> Copies characters from this string into the destination character array.
15	<u>int hashCode()</u> Returns a hash code for this string.
16	<u>int indexOf(int ch)</u> Returns the index within this string of the first occurrence of the specified character.
17	<u>int indexOf(int ch, int fromIndex)</u> Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	<u>int indexOf(String str)</u> Returns the index within this string of the first occurrence of the specified substring.
19	<u>int indexOf(String str, int fromIndex)</u> Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	<u>String intern()</u> Returns a canonical representation for the string object.
21	<u>int lastIndexOf(int ch)</u> Returns the index within this string of the last occurrence of the specified character.
22	<u>int lastIndexOf(int ch, int fromIndex)</u> Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	<u>int lastIndexOf(String str)</u> Returns the index within this string of the rightmost occurrence of the specified substring.
24	<u>int lastIndexOf(String str, int fromIndex)</u> Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
25	<u>int length()</u> Returns the length of this string.
26	<u>boolean matches(String regex)</u> Tells whether or not this string matches the given regular expression.
27	<u>boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)</u> Tests if two string regions are equal.

28	<u>boolean regionMatches(int toffset, String other, int ooffset, int len)</u> Tests if two string regions are equal.
29	<u>String replace(char oldChar, char newChar)</u> Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
30	<u>String replaceAll(String regex, String replacement)</u> Replaces each substring of this string that matches the given regular expression with the given replacement.
31	<u>String replaceFirst(String regex, String replacement)</u> Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	<u>String[] split(String regex)</u> Splits this string around matches of the given regular expression.
33	<u>String[] split(String regex, int limit)</u> Splits this string around matches of the given regular expression.
34	<u>boolean startsWith(String prefix)</u> Tests if this string starts with the specified prefix.
35	<u>boolean startsWith(String prefix, int toffset)</u> Tests if this string starts with the specified prefix beginning a specified index.
36	<u>CharSequence subSequence(int beginIndex, int endIndex)</u> Returns a new character sequence that is a subsequence of this sequence.
37	<u>String substring(int beginIndex)</u> Returns a new string that is a substring of this string.
38	<u>String substring(int beginIndex, int endIndex)</u> Returns a new string that is a substring of this string.
39	<u>char[] toCharArray()</u> Converts this string to a new character array.
40	<u>String toLowerCase()</u> Converts all of the characters in this String to lower case using the rules of the default locale.
41	<u>String toLowerCase(Locale locale)</u> Converts all of the characters in this String to lower case using the rules of the given Locale.

42	<u>String toString()</u> This object (which is already a string!) is itself returned.
43	<u>String toUpperCase()</u> Converts all of the characters in this String to upper case using the rules of the default locale.
44	<u>String toUpperCase(Locale locale)</u> Converts all of the characters in this String to upper case using the rules of the given Locale.
45	<u>String trim()</u> Returns a copy of the string, with leading and trailing whitespace omitted.
46	<u>static String valueOf(primitive data type x)</u> Returns the string representation of the passed data type argument.

Arrays

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables.

This course introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

Syntax

```
dataType[] arrayRefVar;    // preferred way.  
or  
dataType arrayRefVar[];    // works but not preferred way.
```

Note: The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example

The following code snippets are examples of this syntax –

```
double[] myList;    // preferred way.  
or  
double myList[];    // works but not preferred way.
```

Creating Arrays

You can create an array by using the `new` operator with the following syntax –

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things –

- It creates an array using `new dataType[arraySize]`.
- It assigns the reference of the newly created array to the variable `arrayRefVar`.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below –

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows –

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

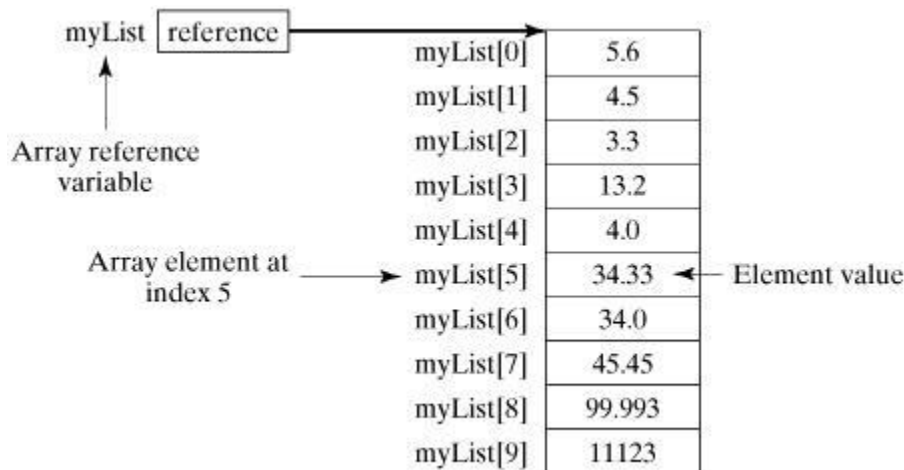
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList –

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Example

Here is a complete example showing how to create, initialize, and process arrays –

```
public class TestArray {  
  
    public static void main(String[] args) {  
        double[] myList = {1.9, 2.9, 3.4, 3.5};  
  
        // Print all the array elements  
        for (int i = 0; i < myList.length; i++) {  
            System.out.println(myList[i] + " ");  
        }  
  
        // Summing all elements  
        double total = 0;  
        for (int i = 0; i < myList.length; i++) {  
            total += myList[i];  
        }  
        System.out.println("Total is " + total);  
    }  
}
```

```

    // Finding the largest element
    double max = myList[0];
    for (int i = 1; i < myList.length; i++) {
        if (myList[i] > max) max = myList[i];
    }
    System.out.println("Max is " + max);
}
}

```

This will produce the following result:

Output

```

1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5

```

The foreach Loops

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example

The following code displays all the elements in the array myList –

```

public class TestArray {
    public static void main(String[] args) {
        double[] myList = {1.9, 2.9, 3.4, 3.5};

        // Print all the array elements
        for (double element:myList) System.out.println(element);
    }
}

```

This will produce the following result:

Output

```

1.9
2.9
3.4
3.5

```

Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an **int** array –

```

public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}

```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2 –

```

printArray(new int[]{3, 1, 2, 6, 4, 2});

```

Returning an Array from a Method

A method may also return an array. For example, the following method returns an array that is the reversal of another array –

Example

```
public static int[] reverse(int[] list) {  
    int[] result = new int[list.length];  
  
    for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {  
        result[j] = list[i];  
    }  
    return result;  
}
```

The Arrays Class

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

No.	Method & Description
1	public static int binarySearch(Object[] a, Object key) Searches the specified array of Object (Byte, Int , double, etc.) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, it returns (– (insertion point + 1)).
2	public static boolean equals(long[] a, long[] a2) Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other primitive data types (Byte, short, Int, etc.)
3	public static void fill(int[] a, int val) Assigns the specified int value to each element of the specified array of ints. The same method could be used by all other primitive data types (Byte, short, Int, etc.)
4	public static void sort(Object[] a) Sorts the specified array of objects into an ascending order, according to the natural ordering of its elements. The same method could be used by all other primitive data types (Byte, short, Int, etc.)

Date and Time

Java provides the **Date** class available in **java.util** package, this class encapsulates the current date and time.

The Date class supports two constructors as shown in the following table.

No.	Constructor & Description
1	Date() This constructor initializes the object with the current date and time.
2	Date(long millisec) This constructor accepts an argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

Following are the methods of the date class.

No.	Method & Description
1	boolean after(Date date) Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false.
2	boolean before(Date date) Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false.
3	Object clone() Duplicates the invoking Date object.
4	int compareTo(Date date) Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date.
5	int compareTo(Object obj) Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException.
6	boolean equals(Object date) Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false.
7	long getTime() Returns the number of milliseconds that have elapsed since January 1, 1970.

8	int hashCode() Returns a hash code for the invoking object.
9	void setTime(long time) Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970.
10	String toString() Converts the invoking Date object into a string and returns the result.

Getting Current Date and Time

This is a very easy method to get current date and time in Java. You can use a simple Date object with *toString()* method to print the current date and time as follows –

Example

```
import java.util.Date;
public class DateDemo {
    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date using toString()
        System.out.println(date.toString());
    }
}
```

This will produce the following result:

Output

on May 04 09:51:52 CDT 2009

Date Comparison

Following are the three ways to compare two dates –

- You can use *getTime()* to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.
- You can use the methods *before()*, *after()*, and *equals()*. Because the 12th of the month comes before the 18th, for example, *new Date(99, 2, 12).before(new Date (99, 2, 18))* returns true.
- You can use the *compareTo()* method, which is defined by the *Comparable* interface and implemented by *Date*.

Date Formatting Using SimpleDateFormat

SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting.

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

public class DateDemo {
    public static void main(String args[]) {
        Date dNow = new Date( );
        SimpleDateFormat ft =
            new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");
        System.out.println("Current Date:" + ft.format(dNow));
    }
}
```

This will produce the following result:

Output

Current Date:Sun 2004.07.18 at 04:14:09 PM PDT

Simple DateFormat Format Codes

To specify the time format, use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following –

Character	Description	Example
G	Era designator	AD
y	Year in four digits	2001
M	Month in year	July or 07
d	Day in month	10
h	Hour in A.M./P.M. (1~12)	12
H	Hour in day (0~23)	22
m	Minute in hour	30
s	Second in minute	55
S	Millisecond	234
E	Day in week	Tuesday
D	Day in year	360
F	Day of week in month	2 (second Wed. in July)

w	Week in year	40
W	Week in month	1
a	A.M./P.M. marker	PM
k	Hour in day (1~24)	24
K	Hour in A.M./P.M. (0~11)	10
z	Time zone	Eastern Standard Time
'	Escape for text	Delimiter
"	Single quote	`

Date Formatting Using printf

Date and time formatting can be done very easily using **printf** method. You use a two-letter format, starting with **t** and ending in one of the letters of the table as shown in the following code.

```
import java.util.Date;
public class DateDemo {
    public static void main(String args[]) {
        Date date = new Date(); // Instantiate a Date object

        // display time and date
        String str = String.format("Current Date/Time:%tc", date);
        System.out.printf(str);
    }
}
```

This will produce the following result:

Output

```
Current Date/Time:Sat Dec 15 16:37:57 MST 2012
```

It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the index of the argument to be formatted.

The index must immediately follow the % and it must be terminated by a \$.

Example

```
import java.util.Date;
public class DateDemo {
    public static void main(String args[]) {
        Date date = new Date(); // Instantiate a Date object

        // display time and date
        System.out.printf("%1$s %2$tB %2$td, %2$tY", "Due date:", date);
    }
}
```

This will produce the following result:

Output

Due date:February 09, 2004

Alternatively, you can use the < flag. It indicates that the same argument as in the preceding format specification should be used again.

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display formatted date
        System.out.printf("%s %tB %<te, %<tY", "Due date:", date);
    }
}
```

This will produce the following result:

Output

Due date:February 09, 2004

Date and Time Conversion Characters

Character	Description	Example
c	Complete date and time	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 date	2004-02-09
D	U.S. formatted date (month/day/year)	02/09/2004
T	24-hour time	18:05:19
r	12-hour time	06:05:19 pm
R	24-hour time, no seconds	18:05
Y	Four-digit year (with leading zeroes)	2004
y	Last two digits of the year (with leading zeroes)	04
C	First two digits of the year (with leading zeroes)	20
B	Full month name	February
b	Abbreviated month name	Feb

m	Two-digit month (with leading zeroes)	02
d	Two-digit day (with leading zeroes)	03
e	Two-digit day (without leading zeroes)	9
A	Full weekday name	Monday
a	Abbreviated weekday name	Mon
j	Three-digit day of year (with leading zeroes)	069
H	Two-digit hour (with leading zeroes), between 00 and 23	18
k	Two-digit hour (without leading zeroes), between 0 and 23	18
l	Two-digit hour (with leading zeroes), between 01 and 12	06
l	Two-digit hour (without leading zeroes), between 1 and 12	6
M	Two-digit minutes (with leading zeroes)	05
S	Two-digit seconds (with leading zeroes)	19
L	Three-digit milliseconds (with leading zeroes)	047
N	Nine-digit nanoseconds (with leading zeroes)	047000000
P	Uppercase morning or afternoon marker	PM
p	Lowercase morning or afternoon marker	pm
z	RFC 822 numeric offset from GMT	-0800
Z	Time zone	PST
s	Seconds since 1970-01-01 00:00:00 GMT	1078884319
Q	Milliseconds since 1970-01-01 00:00:00 GMT	1078884319047

There are other useful classes related to Date and time. For more details, you can refer to [Java Standard documentation](#).

Parsing Strings into Dates

The `SimpleDateFormat` class has some additional methods, notably `parse()`, which tries to parse a string according to the format stored in the given `SimpleDateFormat` object.

