# Java – Basic Syntax

## 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:

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.

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  
 System.*out*.println("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.

# Java – 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 datatypes to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

* Primitive Datatypes
* Reference/Object Datatypes

## Primitive Datatypes

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 datatype 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 datatype 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 datatype 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 datatype 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 datatype 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 datatype 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 datatype is never used for precise values such as currency
* Example: float f1 = 234.5f

**double:**

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

**boolean:**

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

**char:**

* char datatype is a single 16-bit Unicode character
* Minimum