Example

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

public class DateDemo {
    public static void main(String args[]) {
        SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");
        String input = args.length == 0 ? "1818-11-11":args[0];

        System.out.print(input + " Parses as ");
        Date t;
        try {
            t = ft.parse(input);
            System.out.println(t);
        } catch (ParseException e) {
            System.out.println("Unparseable using " + ft);
        }
    }
}
```

A sample run of the above program would produce the following result:

Output

1818-11-11 Parses as Wed Nov 11 00:00:00 EST 1818

Sleeping for a While

You can sleep for any period of time from one millisecond up to the lifetime of your computer. For example, the following program would sleep for 3 seconds –

```
import java.util.*;
public class SleepDemo {
    public static void main(String args[]) {
        try {
            System.out.println(new Date( ) + "\n");
            Thread.sleep(5*60*10);
            System.out.println(new Date( ) + "\n");
        } catch (Exception e) {
            System.out.println("Got an exception!");
        }
    }
}
```

This will produce the following result:

Output

Sun May 03 18:04:41 GMT 2009
Sun May 03 18:04:51 GMT 2009

Measuring Elapsed Time

Sometimes, you may need to measure point in time in milliseconds. So let's re-write the above example once again –

```
import java.util.*;
public class DiffDemo {
    public static void main(String args[]) {
        try {
            long start = System.currentTimeMillis( );
            System.out.println(new Date( ) + "\n");

            Thread.sleep(5*60*10);
            System.out.println(new Date( ) + "\n");

            long end = System.currentTimeMillis( );
            long diff = end - start;
            System.out.println("Difference is:" + diff);
        } catch (Exception e) {
            System.out.println("Got an exception!");
        }
    }
}
```

This will produce the following result:

Output

```
Sun May 03 18:16:51 GMT 2009
Sun May 03 18:16:57 GMT 2009
Difference is:5993
```

GregorianCalendar Class

GregorianCalendar is a concrete implementation of a Calendar class that implements the normal Gregorian calendar with which you are familiar. We did not discuss Calendar class in this course, you can look up standard Java documentation for this.

The **getInstance()** method of Calendar returns a GregorianCalendar initialized with the current date and time in the default locale and time zone. GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for GregorianCalendar objects –

No.	Constructor & Description
1	GregorianCalendar() Constructs a default GregorianCalendar using the current time in the default time zone with the default locale.
2	GregorianCalendar(int year, int month, int date) Constructs a GregorianCalendar with the given date set in the default time zone with the default locale.
3	GregorianCalendar(int year, int month, int date, int hour, int minute) Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.

4	GregorianCalendar(int year, int month, int date, int hour, int minute, int second) Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.
5	GregorianCalendar(Locale aLocale) Constructs a GregorianCalendar based on the current time in the default time zone with the given locale.
6	GregorianCalendar(TimeZone zone) Constructs a GregorianCalendar based on the current time in the given time zone with the default locale.
7	GregorianCalendar(TimeZone zone, Locale aLocale) Constructs a GregorianCalendar based on the current time in the given time zone with the given locale.

Here is the list of few useful support methods provided by GregorianCalendar class –

No.	Method & Description
1	void add(int field, int amount) Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.
2	protected void computeFields() Converts UTC as milliseconds to time field values.
3	protected void computeTime() Overrides Calendar Converts time field values to UTC as milliseconds.
4	boolean equals(Object obj) Compares this GregorianCalendar to an object reference.
5	int get(int field) Gets the value for a given time field.
6	int getActualMaximum(int field) Returns the maximum value that this field could have, given the current date.
7	int getActualMinimum(int field) Returns the minimum value that this field could have, given the current date.
8	int getGreatestMinimum(int field) Returns highest minimum value for the given field if varies.
9	Date getGregorianChange() Gets the Gregorian Calendar change date.

10	int getLeastMaximum(int field) Returns lowest maximum value for the given field if varies.
11	int getMaximum(int field) Returns maximum value for the given field.
12	Date getTime() Gets this Calendar's current time.
13	long getTimeInMillis() Gets this Calendar's current time as a long.
14	TimeZone getTimeZone() Gets the time zone.
15	int getMinimum(int field) Returns minimum value for the given field.
16	int hashCode() Overrides hashCode.
17	boolean isLeapYear(int year) Determines if the given year is a leap year.
18	void roll(int field, boolean up) Adds or subtracts (up/down) a single unit of time on the given time field without changing larger fields.
19	void set(int field, int value) Sets the time field with the given value.
20	void set(int year, int month, int date) Sets the values for the fields year, month, and date.
21	void set(int year, int month, int date, int hour, int minute) Sets the values for the fields year, month, date, hour, and minute.
22	void set(int year, int month, int date, int hour, int minute, int second) Sets the values for the fields year, month, date, hour, minute, and second.
23	void setGregorianChange(Date date) Sets the GregorianCalendar change date.
24	void setTime(Date date) Sets this Calendar's current time with the given Date.
25	void setTimeInMillis(long millis)

	Sets this Calendar's current time from the given long value.
26	void setTimeZone(TimeZone value) Sets the time zone with the given time zone value.
27	String toString() Returns a string representation of this calendar.

Example

```
import java.util.*;
public class GregorianCalendarDemo {
    public static void main(String args[]) {
        String months[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                           "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

        int year;
        // Create a Gregorian calendar initialized
        // with the current date and time in the
        // default locale and timezone.

        GregorianCalendar gcalendar = new GregorianCalendar();

        // Display current time and date information.
        System.out.print("Date:");
        System.out.print(months[gcalendar.get(Calendar.MONTH)]);
        System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
        System.out.println(year = gcalendar.get(Calendar.YEAR));
        System.out.print("Time:");
        System.out.print(gcalendar.get(Calendar.HOUR) + ":");
        System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
        System.out.println(gcalendar.get(Calendar.SECOND));

        // Test if the current year is a leap year
        if(gcalendar.isLeapYear(year)) {
            System.out.println("The current year is a leap year");
        } else {
            System.out.println("The current year is not a leap year");
        }
    }
}
```

This will produce the following result:

Output

```
Date:Apr 22 2009
Time:11:25:27
The current year is not a leap year
```

For a complete list of constant available in Calendar class, you can refer the standard Java documentation.

Regular Expressions

Java provides the `java.util.regex` package for pattern matching with regular expressions. Java regular expressions are very similar to the Perl programming language and very easy to learn. A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

The `java.util.regex` package primarily consists of the following three classes –

- **Pattern Class** – A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static **`compile()`** methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
- **Matcher Class** – A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the **`matcher()`** method on a Pattern object.
- **PatternSyntaxException** – A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Capturing Groups

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression `(dog)` creates a single group containing the letters "d", "o", and "g".

Capturing groups are numbered by counting their opening parentheses from the left to the right. In the expression `((A)(B(C)))`, for example, there are four such groups –

- `((A)(B(C)))`
- `(A)`
- `(B(C))`
- `(C)`

To find out how many groups are present in the expression, call the `groupCount` method on a matcher object. The `groupCount` method returns an **`int`** showing the number of capturing groups present in the matcher's pattern.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by `groupCount`.

Following example illustrates how to find a digit string from the given alphanumeric string –

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
    public static void main( String args[] ) {
        // String to be scanned to find the pattern.
        String line = "This order was placed for QT3000! OK?";
        String pattern = "(.*) (\\d+) (.*)";
```

```

// Create a Pattern object
Pattern r = Pattern.compile(pattern);

// Now create matcher object.
Matcher m = r.matcher(line);
if (m.find( )) {
    System.out.println("Found value:" + m.group(0) );
    System.out.println("Found value:" + m.group(1) );
    System.out.println("Found value:" + m.group(2) );
} else {
    System.out.println("NO MATCH");
}
}
}

```

This will produce the following result:

Output

```

Found value:This order was placed for QT3000! OK?
Found value:This order was placed for QT300
Found value:0

```

Regular Expression Syntax

Here is the table listing down all the regular expression metacharacter syntax available in Java –

Subexpression	Matches
^	Matches the beginning of the line.
\$	Matches the end of the line.
.	Matches any single character except newline. Using m option allows it to match the newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets.
\A	Beginning of the entire string.
\Z	End of the entire string.
\z	End of the entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of the preceding expression.
re+	Matches 1 or more of the previous thing.
re?	Matches 0 or 1 occurrence of the preceding expression.

<code>re{ n}</code>	Matches exactly n number of occurrences of the preceding expression.
<code>re{ n,}</code>	Matches n or more occurrences of the preceding expression.
<code>re{ n, m}</code>	Matches at least n and at most m occurrences of the preceding expression.
<code>a b</code>	Matches either a or b.
<code>(re)</code>	Groups regular expressions and remembers the matched text.
<code>(? : re)</code>	Groups regular expressions without remembering the matched text.
<code>(?> re)</code>	Matches the independent pattern without backtracking.
<code>\w</code>	Matches the word characters.
<code>\W</code>	Matches the nonword characters.
<code>\s</code>	Matches the whitespace. Equivalent to <code>[\t\n\r\f]</code> .
<code>\S</code>	Matches the nonwhitespace.
<code>\d</code>	Matches the digits. Equivalent to <code>[0-9]</code> .
<code>\D</code>	Matches the nondigits.
<code>\A</code>	Matches the beginning of the string.
<code>\Z</code>	Matches the end of the string. If a newline exists, it matches just before newline.
<code>\z</code>	Matches the end of the string.
<code>\G</code>	Matches the point where the last match finished.
<code>\n</code>	Back-reference to capture group number "n".
<code>\b</code>	Matches the word boundaries when outside the brackets. Matches the backspace (0x08) when inside the brackets.
<code>\B</code>	Matches the nonword boundaries.
<code>\n, \t, etc.</code>	Matches newlines, carriage returns, tabs, etc.
<code>\Q</code>	Escape (quote) all characters up to <code>\E</code> .
<code>\E</code>	Ends quoting begun with <code>\Q</code> .

Methods of the Matcher Class

Here is a list of useful instance methods –

Index Methods

Index methods provide useful index values that show precisely where the match was found in the input string –

No.	Method & Description
1	public int start() Returns the start index of the previous match.
2	public int start(int group) Returns the start index of the subsequence captured by the given group during the previous match operation.
3	public int end() Returns the offset after the last character matched.
4	public int end(int group) Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Study Methods

Study methods review the input string and return a Boolean indicating whether or not the pattern is found –

No.	Method & Description
1	public boolean lookingAt() Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2	public boolean find() Attempts to find the next subsequence of the input sequence that matches the pattern.
3	public boolean find(int start) Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4	public boolean matches() Attempts to match the entire region against the pattern.

Replacement Methods

Replacement methods are useful methods for replacing text in an input string –

No.	Method & Description
1	public Matcher appendReplacement(StringBuffer sb, String replacement) Implements a non-terminal append-and-replace step.
2	public StringBuffer appendTail(StringBuffer sb) Implements a terminal append-and-replace step.
3	public String replaceAll(String replacement) Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
4	public String replaceFirst(String replacement) Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
5	public static String quoteReplacement(String s) Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement <i>s</i> in the appendReplacement method of the Matcher class.

The start and end Methods

Following is the example that counts the number of times the word "cat" appears in the input string –

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT = "cat cat cat cattie cat";

    public static void main( String args[] ) {
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);    // get a matcher object
        int count = 0;

        while(m.find()) {
            count++;
            System.out.println("Match number "+count);
            System.out.println("start():"+m.start());
            System.out.println("end():"+m.end());
        }
    }
}
```

This will produce the following result:

Output

Match number 1

```
start():0
end():3
Match number 2
start():4
end():7
Match number 3
start():8
end():11
Match number 4
start():19
end():22
```

You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred.

The start method returns the start index of the subsequence captured by the given group during the previous match operation, and the end returns the index of the last character matched, plus one.

The matches and lookingAt Methods

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality –

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;
    public static void main( String args[] ) {
        pattern = Pattern.compile(REGEX);
        matcher = pattern.matcher(INPUT);
        System.out.println("Current REGEX is:"+REGEX);
        System.out.println("Current INPUT is:"+INPUT);

        System.out.println("lookingAt():"+matcher.lookingAt());
        System.out.println("matches():"+matcher.matches());
    }
}
```

This will produce the following result:

Output

```
Current REGEX is:foo
Current INPUT is:fooooooooooooooooooooo
lookingAt():true
matches():false
```

The replaceFirst and replaceAll Methods

The replaceFirst and replaceAll methods replace the text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences.

Here is the example explaining the functionality –

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. " + "All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);

        Matcher m = p.matcher(INPUT); // get a matcher object
        INPUT = m.replaceAll(REPLACE);
        System.out.println(INPUT);
    }
}
```

This will produce the following result:

Output

The cat says meow. All cats say meow.

The appendReplacement and appendTail Methods

The Matcher class also provides appendReplacement and appendTail methods for text replacement.

Here is the example explaining the functionality –

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches {
    private static String REGEX = "a*b";
    private static String INPUT = "aabfooaabfooabfoob";
    private static String REPLACE = "-";
    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);

        Matcher m = p.matcher(INPUT); // get a matcher object
        StringBuffer sb = new StringBuffer();
        while(m.find()) {
            m.appendReplacement(sb, REPLACE);
        }
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
```

This will produce the following result:

Output

-foo-foo-foo-

PatternSyntaxException Class Methods

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong –

No.	Method & Description
1	public String getDescription() Retrieves the description of the error.
2	public int getIndex() Retrieves the error index.
3	public String getPattern() Retrieves the erroneous regular expression pattern.
4	public String getMessage() Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern.

Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println()` method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, and apply method abstraction in the program design.

Creating Method

Considering the following example to explain the syntax of a method –

Syntax

```
public static int methodName(int a, int b) {  
    // body  
}
```

Here,

- **public static** – modifier
- **int** – return type
- **methodName** – name of the method
- **a, b** – formal parameters
- **int a, int b** – list of parameters

Method definition consists of a method header and a method body. The same is shown in the following syntax –

Syntax

```
modifier returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

The syntax shown above includes –

- **modifier** – It defines the access type of the method and it is optional to use.
- **returnType** – Method may return a value.
- **nameOfMethod** – This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** – The method body defines what the method does with the statements.

Example

Here is the source code of the above defined method called **min()**. This method takes two parameters **num1** and **num2** and returns the maximum between the two –

```
/** the snippet returns the minimum between two numbers */  
public static int minFunction(int n1, int n2) {  
    int min;  
    if (n1 > n2)  
        min = n2;  
    else  
        min = n1;  
  
    return min;  
}
```

Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when –

- the return statement is executed.
- it reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example –

```
System.out.println("This is www.abc.com!");
```

The method returning value can be understood by the following example –

```
int result = sum(6, 9);
```

Following is the example to demonstrate how to define a method and how to call it –

Example

```
public class ExampleMinNumber {
    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        int c = minFunction(a, b);
        System.out.println("Minimum Value = " + c);
    }

    /** returns the minimum of two numbers */
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }
}
```

This will produce the following result:

Output

Minimum value = 6

The void Keyword

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method, which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7)*;. It is a Java statement which ends with a semicolon as shown in the following example.

```
public class ExampleVoid {
    public static void main(String[] args) {
        methodRankPoints(255.7);
    }

    public static void methodRankPoints(double points) {
        if (points >= 202.5) {
```

```

        System.out.println("Rank:A1");
    }else if (points >= 122.4) {
        System.out.println("Rank:A2");
    }else {
        System.out.println("Rank:A3");
    }
}
}

```

This will produce the following result:

Output

Rank:A1

Passing Parameters by Value

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

Example

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```

public class swappingExample {
    public static void main(String[] args) {
        int a = 30;
        int b = 45;
        System.out.println("Before swapping, a = " + a + " and b = " + b);
        swapFunction(a, b); // Invoke the swap method
        System.out.println("\n**Now, Before and After swapping values will be same here**");
        System.out.println("After swapping, a = " + a + " and b is " + b);
    }
    public static void swapFunction(int a, int b) {
        System.out.println("Before swapping(Inside), a = " + a + " b = " + b);

        // Swap n1 with n2
        int c = a;
        a = b;
        b = c;
        System.out.println("After swapping(Inside), a = " + a + " b = " + b);
    }
}

```

This will produce the following result:

Output

```

Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

```

```

**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45

```

Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

Let's consider the example discussed earlier for finding minimum numbers of integer type. If, let's say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters. The following example explains the same –

Example

```
public class ExampleOverloading {
    public static void main(String[] args) {
        int a = 11, b = 6;
        double c = 7.3, d = 9.4;
        int result1 = minFunction(a, b);

        // same function name with different parameters
        double result2 = minFunction(c, d);
        System.out.println("Minimum Value = " + result1);
        System.out.println("Minimum Value = " + result2);
    }

    // for integer
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }

    // for double
    public static double minFunction(double n1, double n2) {
        double min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }
}
```

This will produce the following result:

Output

```
Minimum Value = 6
Minimum Value = 7.3
```

Overloading methods makes program readable. Here, two methods are given by the same name but with different parameters. The minimum number from integer and double types is the result.

Using Command-Line Arguments

Sometimes you will want to pass some information into a program when you run it. This is accomplished by passing command-line arguments to `main()`.

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to `main()`.

Example

The following program displays all of the command-line arguments that it is called with –

```
public class CommandLine {
    public static void main(String args[]) {
        for(int i = 0; i<args.length; i++) {
            System.out.println("args[" + i + "]: " + args[i]);
        }
    }
}
```

Try executing this program as shown here –

```
$java CommandLine this is a command line 200 -100
```

This will produce the following result:

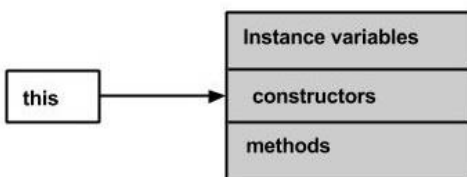
Output

```
args[0]:this
args[1]:is
args[2]:a
args[3]:command
args[4]:line
args[5]:200
args[6]:-100
```

The this keyword

this is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

Note: The keyword *this* is used only within instance methods or constructors



In general, the keyword *this* is used to –

- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
class Student {
    int age;
    Student(int age) {
        this.age = age;
    }
}
```

- Call one type of constructor (Parameterized constructor or default) from other in a class. It is known as explicit constructor invocation.

```
class Student {  
    int age  
    Student() {  
        this(20);  
    }  
  
    Student(int age) {  
        this.age = age;  
    }  
}
```

Example

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the following program in a file with the name, **This_Example.java**.

```
public class This_Example {  
    // Instance variable num  
    int num = 10;  
  
    This_Example() {  
        System.out.println("This is an example program on keyword this");  
    }  
  
    This_Example(int num) {  
        // Invoking the default constructor  
        this();  
  
        // Assigning the local variable num to the instance variable num  
        this.num = num;  
    }  
  
    public void greet() {  
        System.out.println("Hi Welcome to abc.com");  
    }  
  
    public void print() {  
        // Local variable num  
        int num = 20;  
  
        // Printing the local variable  
        System.out.println("value of local variable num is:"+num);  
  
        // Printing the instance variable  
        System.out.println("value of instance variable num is:"+this.num);  
  
        // Invoking the greet method of a class  
        this.greet();  
    }  
  
    public static void main(String[] args) {  
        // Instantiating the class  
        This_Example obj1 = new This_Example();  
  
        // Invoking the print method
```

```

    obj1.print();

    // Passing a new value to the num variable through Parameterized constructor
    This_Example obj2 = new This_Example(30);

    // Invoking the print method again
    obj2.print();
}
}

```

This will produce the following result:

Output

```

This is an example program on keyword this
value of local variable num is:20
value of instance variable num is:10
Hi Welcome to abc.com
This is an example program on keyword this
value of local variable num is:20
value of instance variable num is:30
Hi Welcome to abc.com

```

Variable Arguments(var-args)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method.

The parameter in the method is declared as follows –

```

typeName... parameterName

```

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example

```

public class VarargsDemo {
    public static void main(String args[]) {
        // Call method with variable args
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }
    public static void printMax( double... numbers) {
        if (numbers.length == 0) {
            System.out.println("No argument passed");
            return;
        }
        double result = numbers[0];

        for (int i = 1; i < numbers.length; i++)
            if (numbers[i] > result) result = numbers[i];
        System.out.println("The max value is " + result);
    }
}

```

This will produce the following result:

Output

```

The max value is 56.5
The max value is 3.0

```

The finalize() Method

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize()**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize() to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize() method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the finalize() method, you will specify those actions that must be performed before an object is destroyed.

The finalize() method has this general form –

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

Here, the keyword protected is a specifier that prevents access to finalize() by code defined outside its class.

This means that you cannot know when or even if finalize() will be executed. For example, if your program ends before garbage collection occurs, finalize() will not execute.

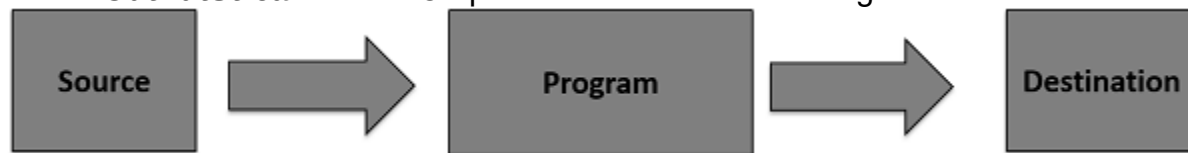
Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams –

- **InPutStream** – The InputStream is used to read data from a source.
- **OutPutStream** – The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this course covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one –

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file –

```
import java.io.*;
public class CopyFile {
    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Now let's have a file **input.txt** with the following content –
This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following –

```
$javac CopyFile.java  
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file –

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

Now let's have a file **input.txt** with the following content –
This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following –

```
$javac CopyFile.java  
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams –

- **Standard Input** – This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" –

```
import java.io.*;
public class ReadConsole {
    public static void main(String args[]) throws IOException {
        InputStreamReader cin = null;

        try {
            cin = new InputStreamReader(System.in);
            System.out.println("Enter characters, 'q' to quit.");
            char c;
            do {
                c = (char) cin.read();
                System.out.print(c);
            } while(c != 'q');
        } finally {
            if (cin != null) {
                cin.close();
            }
        }
    }
}
```

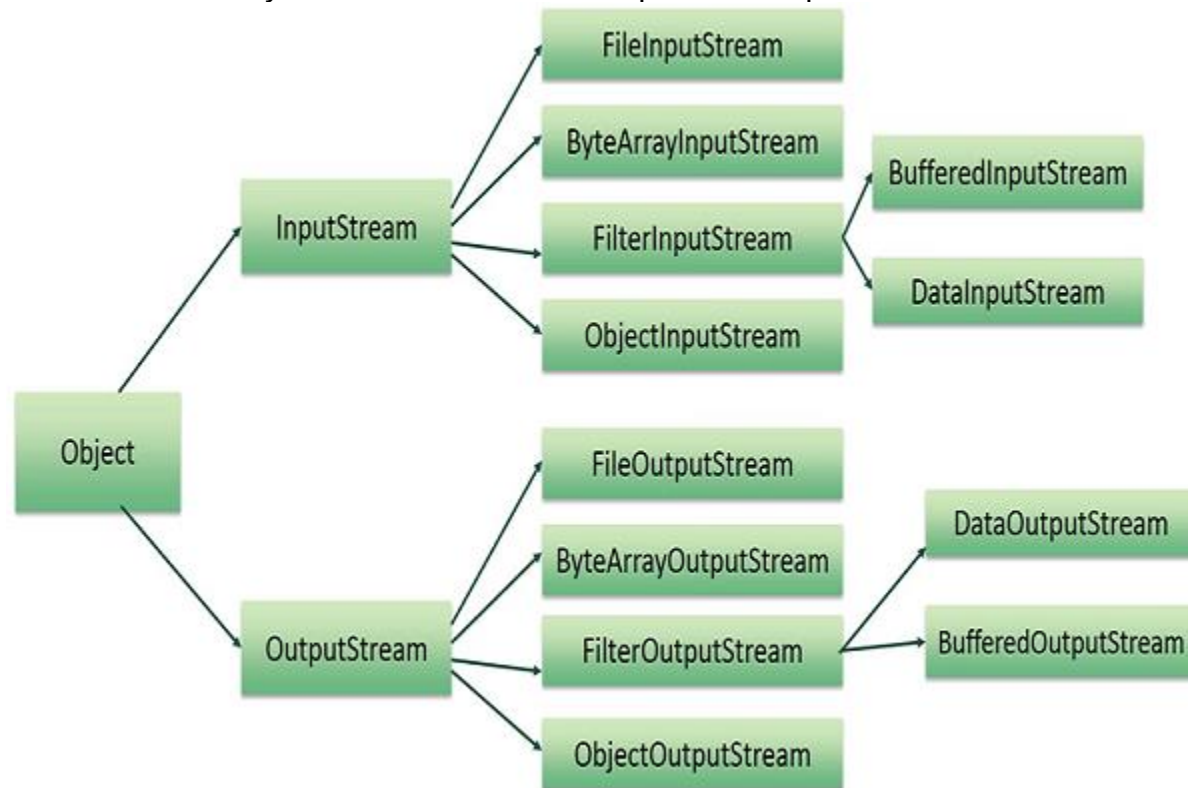
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' –

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this course.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file –

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows –

```
File f = new File("C:/java/hello");  
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

No.	Method & Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{} This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	public int read(byte[] r) throws IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	public int available() throws IOException{} Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links –

- [ByteArrayInputStream](#)
- [DataInputStream](#)

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file –

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows –

```
File f = new File("C:/java/hello");  
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

No.	Method & Description
1	public void close() throws IOException{ This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException { This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public void write(int w)throws IOException{ This methods writes the specified byte to the output stream.
4	public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links –

- [ByteArrayOutputStream](#)
- [DataOutputStream](#)

Following is the example to demonstrate InputStream and OutputStream –

```
import java.io.*;
public class FileStreamTest {
    public static void main(String args[]) {

        try {
            byte bWrite [] = {11,21,3,40,5};
            OutputStream os = new FileOutputStream("test.txt");
            for(int x = 0; x < bWrite.length ; x++) {
                os.write( bWrite[x] );    // writes the bytes
            }
            os.close();

            InputStream is = new FileInputStream("test.txt");
            int size = is.available();

            for(int i = 0; i < size; i++) {
                System.out.print((char)is.read() + " ");
            }
            is.close();
        } catch (IOException e) {
            System.out.print("Exception");
        }
    }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories –

- The **mkdir()** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory –

```
import java.io.File;
public class CreateDir {
    public static void main(String args[]) {
        String dirname = "/tmp/user/java/bin";
        File d = new File(dirname);

        // Create directory now.
        d.mkdirs();
    }
}
```

Compile and execute the above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list()** method provided by **File** object to list down all the files and directories available in a directory as follows –

```
import java.io.File;
public class ReadDir {
    public static void main(String[] args) {
        File file = null;
        String[] paths;

        try {
            // create new file object
            file = new File("/tmp");

            // array of files and directory
            paths = file.list();

            // for each name in the path array
            for(String path:paths) {
                // prints filename and directory name
                System.out.println(path);
            }
        } catch (Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

This will produce the following result based on the directories and files available in your **/tmp** directory –

Output

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions** – A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

```
import java.io.File;
import java.io.FileReader;

public class FileNotFound_Demo {
    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FileNotFound_Demo.java
FileNotFound_Demo.java:8:error:unreported exception
FileNotFoundException; must be caught or declared to be thrown
    FileReader fr = new FileReader(file);
                      ^
1 error
```

Note: Since the methods **read()** and **close()** of **FileReader** class throws **IOException**, you can observe that the compiler notifies to handle **IOException**, along with **FileNotFoundException**.

- **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsException* occurs.

```
public class Unchecked_Demo {  
    public static void main(String args[]) {  
        int num[] = {1, 2, 3, 4};  
        System.out.println(num[5]);  
    }  
}
```

If you compile and execute the above program, you will get the following exception.

Output

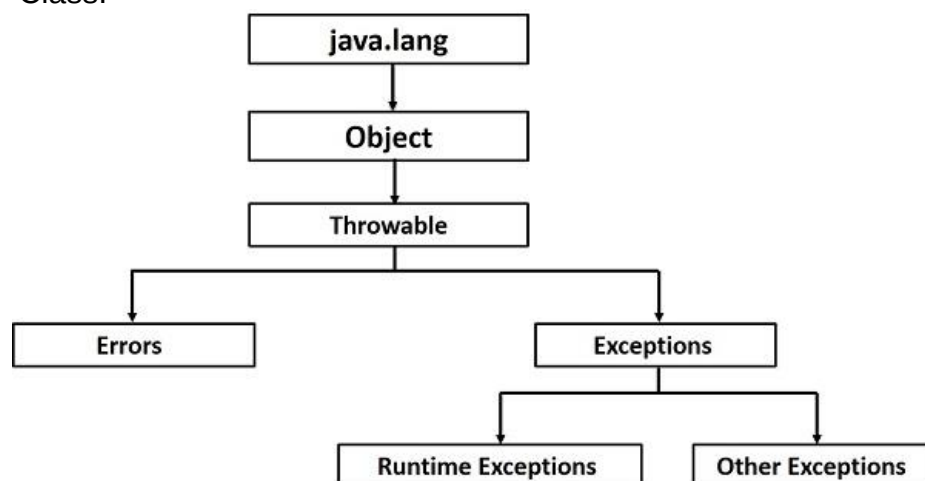
```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:5  
    at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

- **Errors** – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the **java.lang.Exception** class. The exception class is a subclass of the **Throwable** class. Other than the exception class there is another subclass called **Error** which is derived from the **Throwable** class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors. The **Exception** class has two main subclasses: **IOException** class and **RuntimeException** Class.



Exceptions Methods

Following is the list of important methods available in the Throwable class.

No.	Method & Description
1	public String getMessage() Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	public Throwable getCause() Returns the cause of the exception as represented by a Throwable object.
3	public String toString() Returns the name of the class concatenated with the result of getMessage().
4	public void printStackTrace() Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	public StackTraceElement [] getStackTrace() Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	public Throwable fillInStackTrace() Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

Syntax

```
try {  
    // Protected code  
} catch (ExceptionName e1) {  
    // Catch block  
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked.

If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name:ExcepTest.java
import java.io.*;

public class ExcepTest {
    public static void main(String args[]) {
        try {
            int a[] = new int[2];
            System.out.println("Access element three:" + a[3]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception thrown :" + e);
        }
        System.out.println("Out of the block");
    }
}
```

This will produce the following result:

Output

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following –

Syntax

```
try {
    // Protected code
} catch (ExceptionType1 e1) {
    // Catch block
} catch (ExceptionType2 e2) {
    // Catch block
} catch (ExceptionType3 e3) {
    // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches `ExceptionType1`, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Here is code segment showing how to use multiple try/catch statements.

```
try {
    file = new FileInputStream(fileName);
    x = (byte) file.read();
} catch (IOException i) {
    i.printStackTrace();
    return -1;
} catch (FileNotFoundException f) // Not valid! {
    f.printStackTrace();
    return -1;
}
```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it –

```
catch (IOException|FileNotFoundException ex) {
    logger.log(ex);
    throw ex;
```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException –

```
import java.io.*;
public class className {
    public void deposit(double amount) throws RemoteException {
        // Method implementation
        throw new RemoteException();
    }
    // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException –

```
import java.io.*;
public class className {
    public void withdraw(double amount) throws RemoteException,
        InsufficientFundsException {
        // Method implementation
    }
    // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax –
Syntax

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
} finally {  
    // The finally block always executes.  
}
```

Example

```
public class ExcepTest {  
    public static void main(String args[]) {  
        int a[] = new int[2];  
        try {  
            System.out.println("Access element three:" + a[3]);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Exception thrown :" + e);  
        } finally {  
            a[0] = 6;  
            System.out.println("First element value:" + a[0]);  
            System.out.println("The finally statement is executed");  
        }  
    }  
}
```

This will produce the following result:

Output

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3  
First element value:6  
The finally statement is executed
```

Note the following –

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

The try-with-resources

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

Example

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {
    public static void main(String args[]) {
        FileReader fr = null;
        try {
            File file = new File("file.txt");
            fr = new FileReader(file); char [] a = new char[50];
            fr.read(a); // reads the content to the array
            for(char c:a) System.out.print(c); // prints the characters one by one
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                fr.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

try-with-resources, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

Syntax

```
try(FileReader fr = new FileReader("file path")) {
    // use the resource
} catch () {
    // body of catch
}
```

Following is the program that reads the data in a file using try-with-resources statement.

```
import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {
    public static void main(String args[]) {
        try (FileReader fr = new FileReader("E://file.txt")) {
            char [] a = new char[50];
            fr.read(a);    // reads the content to the array
            for(char c:a) System.out.print(c);    // prints the characters one by one
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Following points are to be kept in mind while working with try-with-resources statement.

- To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
- You can declare more than one class in try-with-resources statement.
- While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block.
- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes –

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below –

```
class MyException extends Exception {
}
```

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception.

An exception class is like any other class, containing useful fields and methods.

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception {
    private double amount;

    public InsufficientFundsException(double amount) {
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount {
    private double balance;
    private int number;

    public CheckingAccount(int number) {
        this.number = number;
    }

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

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount <= balance) {
            balance -= amount;
        } else {
            double needs = amount - balance;
            throw new InsufficientFundsException(needs);
        }
    }

    public double getBalance() {
        return balance;
    }

    public int getNumber() {
        return number;
    }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo {
    public static void main(String [] args) {
        CheckingAccount c = new CheckingAccount(101);
        System.out.println("Depositing $500...");
        c.deposit(500.00);

        try {
            System.out.println("\nWithdrawing $100...");
            c.withdraw(100.00);
            System.out.println("\nWithdrawing $600...");
            c.withdraw(600.00);
        } catch (InsufficientFundsException e) {
            System.out.println("Sorry, but you are short $" + e.getAmount());
            e.printStackTrace();
        }
    }
}
```

Compile all the above three files and run BankDemo. This will produce the following result:

Output

Depositing \$500...

Withdrawing \$100...

Withdrawing \$600...

Sorry, but you are short \$200.0

InsufficientFundsException

at CheckingAccount.withdraw(CheckingAccount.java:25)

at BankDemo.main(BankDemo.java:13)

Common Exceptions

In Java, it is possible to define two categories of Exceptions and Errors.

- **JVM Exceptions** – These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.
- **Programmatic Exceptions** – These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

Inner classes

In this chapter, we will discuss inner classes of Java.

Nested Classes

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.

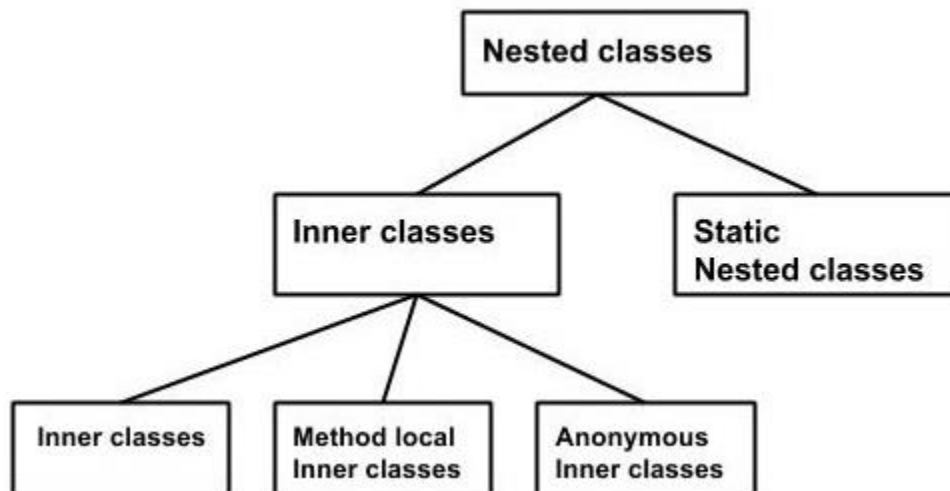
Syntax

Following is the syntax to write a nested class. Here, the class **Outer_Demo** is the outer class and the class **Inner_Demo** is the nested class.

```
class Outer_Demo {  
    class Inner_Demo {  
    }  
}
```

Nested classes are divided into two types –

- **Non-static nested classes** – These are the non-static members of a class.
- **Static nested classes** – These are the static members of a class.



Inner Classes (Non-static Nested Classes)

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are –

- Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

```
class Outer_Demo {
    int num;

    // inner class
    private class Inner_Demo {
        public void print() {
            System.out.println("This is an inner class");
        }
    }

    // Accessing the inner class from the method within
    void display_Inner() {
        Inner_Demo inner = new Inner_Demo();
        inner.print();
    }
}

public class My_class {
    public static void main(String args[]) {
        // Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();

        // Accessing the display_Inner() method.
        outer.display_Inner();
    }
}
```

Here you can observe that **Outer_Demo** is the outer class, **Inner_Demo** is the inner class, **display_Inner()** is the method inside which we are instantiating the inner class, and this method is invoked from the **main** method.

If you compile and execute the above program, you will get the following result:

Output

This is an inner class.

Accessing the Private Members

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, **getValue()**, and finally from another class (from which you want to access the private members) call the **getValue()** method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

```
Outer_Demo outer = new Outer_Demo();
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

The following program shows how to access the private members of a class using inner class.

```
class Outer_Demo {
    // private variable of the outer class
    private int num = 175;

    // inner class
    public class Inner_Demo {
        public int getNum() {
            System.out.println("This is the getnum method of the inner class");
            return num;
        }
    }
}

public class My_class2 {
    public static void main(String args[]) {
        // Instantiating the outer class
        Outer_Demo outer = new Outer_Demo();

        // Instantiating the inner class
        Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
        System.out.println(inner.getNum());
    }
}
```

If you compile and execute the above program, you will get the following result:

Output

This is the getnum method of the inner class:175

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

```
public class Outerclass {
    // instance method of the outer class
    void my_Method() {
        int num = 23;

        // method-local inner class
        class MethodInner_Demo {
            public void print() {
                System.out.println("This is method inner class "+num);
            }
        } // end of inner class

        // Accessing the inner class
        MethodInner_Demo inner = new MethodInner_Demo();
        inner.print();
    }

    public static void main(String args[]) {
        Outerclass outer = new Outerclass();
        outer.my_Method();
    }
}
```

If you compile and execute the above program, you will get the following result:

Output

```
This is method inner class 23
```

Anonymous Inner Class

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows –

Syntax

```
AnonymousInner an_inner = new AnonymousInner() {  
    public void my_method() {  
        .....  
        .....  
    }  
};
```

The following program shows how to override the method of a class using anonymous inner class.

Example

```
abstract class AnonymousInner {  
    public abstract void mymethod();  
}  
  
public class Outer_class {  
    public static void main(String args[]) {  
        AnonymousInner inner = new AnonymousInner() {  
            public void mymethod() {  
                System.out.println("This is an example of anonymous inner class");  
            }  
        };  
        inner.mymethod();  
    }  
}
```

If you compile and execute the above program, you will get the following result:

Output

```
This is an example of anonymous inner class
```

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument –

```
obj.my_Method(new My_Class() {  
    public void Do() {  
        .....  
        .....  
    }  
});
```

The following program shows how to pass an anonymous inner class as a method argument.

```
// interface  
interface Message {  
    String greet();  
}  
  
public class My_class {  
    // method which accepts the object of interface Message  
    public void displayMessage(Message m) {  
        System.out.println(m.greet() +  
            ", This is an example of anonymous inner class as an argument");  
    }  
    public static void main(String args[]) {  
        // Instantiating the class  
        My_class obj = new My_class();  
  
        // Passing an anonymous inner class as an argument  
        obj.displayMessage(new Message() {  
            public String greet() {  
                return "Hello";  
            }  
        });  
    }  
}
```

If you compile and execute the above program, it gives you the following result:

Output

Hello, This is an example of anonymous inner class as an argument

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows –

Syntax

```
class MyOuter {  
    static class Nested_Demo {  
    }  
}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

```
public class Outer {  
    static class Nested_Demo {  
        public void my_method() {  
            System.out.println("This is my nested class");  
        }  
    }  
  
    public static void main(String args[]) {  
        Outer.Nested_Demo nested = new Outer.Nested_Demo();  
        nested.my_method();  
    }  
}
```

If you compile and execute the above program, you will get the following result:

Output

This is my nested class

Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
    ....
    ....
}
class Sub extends Super {
    ....
    ....
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

Copy and paste the following program in a file with name My_Calculation.java

```
class Calculation {
    int z;

    public void addition(int x, int y) {
        z = x + y;
        System.out.println("The sum of the given numbers:"+z);
    }

    public void Subtraction(int x, int y) {
        z = x - y;
        System.out.println("The difference between the given numbers:"+z);
    }
}

public class My_Calculation extends Calculation {
    public void multiplication(int x, int y) {
        z = x * y;
        System.out.println("The product of the given numbers:"+z);
    }
}
```

```

public static void main(String args[]) {
    int a = 20, b = 10;
    My_Calculation demo = new My_Calculation();
    demo.addition(a, b);
    demo.Subtraction(a, b);
    demo.multiplication(a, b);
}
}

```

Compile and execute the above code as shown below.

```

javac My_Calculation.java
java My_Calculation

```

After executing the program, it will produce the following result:

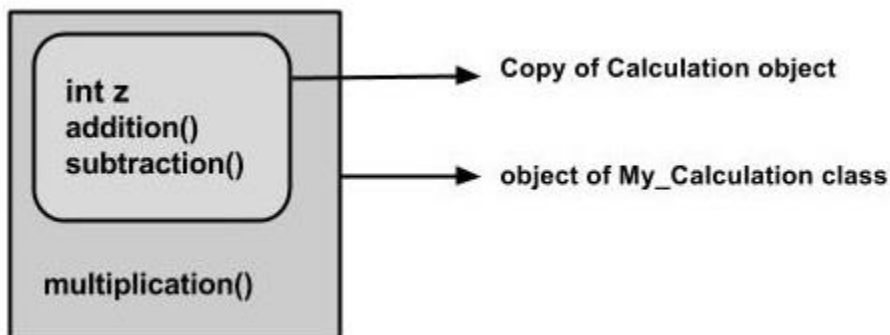
Output

```

The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200

```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass **My_Calculation**.

```

Calculation demo = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);

```

Note: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable  
super.method();
```

Sample Code

This section provides you a program that demonstrates the usage of the **super** keyword.

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named `display()` with different implementations, and a variable named `num` with different values. We are invoking `display()` method of both classes and printing the value of the variable `num` of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name `Sub_class.java`.

```
class Super_class {  
    int num = 20;  
  
    // display method of superclass  
    public void display() {  
        System.out.println("This is the display method of superclass");  
    }  
}  
  
public class Sub_class extends Super_class {  
    int num = 10;  
  
    // display method of sub class  
    public void display() {  
        System.out.println("This is the display method of subclass");  
    }  
    public void my_method() {  
        // Instantiating subclass  
        Sub_class sub = new Sub_class();  
  
        // Invoking the display() method of sub class  
        sub.display();  
  
        // Invoking the display() method of superclass  
        super.display();  
  
        // printing the value of variable num of subclass  
        System.out.println("value of the variable named num in sub class:" + sub.num);  
    }  
}
```

```

        // printing the value of variable num of superclass
        System.out.println("value of the variable named num in super class:"+
            super.num);
    }

    public static void main(String args[]) {
        Sub_class obj = new Sub_class();
        obj.my_method();
    }
}

```

Compile and execute the above code using the following syntax.

```
javac Super_Demo
```

```
java Super
```

On executing the program, you will get the following result:

Output

```

This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20

```

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the Parameterized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

```

class Superclass {
    int age;
    Superclass(int age) {
        this.age = age;
    }
    public void getAge() {
        System.out.println("The value of the variable named age in super class is:" +
            age);
    }
}

public class Subclass extends Superclass {
    Subclass(int age) {
        super(age);
    }
}

```

```
public static void main(String args[]) {  
    Subclass s = new Subclass(24);  
    s.getAge();  
}  
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass  
java Subclass
```

On executing the program, you will get the following result:

Output

The value of the variable named age in super class is:24

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal {  
}  
  
public class Mammal extends Animal {  
}  
  
public class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
}
```

Now, based on the above example, in Object-Oriented terms, the following are true –

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say –

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example

```
class Animal {  
}  
  
class Mammal extends Animal {  
}  
  
class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
    public static void main(String args[]) {  
        Animal a = new Animal();  
        Mammal m = new Mammal();  
        Dog d = new Dog();  
  
        System.out.println(m instanceof Animal);  
        System.out.println(d instanceof Mammal);  
        System.out.println(d instanceof Animal);  
    }  
}
```

This will produce the following result:

Output

```
true  
true  
true
```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {  
}  
  
public class Mammal implements Animal {  
}  
  
public class Dog extends Mammal {  
}
```

The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

```
interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {
    public static void main(String args[]) {
        Mammal m = new Mammal();
        Dog d = new Dog();

        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}
```

This will produce the following result:

Output

```
true
true
true
```

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets look into an example –

Example

```
public class Vehicle{}
public class Speed{}

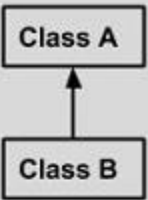
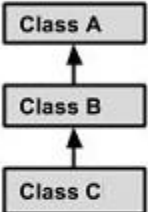
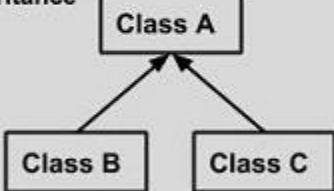
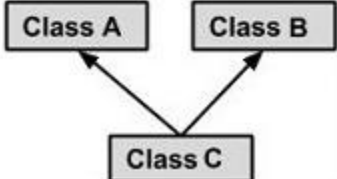
public class Van extends Vehicle {
    private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

Types of Inheritance

There are various types of inheritance as demonstrated below.

Single Inheritance  <pre>graph BT; B[Class B] --> A[Class A]</pre>	<pre>public class A { } public class B extends A { }</pre>
Multi Level Inheritance  <pre>graph BT; C[Class C] --> B[Class B]; B --> A[Class A]</pre>	<pre>public class A {} public class B extends A {.....} public class C extends B {.....}</pre>
Hierarchical Inheritance  <pre>graph BT; B[Class B] --> A[Class A]; C[Class C] --> A</pre>	<pre>public class A {} public class B extends A {.....} public class C extends A {.....}</pre>
Multiple Inheritance  <pre>graph BT; C[Class C] --> A[Class A]; C --> B[Class B]</pre>	<pre>public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance</pre>

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal –

```
public class extends Animal, Mammal{}
```

However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance.

Overriding

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

Example

Let us look at an example.

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {
    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and object
        Animal b = new Dog();       // Animal reference but Dog object

        a.move();    // runs the method in Animal class
        b.move();    // runs the method in Dog class
    }
}
```

This will produce the following result:

Output

```
Animals can move
Dogs can walk and run
```

In the above example, you can see that even though **b** is a type of **Animal** it runs the **move** method in the **Dog** class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since **Animal** class has the method **move**. Then, at the runtime, it runs the method specific for that object.

Consider the following example –

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
    public void bark() {
        System.out.println("Dogs can bark");
    }
}

public class TestDog {
    public static void main(String args[]) {
        Animal a = new Animal();    // Animal reference and object
        Animal b = new Dog();        // Animal reference but Dog object

        a.move();    // runs the method in Animal class
        b.move();    // runs the method in Dog class
        b.bark();
    }
}
```

This will produce the following result:

Output

```
TestDog.java:26:error:cannot find symbol
    b.bark();
      ^
symbol:   method bark()
location: variable b of type Animal
1 error
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.

- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Using the super Keyword

When invoking a superclass version of an overridden method the **super** keyword is used.

Example

```
class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

class Dog extends Animal {
    public void move() {
        super.move(); // invokes the super class method
        System.out.println("Dogs can walk and run");
    }
}

public class TestDog {
    public static void main(String args[]) {
        Animal b = new Dog(); // Animal reference but Dog object
        b.move(); // runs the method in Dog class
    }
}
```

This will produce the following result:

Output

```
Animals can move
Dogs can walk and run
```

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian{}  
public class Animal{}  
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples –

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal –

```
Deer d = new Deer();  
Animal a = d;  
Vegetarian v = d;  
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

Example

```
/* File name:Employee.java */
public class Employee {
    private String name;
    private String address;
    private int number;

    public Employee(String name, String address, int number) {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }

    public void mailCheck() {
        System.out.println("Mailing a check to " + this.name + " " + this.address);
    }

    public String toString() {
        return name + " " + address + " " + number;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String newAddress) {
        address = newAddress;
    }

    public int getNumber() {
        return number;
    }
}
```

Now suppose we extend Employee class as follows –

```
/* File name:Salary.java */
public class Salary extends Employee {
    private double salary; // Annual salary

    public Salary(String name, String address, int number, double salary) {
        super(name, address, number);
        setSalary(salary);
    }

    public void mailCheck() {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
            + " with salary " + salary);
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double newSalary) {
        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }

    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}
```

Now, you study the following program carefully and try to determine its output –

```
/* File name:VirtualDemo.java */
public class VirtualDemo {
    public static void main(String [] args) {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}
```

This will produce the following result:

Output

Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0

Here, we instantiate two Salary objects. One using a Salary reference **s**, and the other using an Employee reference **e**.

While invoking *s.mailCheck()*, the compiler sees *mailCheck()* in the Salary class at compile time, and the JVM invokes *mailCheck()* in the Salary class at run time.

mailCheck() on **e** is quite different because **e** is an Employee reference. When the compiler sees *e.mailCheck()*, the compiler sees the *mailCheck()* method in the Employee class. Here, at compile time, the compiler used *mailCheck()* in Employee to validate this statement. At run time, however, the JVM invokes *mailCheck()* in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

Abstraction

As per dictionary, **abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (`public void get();`)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example

This section provides you an example of the abstract class. To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

```
/* File name:Employee.java */
public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public Employee(String name, String address, int number) {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }

    public double computePay() {
        System.out.println("Inside Employee computePay");
        return 0.0;
    }

    public void mailCheck() {
```

```

        System.out.println("Mailing a check to " + this.name + " " + this.address);
    }

    public String toString() {
        return name + " " + address + " " + number;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String newAddress) {
        address = newAddress;
    }

    public int getNumber() {
        return number;
    }
}

```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way –

```

/* File name:AbstractDemo.java */
public class AbstractDemo {
    public static void main(String [] args) {
        /* Following is not allowed and would raise error */
        Employee e = new Employee("George W.", "Houston, TX", 43);
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

```

When you compile the above class, it gives you the following error –

```

Employee.java:46:Employee is abstract; cannot be instantiated
        Employee e = new Employee("George W.", "Houston, TX", 43);
                        ^
1 error

```

Inheriting the Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way-

```
/* File name:Salary.java */
public class Salary extends Employee {
    private double salary;    // Annual salary

    public Salary(String name, String address, int number, double salary) {
        super(name, address, number);
        setSalary(salary);
    }

    public void mailCheck() {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName() + " with salary " + salary);
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double newSalary) {
        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }

    public double computePay() {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```
/* File name:AbstractDemo.java */
public class AbstractDemo {
    public static void main(String [] args) {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();
        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}
```

This produces the following result:

Output

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semicolon (;) at the end.

Following is an example of the abstract method.

```
public abstract class Employee {  
    private String name;  
    private String address;  
    private int number;  
  
    public abstract double computePay();  
    // Remainder of class definition  
}
```

Declaring a method as abstract has two consequences –

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note: Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below –

```
/* File name:Salary.java */  
public class Salary extends Employee {  
    private double salary;    // Annual salary  
    public double computePay() {  
        System.out.println("Computing salary pay for " + getName());  
        return salary/52;  
    }  
    // Remainder of class definition  
}
```

Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Following is an example that demonstrates how to achieve Encapsulation in Java –

```
/* File name:EncapTest.java */
public class EncapTest {
    private String name;
    private String idNum;
    private int age;
    public int getAge() {
        return age;
    }
    public String getName() {
        return name;
    }
    public String getIdNum() {
        return idNum;
    }
    public void setAge( int newAge) {
        age = newAge;
    }
    public void setName(String newName) {
        name = newName;
    }
    public void setIdNum( String newId) {
        idNum = newId;
    }
}
```

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program –

```
/* File name:RunEncap.java */
public class RunEncap {
    public static void main(String args[]) {
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name:" + encap.getName() + " Age:" + encap.getAge());
    }
}
```

This will produce the following result:

Output

Name:James Age:20

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways –

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including –

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface –

Example

Following is an example of an interface –

```
/* File name:NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
    // Any number of final, static fields
    // Any number of abstract method declarations\
}
```

Interfaces have the following properties –

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name:Animal.java */  
interface Animal {  
    public void eat();  
    public void travel();  
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name:MammalInt.java */  
public class MammalInt implements Animal {  
    public void eat() {  
        System.out.println("Mammal eats");  
    }  
  
    public void travel() {  
        System.out.println("Mammal travels");  
    }  
  
    public int noOfLegs() {  
        return 0;  
    }  
  
    public static void main(String args[]) {  
        MammalInt m = new MammalInt();  
        m.eat();  
        m.travel();  
    }  
}
```

This will produce the following result:

Output

```
Mammal eats  
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed –

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementing interfaces, there are several rules –

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

```
// Filename:Sports.java
public interface Sports {
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}

// Filename:Football.java
public interface Football extends Sports {
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}

// Filename:Hockey.java
public interface Hockey extends Sports {
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
    public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The `extends` keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the `Hockey` interface extended both `Sports` and `Event`, it would be declared as –

Example

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the `MouseListener` interface in the `java.awt.event` package extended `java.util.EventListener`, which is defined as –

Example

```
package java.util;  
public interface EventListener  
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces –

- **Creates a common parent** – As with the `EventListener` interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends `EventListener`, the JVM knows that this particular interface is going to be used in an event delegation scenario.
- **Adds a data type to a class** – This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

Packages

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are –

- **java.lang** – bundles the fundamental classes
- **java.io** – classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* –

```
/* File name:Animal.java */
package animals;

interface Animal {
    public void eat();
    public void travel();
}
```

Now, let us implement the above interface in the same package *animals* –

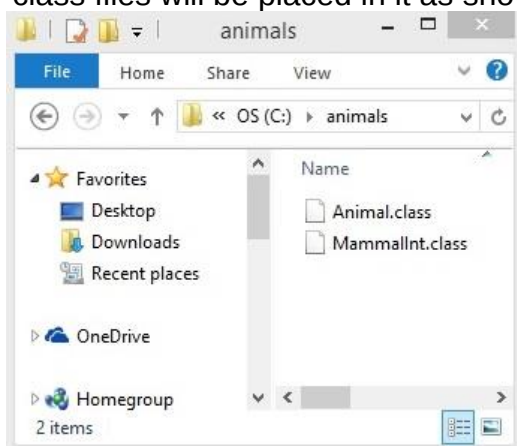
```
package animals;
/* File name:MammalInt.java */

public class MammalInt implements Animal {
    public void eat() {
        System.out.println("Mammal eats");
    }
    public void travel() {
        System.out.println("Mammal travels");
    }
    public int noOfLegs() {
        return 0;
    }
    public static void main(String args[]) {
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}
```

Now compile the java files as shown below –

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.



You can execute the class file within the package and get the result as shown below.

```
Mammal eats
Mammal travels
```

The import Keyword

If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.

Example

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;
public class Boss {
    public void payEmployee(Employee e) {
        e.mailCheck();
    }
}
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example –

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example –

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example –

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages

Two major results occur when a class is placed in a package –

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java –

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**.

For example –

```
// File Name: Car.java
package vehicle;

public class Car {
    // Class implementation.
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs –

```
....\vehicle\Car.java
```

Now, the qualified class name and pathname would be as follows –

- Class name → vehicle.Car
- Path name → vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names.

Example – A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example – The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this –

```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**.

For example –

```
// File Name:Dell.java
package com.apple.computers;

public class Dell {
}

class Ups {
}
```

Now, compile this file as follows using -d option –

```
$javac -d . Dell.java
```

The files will be compiled as follows –

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in `\com\apple\computers\` as follows –

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as –

```
<path-one>\sources\com\apple\computers\Dell.java
```

```
<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give access to the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\computers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell) –

- In Windows → C:\> set CLASSPATH
- In UNIX → % echo \$CLASSPATH

To delete the current contents of the CLASSPATH variable, use –

- In Windows → C:\> set CLASSPATH =
- In UNIX → % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable –

- In Windows → set CLASSPATH = C:\users\jack\java\classes
- In UNIX → % CLASSPATH = /home/jack/java/classes; export CLASSPATH

Data Structures

The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes –

- Enumeration
- BitSet
- Vector
- Stack
- Dictionary
- Hashtable
- Properties

All these classes are now legacy and Java-2 has introduced a new framework called Collections Framework, which is discussed in the next chapter. –

The Enumeration

The Enumeration interface isn't itself a data structure, but it is very important within the context of other data structures. The Enumeration interface defines a means to retrieve successive elements from a data structure.

For example, Enumeration defines a method called `nextElement` that is used to get the next element in a data structure that contains multiple elements.

The BitSet

The `BitSet` class implements a group of bits or flags that can be set and cleared individually. This class is very useful in cases where you need to keep up with a set of Boolean values; you just assign a bit to each value and set or clear it as appropriate.

The Vector

The `Vector` class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.

Like an array, elements of a `Vector` object can be accessed via an index into the vector. The nice thing about using the `Vector` class is that you don't have to worry about setting it to a specific size upon creation; it shrinks and grows automatically when necessary.

The Stack

The `Stack` class implements a last-in-first-out (LIFO) stack of elements. You can think of a stack literally as a vertical stack of objects; when you add a new element, it gets stacked on top of the others.

When you pull an element off the stack, it comes off the top. In other words, the last element you added to the stack is the first one to come back off.

The Dictionary

The Dictionary class is an abstract class that defines a data structure for mapping keys to values.

This is useful in cases where you want to be able to access data via a particular key rather than an integer index.

Since the Dictionary class is abstract, it provides only the framework for a key-mapped data structure rather than a specific implementation.

The Hashtable

The Hashtable class provides a means of organizing data based on some user-defined key structure.

For example, in an address list hash table you could store and sort data based on a key such as ZIP code rather than on a person's name.

The specific meaning of keys with regard to hash tables is totally dependent on the usage of the hash table and the data it contains.

The Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by `System.getProperties()` when obtaining environmental values.

Collections Framework

Prior to Java 2, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used **Vector** was different from the way that you used **Properties**.

The collections framework was designed to meet several goals, such as –

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.

Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList**, **HashSet**, and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following –

- **Interfaces** – These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations, i.e., Classes** – These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms** – These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

The Collection Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface –

No.	Interface & Description
1	<u>The Collection Interface</u> This enables you to work with groups of objects; it is at the top of the collections hierarchy.
2	<u>The List Interface</u> This extends Collection and an instance of List stores an ordered collection of elements.
3	<u>The Set</u> This extends Collection to handle sets, which must contain unique elements.
4	<u>The SortedSet</u> This extends Set to handle sorted sets.
5	<u>The Map</u> This maps unique keys to values.
6	<u>The Map.Entry</u> This describes an element (a key/value pair) in a map. This is an inner class of Map.
7	<u>The SortedMap</u> This extends Map so that the keys are maintained in an ascending order.
8	<u>The Enumeration</u> This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table –

No.	Class & Description
1	AbstractCollection Implements most of the Collection interface.
2	AbstractList Extends AbstractCollection and implements most of the List interface.
3	AbstractSequentialList Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
4	<u>LinkedList</u> Implements a linked list by extending AbstractSequentialList.
5	<u>ArrayList</u> Implements a dynamic array by extending AbstractList.
6	AbstractSet Extends AbstractCollection and implements most of the Set interface.
7	<u>HashSet</u> Extends AbstractSet for use with a hash table.
8	<u>LinkedHashSet</u> Extends HashSet to allow insertion-order iterations.
9	<u>TreeSet</u> Implements a set stored in a tree. Extends AbstractSet.
10	AbstractMap Implements most of the Map interface.
11	<u>HashMap</u> Extends AbstractMap to use a hash table.
12	<u>TreeMap</u> Extends AbstractMap to use a tree.
13	<u>WeakHashMap</u> Extends AbstractMap to use a hash table with weak keys.
14	<u>LinkedHashMap</u> Extends HashMap to allow insertion-order iterations.
15	<u>IdentityHashMap</u>

	Extends AbstractMap and uses reference equality when comparing documents.
--	---

The *AbstractCollection*, *AbstractSet*, *AbstractList*, *AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

The following legacy classes defined by java.util have been discussed in the previous chapter –

No.	Class & Description
1	<u>Vector</u> This implements a dynamic array. It is similar to ArrayList, but with some differences.
2	<u>Stack</u> Stack is a subclass of Vector that implements a standard last-in, first-out stack.
3	<u>Dictionary</u> Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
4	<u>Hashtable</u> Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
5	<u>Properties</u> Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
6	<u>BitSet</u> A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

The Collection Algorithms

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections define three static variables: EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. All are immutable.

No.	Algorithm & Description
1	<u>The Collection Algorithms</u> Here is a list of all the algorithm implementation.

How to Use an Iterator?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

No.	Iterator Method & Description
1	<u>Using Java Iterator</u> Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces.

How to Use a Comparator?

Both TreeSet and TreeMap store elements in a sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Besides, this interface can be used to sort any instances of any class (even classes we cannot modify).

No.	Iterator Method & Description
1	<u>Using Java Comparator</u> Here is a list of all the methods with examples provided by Comparator Interface.

Summary

The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.

A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.

The classes and interfaces of the collections framework are in package java.util.

Generics

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods –

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

Following example illustrates how we can print an array of different type using a single Generic method –

```
public class GenericMethodTest {
    // generic method printArray
    public static < E > void printArray( E[] inputArray ) {
        // Display array elements
        for(E element:inputArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }

    public static void main(String args[]) {
        // Create arrays of Integer, Double and Character
```

```

Integer[] intArray = { 1, 2, 3, 4, 5 };
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println("Array integerArray contains:");
printArray(intArray);    // pass an Integer array

System.out.println("\nArray doubleArray contains:");
printArray(doubleArray); // pass a Double array

System.out.println("\nArray characterArray contains:");
printArray(charArray);   // pass a Character array
}
}

```

This will produce the following result:

Output

```

Array integerArray contains:
1 2 3 4 5

```

```

Array doubleArray contains:
1.1 2.2 3.3 4.4

```

```

Array characterArray contains:
H E L L O

```

Bounded Type Parameters

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its upper bound.

Example

Following example illustrates how `extends` is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three `Comparable` objects –

```

public class MaximumTest {
    // determines the largest of three Comparable objects
    public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
        T max = x;    // assume x is initially the largest

        if (y.compareTo(max) > 0) {
            max = y;    // y is the largest so far
        }
        if (z.compareTo(max) > 0) {
            max = z;    // z is the largest now
        }
        return max;    // returns the largest object
    }
}

```

```

public static void main(String args[]) {
    System.out.printf("Max of %d, %d and %d is %d\n\n",
        3, 4, 5, maximum( 3, 4, 5 ));

    System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
        6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

    System.out.printf("Max of %s, %s and %s is %s\n", "pear",
        "apple", "orange", maximum("pear", "apple", "orange"));
}
}

```

This will produce the following result:

Output

Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear

Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example

Following example illustrates how we can define a generic class –

```

public class Box<T> {
    private T t;
    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        Box<String> stringBox = new Box<String>();
        integerBox.add(new Integer(10));
        stringBox.add(new String("Hello World"));
        System.out.printf("Integer Value:%d\n\n", integerBox.get());
        System.out.printf("String Value:%s\n", stringBox.get());
    }
}

```

This will produce the following result:

Output

Integer Value:10

String Value:Hello World

Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out –

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object –

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface –

```
public class Employee implements java.io.Serializable {
    public String name;
    public String address;
    public transient int SSN;
    public int number;

    public void mailCheck() {
        System.out.println("Mailing a check to " + name + " " + address);
    }
}
```

Notice that for a class to be serialized successfully, two conditions must be met –

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object

The `ObjectOutputStream` class is used to serialize an Object. The following `SerializeDemo` program instantiates an `Employee` object and serializes it to a file.

When the program is done executing, a file named `employee.ser` is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a `.ser` extension.

Example

```
import java.io.*;
public class SerializeDemo {
    public static void main(String [] args) {
        Employee e = new Employee();
        e.name = "Reyan Ali";
        e.address = "Phokka Kuan, Ambehta Peer";
        e.SSN = 11122333;
        e.number = 101;

        try {
            FileOutputStream fileOut =
                new FileOutputStream("/tmp/employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(e);
            out.close();
            fileOut.close();
            System.out.printf("Serialized data is saved in /tmp/employee.ser");
        } catch (IOException i) {
            i.printStackTrace();
        }
    }
}
```

Deserializing an Object

The following `DeserializeDemo` program deserializes the `Employee` object created in the `SerializeDemo` program. Study the program and try to determine its output –

```
import java.io.*;
public class DeserializeDemo {
    public static void main(String [] args) {
        Employee e = null;
        try {
            FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            e = (Employee) in.readObject();
            in.close();
            fileIn.close();
        } catch (IOException i) {
            i.printStackTrace();
        }
        return;
    } catch (ClassNotFoundException c) {
```

```
        System.out.println("Employee class not found");
        c.printStackTrace();
        return;
    }

    System.out.println("Deserialized Employee...");
    System.out.println("Name:" + e.name);
    System.out.println("Address:" + e.address);
    System.out.println("SSN:" + e.SSN);
    System.out.println("Number:" + e.number);
}
}
```

This will produce the following result:

Output

```
Deserialized Employee...
Name:Reyan Ali
Address:Phokka Kuan, Ambhehta Peer
SSN:0
Number:101
```

Here are following important points to be noted –

- The try/catch block tries to catch a `ClassNotFoundException`, which is declared by the `readObject()` method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a `ClassNotFoundException`.
- Notice that the return value of `readObject()` is cast to an `Employee` reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized `Employee` object is 0.

Multithreading

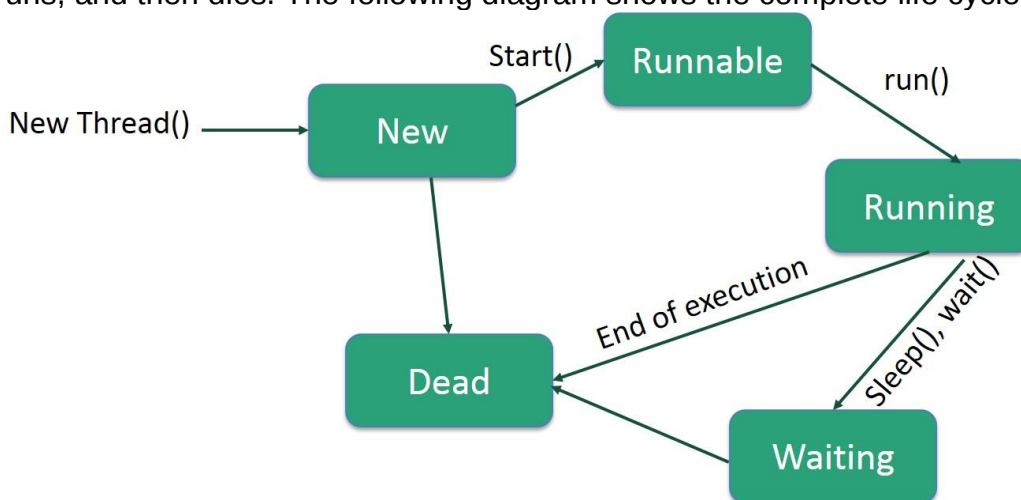
Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle –

- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10). By default, every thread is given priority `NORM_PRIORITY` (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic steps –

1. As a first step, you need to implement a `run()` method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the `run()` method –
`public void run( )`
2. As a second step, you will instantiate a **Thread** object using the following constructor –
`Thread(Runnable threadObj, String threadName);`

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

3. Once a Thread object is created, you can start it by calling **start()** method, which executes a call to `run( )` method. Following is a simple syntax of `start()` method –
`void start();`

Example

Here is an example that creates a new thread and starts running it –

```
class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;

    RunnableDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }

    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread:" + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
    }
}
```

```

        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {
    public static void main(String args[]) {
        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}

```

This will produce the following result:

Output

```

Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread:Thread-1, 4
Running Thread-2
Thread:Thread-2, 4
Thread:Thread-1, 3
Thread:Thread-2, 3
Thread:Thread-1, 2
Thread:Thread-2, 2
Thread:Thread-1, 1
Thread:Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.

```

Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run()** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method.

Following is a simple syntax of run() method –

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to **run()** method. Following is a simple syntax of **start()** method –

```
void start( );
```

Example

Here is the preceding program rewritten to extend the Thread –

```
class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;

    ThreadDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }

    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread:" + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {
    public static void main(String args[]) {
        ThreadDemo T1 = new ThreadDemo( "Thread-1");
        T1.start();

        ThreadDemo T2 = new ThreadDemo( "Thread-2");
        T2.start();
    }
}
```

This will produce the following result:

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
```

```

Running Thread-1
Thread:Thread-1, 4
Running Thread-2
Thread:Thread-2, 4
Thread:Thread-1, 3
Thread:Thread-2, 3
Thread:Thread-1, 2
Thread:Thread-2, 2
Thread:Thread-1, 1
Thread:Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.

```

Thread Methods

Following is the list of important methods available in the Thread class.

No.	Method & Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

No.	Method & Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable** –

```
// File Name: DisplayMessage.java
// Create a thread to implement Runnable

public class DisplayMessage implements Runnable {
    private String message;

    public DisplayMessage(String message) {
        this.message = message;
    }

    public void run() {
        while(true) {
            System.out.println(message);
        }
    }
}
```

Following is another **class** which **extends** the Thread **class** ?

```
// File Name:GuessANumber.java
// Create a thread to extend Thread

public class GuessANumber extends Thread {
    private int number;
    public GuessANumber(int number) {
        this.number = number;
    }

    public void run() {
        int counter = 0;
        int guess = 0;
        do {
            guess = (int) (Math.random() * 100 + 1);
            System.out.println(this.getName() + " guesses " + guess);
            counter++;
        } while(guess != number);
        System.out.println("** Correct!" + this.getName() + " in " + counter +
"guesses.**");
    }
}
```

Following is the main program, which makes use of the above-defined classes –

```
// File Name:ThreadClassDemo.java
public class ThreadClassDemo {
    public static void main(String [] args) {
        Runnable hello = new DisplayMessage("Hello");
        Thread thread1 = new Thread(hello);
        thread1.setDaemon(true);
        thread1.setName("hello");
        System.out.println("Starting hello thread...");
        thread1.start();

        Runnable bye = new DisplayMessage("Goodbye");
        Thread thread2 = new Thread(bye);
        thread2.setPriority(Thread.MIN_PRIORITY);
        thread2.setDaemon(true);
        System.out.println("Starting goodbye thread...");
        thread2.start();

        System.out.println("Starting thread3...");
        Thread thread3 = new GuessANumber(27);
        thread3.start();
        try {
            thread3.join();
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted.");
        }
        System.out.println("Starting thread4...");
        Thread thread4 = new GuessANumber(75);

        thread4.start();
        System.out.println("main() is ending...");
    }
}
```

This will produce the following result. You can try this example again and again and you will get a different result every time.

Output

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.....
```

Major Java Multithreading Concepts

While doing Multithreading programming in Java, you would need to have the following concepts very handy –

- What is thread synchronization?
- Handling interthread communication
- Handling thread deadlock
- Major thread operations

Networking

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols –

- **TCP** – TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP** – UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects –

- **Socket Programming** – This is the most widely used concept in Networking and it has been explained in very detail.
- **URL Processing** – This would be covered separately. Click here to learn about [URL Processing](#) in Java language.

Socket Programming

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them. The following steps occur when establishing a TCP connection between two computers using sockets –

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.

ServerSocket Class Methods

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests.

The ServerSocket class has four constructors –

No.	Method & Description
1	public ServerSocket(int port) throws IOException Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	public ServerSocket(int port, int backlog) throws IOException Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.
3	public ServerSocket(int port, int backlog, InetAddress address) throws IOException Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on.
4	public ServerSocket() throws IOException Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket.

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Following are some of the common methods of the ServerSocket class –

No.	Method & Description
1	public int getLocalPort() Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2	public Socket accept() throws IOException

	Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the <code>setSoTimeout()</code> method. Otherwise, this method blocks indefinitely.
3	public void setSoTimeout(int timeout) Sets the time-out value for how long the server socket waits for a client during the <code>accept()</code> .
4	public void bind(SocketAddress host, int backlog) Binds the socket to the specified server and port in the <code>SocketAddress</code> object. Use this method if you have instantiated the <code>ServerSocket</code> using the no-argument constructor.

When the `ServerSocket` invokes `accept()`, the method does not return until a client connects. After a client does connect, the `ServerSocket` creates a new `Socket` on an unspecified port and returns a reference to this new `Socket`. A TCP connection now exists between the client and the server, and communication can begin.

Socket Class Methods

The **java.net.Socket** class represents the socket that both the client and the server use to communicate with each other. The client obtains a `Socket` object by instantiating one, whereas the server obtains a `Socket` object from the return value of the `accept()` method.

The `Socket` class has five constructors that a client uses to connect to a server –

No.	Method & Description
1	public Socket(String host, int port) throws UnknownHostException, IOException. This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2	public Socket(InetAddress host, int port) throws IOException This method is identical to the previous constructor, except that the host is denoted by an <code>InetAddress</code> object.
3	public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException. Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.

	This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String.
5	public Socket() Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and the server have a Socket object, so these methods can be invoked by both the client and the server.

No.	Method & Description
1	public void connect(SocketAddress host, int timeout) throws IOException This method connects the socket to the specified host. This method is needed only when you instantiate the Socket using the no-argument constructor.
2	public InetAddress getInetAddress() This method returns the address of the other computer that this socket is connected to.
3	public int getPort() Returns the port the socket is bound to on the remote machine.
4	public int getLocalPort() Returns the port the socket is bound to on the local machine.
5	public SocketAddress getRemoteSocketAddress() Returns the address of the remote socket.
6	public InputStream getInputStream() throws IOException Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7	public OutputStream getOutputStream() throws IOException Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket.
8	public void close() throws IOException Closes the socket, which makes this Socket object no longer capable of connecting again to any server.

InetAddress Class Methods

This class represents an Internet Protocol (IP) address. Here are following usefull methods which you would need while doing socket programming –

No.	Method & Description
1	static InetAddress getByAddress(byte[] addr) Returns an InetAddress object given the raw IP address.
2	static InetAddress getByAddress(String host, byte[] addr) Creates an InetAddress based on the provided host name and IP address.
3	static InetAddress getName(String host) Determines the IP address of a host, given the host's name.
4	String getHostAddress() Returns the IP address string in textual presentation.
5	String getHostName() Gets the host name for this IP address.
6	static InetAddress InetAddress getLocalHost() Returns the local host.
7	String toString() Converts this IP address to a String.

Socket Client Example

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;

public class GreetingClient {
    public static void main(String [] args) {
        String serverName = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            System.out.println("Connecting to " + serverName + " on port " + port);
            Socket client = new Socket(serverName, port);

            System.out.println("Just connected to " + client.getRemoteSocketAddress());
            OutputStream outToServer = client.getOutputStream();
            DataOutputStream out = new DataOutputStream(outToServer);

            out.writeUTF("Hello from " + client.getLocalSocketAddress());
            InputStream inFromServer = client.getInputStream();
            DataInputStream in = new DataInputStream(inFromServer);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        System.out.println("Server says " + in.readUTF());
        client.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Socket Server Example

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument –

```

// File Name GreetingServer.java
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
    private ServerSocket serverSocket;
    public GreetingServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(10000);
    }
    public void run() {
        while(true) {
            try {
                System.out.println("Waiting for client on port " +
                    serverSocket.getLocalPort() + "...");
                Socket server = serverSocket.accept();

                System.out.println("Just connected to " + server.getRemoteSocketAddress());
                DataInputStream in = new DataInputStream(server.getInputStream());

                System.out.println(in.readUTF());
                DataOutputStream out = new DataOutputStream(server.getOutputStream());
                out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress()
                    + "\nGoodbye!");
                server.close();
            } catch (SocketTimeoutException s) {
                System.out.println("Socket timed out!");
                break;
            } catch (IOException e) {
                e.printStackTrace();
                break;
            }
        }
    }
    public static void main(String [] args) {
        int port = Integer.parseInt(args[0]);
        try {
            Thread t = new GreetingServer(port);
            t.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Compile the client and the server and then start the server as follows –

```
$ java GreetingServer 6066  
Waiting for client on port 6066...
```

Check the client program as follows –

Output

```
$ java GreetingClient localhost 6066  
Connecting to localhost on port 6066  
Just connected to localhost/127.0.0.1:6066  
Server says Thank you for connecting to /127.0.0.1:6066  
Goodbye!
```

Sending Email

To send an e-mail using your Java Application is simple enough but to start with you should have **JavaMail API** and **Java Activation Framework (JAF)** installed on your machine.

- You can download latest version of JavaMail (Version 1.2) from Java's standard website.
- You can download latest version of JAF (Version 1.1.1) from Java's standard website.

Download and unzip these files, in the newly created top level directories you will find a number of jar files for both the applications. You need to add **mail.jar** and **activation.jar** files in your CLASSPATH.

Send a Simple E-mail

Here is an example to send a simple e-mail from your machine. It is assumed that your **localhost** is connected to the Internet and capable enough to send an e-mail.

```
// File Name SendEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail {
    public static void main(String [] args) {
        // Recipient's email ID needs to be mentioned.
        String to = "abcd@gmail.com";

        // Sender's email ID needs to be mentioned
        String from = "web@gmail.com";

        // Assuming you are sending email from localhost
        String host = "localhost";

        // Get system properties
        Properties properties = System.getProperties();

        // Setup mail server
        properties.setProperty("mail.smtp.host", host);

        // Get the default Session object.
        Session session = Session.getDefaultInstance(properties);

        try {
            // Create a default MimeMessage object.
            MimeMessage message = new MimeMessage(session);

            // Set From:header field of the header.
            message.setFrom(new InternetAddress(from));

            // Set To:header field of the header.
            message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
```

```

// Set Subject:header field
message.setSubject("This is the Subject Line!");

// Now set the actual message
message.setText("This is actual message");

// Send message
Transport.send(message);
System.out.println("Sent message successfully....");
} catch (MessagingException mex) {
    mex.printStackTrace();
}
}
}

```

Compile and run this program to send a simple e-mail –

Output

```

$ java SendEmail
Sent message successfully....

```

If you want to send an e-mail to multiple recipients then the following methods would be used to specify multiple e-mail IDs –

```

void addRecipients(Message.RecipientType type, Address[] addresses)
    throws MessagingException

```

Here is the description of the parameters –

- **type** – This would be set to TO, CC or BCC. Here CC represents Carbon Copy and BCC represents Black Carbon Copy. Example: *Message.RecipientType.TO*
- **addresses** – This is an array of e-mail ID. You would need to use *InternetAddress()* method while specifying email IDs.

Send an HTML E-mail

Here is an example to send an HTML e-mail from your machine. Here it is assumed that your **localhost** is connected to the Internet and capable enough to send an e-mail.

This example is very similar to the previous one, except here we are using *setContent()* method to set content whose second argument is "text/html" to specify that the HTML content is included in the message.

Using this example, you can send as big as HTML content you like.

```

// File Name SendHTMLEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendHTMLEmail {
    public static void main(String [] args) {
        // Recipient's email ID needs to be mentioned.
        String to = "abcd@gmail.com";
    }
}

```

```

// Sender's email ID needs to be mentioned
String from = "web@gmail.com";

// Assuming you are sending email from localhost
String host = "localhost";

// Get system properties
Properties properties = System.getProperties();

// Setup mail server
properties.setProperty("mail.smtp.host", host);

// Get the default Session object.
Session session = Session.getDefaultInstance(properties);

try {
    // Create a default MimeMessage object.
    MimeMessage message = new MimeMessage(session);

    // Set From:header field of the header.
    message.setFrom(new InternetAddress(from));

    // Set To:header field of the header.
    message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

    // Set Subject:header field
    message.setSubject("This is the Subject Line!");

    // Send the actual HTML message, as big as you like
    message.setContent("<h1>This is actual message</h1>", "text/html");

    // Send message
    Transport.send(message);
    System.out.println("Sent message successfully....");
} catch (MessagingException mex) {
    mex.printStackTrace();
}
}
}

```

Compile and run this program to send an HTML e-mail –

Output

```

$ java SendHTMLEmail
Sent message successfully....

```

Send Attachment in E-mail

Here is an example to send an e-mail with attachment from your machine. Here it is assumed that your **localhost** is connected to the internet and capable enough to send an e-mail.

```

// File Name SendFileEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

```

```

public class SendFileEmail {
    public static void main(String [] args) {
        // Recipient's email ID needs to be mentioned.
        String to = "abcd@gmail.com";

        // Sender's email ID needs to be mentioned
        String from = "web@gmail.com";

        // Assuming you are sending email from localhost
        String host = "localhost";

        // Get system properties
        Properties properties = System.getProperties();

        // Setup mail server
        properties.setProperty("mail.smtp.host", host);

        // Get the default Session object.
        Session session = Session.getDefaultInstance(properties);

        try {
            // Create a default MimeMessage object.
            MimeMessage message = new MimeMessage(session);

            // Set From:header field of the header.
            message.setFrom(new InternetAddress(from));

            // Set To:header field of the header.
            message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));

            // Set Subject:header field
            message.setSubject("This is the Subject Line!");

            // Create the message part
            BodyPart messageBodyPart = new MimeBodyPart();

            // Fill the message
            messageBodyPart.setText("This is message body");

            // Create a multipar message
            Multipart multipart = new MimeMultipart();

            // Set text message part
            multipart.addBodyPart(messageBodyPart);

            // Part two is attachment
            messageBodyPart = new MimeBodyPart();
            String filename = "file.txt";
            DataSource source = new FileDataSource(filename);
            messageBodyPart.setDataHandler(new DataHandler(source));
            messageBodyPart.setFileName(filename);
            multipart.addBodyPart(messageBodyPart);

            // Send the complete message parts
            message.setContent(multipart );

            // Send message

```

```
        Transport.send(message);  
        System.out.println("Sent message successfully....");  
    } catch (MessagingException mex) {  
        mex.printStackTrace();  
    }  
}  
}
```

Compile and run this program to send an HTML e-mail –

Output

```
$ java SendFileEmail  
Sent message successfully....
```

User Authentication Part

If it is required to provide user ID and Password to the e-mail server for authentication purpose, then you can set these properties as follows –

```
props.setProperty("mail.user", "myuser");  
props.setProperty("mail.password", "mypwd");
```

Rest of the e-mail sending mechanism would remain as explained above.

Applet Basics

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.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown –

```
<applet codebase = "https://amrood.com/applets" code = "HelloWorldApplet.class"
  width = "320" height = "120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example –

```
<applet code = "mypackage.subpackage.TestApplet.class"
  width = "320" height = "120">
```

Getting Applet Parameters

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color.

The second color and the size of each square may be specified as parameters to the applet within the document.

CheckerApplet gets its parameters in the `init()` method. It may also get its parameters in the `paint()` method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.

The applet viewer or browser calls the `init()` method of each applet it runs. The viewer calls `init()` once, immediately after loading the applet. (`Applet.init()` is implemented to do nothing.) Override the default implementation to insert custom initialization code.

The `Applet.getParameter()` method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.

The following is a skeleton of `CheckerApplet.java` –

```
import java.applet.*;
import java.awt.*;

public class CheckerApplet extends Applet {
    int squareSize = 50;    // initialized to default size
    public void init() {}
    private void parseSquareSize (String param) {}
    private Color parseColor (String param) {}
    public void paint (Graphics g) {}
}
```

Here are `CheckerApplet`'s `init()` and private `parseSquareSize()` methods –

```
public void init () {
    String squareSizeParam = getParameter ("squareSize");
    parseSquareSize (squareSizeParam);

    String colorParam = getParameter ("color");
    Color fg = parseColor (colorParam);

    setBackground (Color.black);
    setForeground (fg);
}

private void parseSquareSize (String param) {
    if (param == null) return;
    try {
        squareSize = Integer.parseInt (param);
    } catch (Exception e) {
        // Let default value remain
    }
}
```

The applet calls `parseSquareSize()` to parse the `squareSize` parameter. `parseSquareSize()` calls the library method `Integer.parseInt()`, which parses a string and returns an integer. `Integer.parseInt()` throws an exception whenever its argument is invalid. Therefore, `parseSquareSize()` catches exceptions, rather than allowing the applet to fail on bad input.

The applet calls `parseColor()` to parse the color parameter into a `Color` value. `parseColor()` does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet work.

Specifying Applet Parameters

The following is an example of an HTML file with a `CheckerApplet` embedded in it. The HTML file specifies both parameters to the applet by means of the `<param>` tag.

```
<html>
  <title>Checkerboard Applet</title>
  <hr>
  <applet code = "CheckerApplet.class" width = "480" height = "320">
    <param name = "color" value = "blue">
    <param name = "squareSize" value = "30">
  </applet>
  <hr>
</html>
```

Note: Parameter names are not case sensitive.

Application Conversion to Applets

It is easy to convert a graphical Java application (that is, an application that uses the AWT and that you can start with the Java program launcher) into an applet that you can embed in a web page.

Following are the specific steps for converting an application to an applet.

- Make an HTML page with the appropriate tag to load the applet code.
- Supply a subclass of the `JApplet` class. Make this class public. Otherwise, the applet cannot be loaded.
- Eliminate the `main` method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.
- Move any initialization code from the frame window constructor to the `init` method of the applet. You don't need to explicitly construct the applet object. The browser instantiates it for you and calls the `init` method.
- Remove the call to `setSize`; for applets, sizing is done with the width and height parameters in the HTML file.
- Remove the call to `setDefaultCloseOperation`. An applet cannot be closed; it terminates when the browser exits.
- If the application calls `setTitle`, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML title tag.)
- Don't call `setVisible(true)`. The applet is displayed automatically.

Event Handling

Applets inherit a group of event-handling methods from the Container class. The Container class defines several methods, such as processKeyEvent and processMouseEvent, for handling particular types of events, and then one catch-all method called processEvent.

In order to react to an event, an applet must override the appropriate event-specific method.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class ExampleEventHandling extends Applet implements MouseListener {
    StringBuffer strBuffer;

    public void init() {
        addMouseListener(this);
        strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }

    public void start() {
        addItem("starting the applet ");
    }

    public void stop() {
        addItem("stopping the applet ");
    }

    public void destroy() {
        addItem("unloading the applet");
    }

    void addItem(String word) {
        System.out.println(word);
        strBuffer.append(word);
        repaint();
    }

    public void paint(Graphics g) {
        // Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
            getWidth() - 1,
            getHeight() - 1);

        // display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20);
    }

    public void mouseEntered(MouseEvent event) { }
    public void mouseExited(MouseEvent event) { }
    public void mousePressed(MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
    public void mouseClicked(MouseEvent event) {
        addItem("mouse clicked! ");
    }
}
```

Now, let us call this applet as follows –

```
<html>
  <title>Event Handling</title>
  <hr>
    <applet code = "ExampleEventHandling.class"
      width = "300" height = "300">
    </applet>
  <hr>
</html>
```

Initially, the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle, "mouse clicked" will be displayed as well.

Displaying Images

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within the applet, you use the `drawImage()` method found in the `java.awt.Graphics` class.

Following is an example illustrating all the steps to show images –

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class ImageDemo extends Applet {
    private Image image;
    private AppletContext context;

    public void init() {
        context = this.getAppletContext();
        String imageURL = this.getParameter("image");
        if(imageURL == null) {
            imageURL = "java.jpg";
        }
        try {
            URL url = new URL(this.getDocumentBase(), imageURL);
            image = context.getImage(url);
        } catch (MalformedURLException e) {
            e.printStackTrace();
            // Display in browser status bar
            context.showStatus("Could not load image!");
        }
    }

    public void paint(Graphics g) {
        context.showStatus("Displaying image");
        g.drawImage(image, 0, 0, 200, 84, null);
        g.drawString("www.javalicense.com", 35, 100);
    }
}
```

Now, let us call this applet as follows –

```
<html>
  <title>The ImageDemo applet</title>
  <hr>
  <applet code = "ImageDemo.class" width = "300" height = "200">
    <param name = "image" value = "java.jpg">
  </applet>
  <hr>
</html>
```

Playing Audio

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including –

- **public void play()** – Plays the audio clip one time, from the beginning.
- **public void loop()** – Causes the audio clip to replay continually.
- **public void stop()** – Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is an example illustrating all the steps to play an audio –

```
import java.applet.*;
import java.awt.*;
import java.net.*;

public class AudioDemo extends Applet {
    private AudioClip clip;
    private AppletContext context;
    public void init() {
        context = this.getAppletContext();
        String audioURL = this.getParameter("audio");
        if(audioURL == null) audioURL = "default.au";
        try {
            URL url = new URL(this.getDocumentBase(), audioURL);
            clip = context.getAudioClip(url);
        } catch (MalformedURLException e) {
            e.printStackTrace();
            context.showStatus("Could not load audio file!");
        }
    }
    public void start() {
        if(clip != null) clip.loop();
    }
    public void stop() {
        if(clip != null) clip.stop();
    }
}
```

Now, let us call this applet as follows –

```
<html>
  <title>The ImageDemo applet</title>
  <hr>
  <applet code = "ImageDemo.class" width = "0" height = "0">
    <param name = "audio" value = "test.wav">
  </applet>
  <hr>
</html>
```

You can use test.wav on your PC to test the above example.

Documentation Comments

The Java language supports three types of comments –

No.	Comment & Description
1	<code>/* text */</code> The compiler ignores everything from <code>/*</code> to <code>*/</code> .
2	<code>//text</code> The compiler ignores everything from <code>//</code> to the end of the line.
3	<code>/** documentation */</code> This is a documentation comment and in general its called doc comment . The JDK javadoc tool uses <i>doc comments</i> when preparing automatically generated documentation.

This chapter is all about explaining Javadoc. We will see how we can make use of Javadoc to generate useful documentation for Java code.

What is Javadoc?

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code, which requires documentation in a predefined format.

Following is a simple example where the lines inside `/*....*/` are Java multi-line comments. Similarly, the line which preceeds `//` is Java single-line comment.

Example

```
/**
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 *
 * @author  Zara Ali
 * @version 1.0
 * @since   2014-03-31
 */
public class HelloWorld {
    public static void main(String[] args) {
        // Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

You can include required HTML tags inside the description part. For instance, the following example makes use of `<h1>....</h1>` for heading and `<p>` has been used for creating paragraph break –

Example

```
/**
 * <h1>Hello, World!</h1>
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 * <p>
 * Giving proper comments in your program makes it more
 * user friendly and it is assumed as a high quality code.
 *
 *
 * @author Ali
 * @version 1.0
 * @since 2014-03-31
 */
public class HelloWorld {
    public static void main(String[] args) {
        // Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

The javadoc Tags

The javadoc tool recognizes the following tags –

Tag	Description	Syntax
@author	Adds the author of a class.	@author name-text
{@code}	Displays text in code font without interpreting the text as HTML markup or nested javadoc tags.	{@code text}
{@docRoot}	Represents the relative path to the generated document's root directory from any generated page.	{@docRoot}
@deprecated	Adds a comment indicating that this API should no longer be used.	@deprecated deprecatedtext
@exception	Adds a Throws subheading to the generated documentation, with the classname and description text.	@exception class-name description
{@inheritDoc}	Inherits a comment from the nearest inheritable class or implementable interface.	Inherits a comment from the immediate superclass.
{@link}	Inserts an in-line link with the visible text label that points to the documentation for the specified package, class, or member name of a referenced class.	{@link package.class#member label}

<code>{@linkplain}</code>	Identical to <code>{@link}</code> , except the link's label is displayed in plain text than code font.	<code>{@linkplain package.class#member label}</code>
<code>@param</code>	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section.	<code>@param parameter- name description</code>
<code>@return</code>	Adds a "Returns" section with the description text.	<code>@return description</code>
<code>@see</code>	Adds a "See Also" heading with a link or text entry that points to reference.	<code>@see reference</code>
<code>@serial</code>	Used in the doc comment for a default serializable field.	<code>@serial field- description include exclude</code>
<code>@serialData</code>	Documents the data written by the <code>writeObject()</code> or <code>writeExternal()</code> methods.	<code>@serialData data- description</code>
<code>@serialField</code>	Documents an <code>ObjectStreamField</code> component.	<code>@serialField field-name field-type field- description</code>
<code>@since</code>	Adds a "Since" heading with the specified since-text to the generated documentation.	<code>@since release</code>
<code>@throws</code>	The <code>@throws</code> and <code>@exception</code> tags are synonyms.	<code>@throws class-name description</code>
<code>{@value}</code>	When <code>{@value}</code> is used in the doc comment of a static field, it displays the value of that constant.	<code>{@value package.class#field}</code>
<code>@version</code>	Adds a "Version" subheading with the specified version-text to the generated docs when the <code>-version</code> option is used.	<code>@version version-text</code>

Example

Following program uses few of the important tags available for documentation comments. You can make use of other tags based on your requirements.

The documentation about the AddNum class will be produced in HTML file AddNum.html but at the same time a master file with a name index.html will also be created.

```
import java.io.*;
/**
 * <h1>Add Two Numbers!</h1>
 * The AddNum program implements an application that
 * simply adds two given integer numbers and Prints
 * the output on the screen.
 * <p>
 * <b>Note:</b> Giving proper comments in your program makes it more
 * user friendly and it is assumed as a high quality code.
 *
 * @author Ali
 * @version 1.0
 * @since 2014-03-31
 */
public class AddNum {
    /**
     * This method is used to add two integers. This is
     * a the simplest form of a class method, just to
     * show the usage of various javadoc Tags.
     * @param numA This is the first paramter to addNum method
     * @param numB This is the second parameter to addNum method
     * @return int This returns sum of numA and numB.
     */
    public int addNum(int numA, int numB) {
        return numA + numB;
    }

    /**
     * This is the main method which makes use of addNum method.
     * @param args Unused.
     * @return Nothing.
     * @exception IOException On input error.
     * @see IOException
     */
    public static void main(String args[]) throws IOException {
        AddNum obj = new AddNum();
        int sum = obj.addNum(10, 20);

        System.out.println("Sum of 10 and 20 is:" + sum);
    }
}
```

Now, process the above AddNum.java file using javadoc utility as follows –

```
$ javadoc AddNum.java
Loading source file AddNum.java...
Constructing Javadoc information...
Standard Doclet version 1.7.0_51
Building tree for all the packages and classes...
```

```
Generating /AddNum.html...
AddNum.java:36:warning - @return tag cannot be used in method with void
return type.
Generating /package-frame.html...
Generating /package-summary.html...
Generating /package-tree.html...
Generating /constant-values.html...
Building index for all the packages and classes...
Generating /overview-tree.html...
Generating /index-all.html...
Generating /deprecated-list.html...
Building index for all classes...
Generating /allclasses-frame.html...
Generating /allclasses-noframe.html...
Generating /index.html...
Generating /help-doc.html...
1 warning
$
```

You can check all the generated documentation here – [AddNum](#). If you are using JDK 1.7 then javadoc does not generate a great **stylesheet.css**, so we suggest to download and use standard stylesheet from <https://docs.oracle.com/javase/7/docs/api/stylesheet.css>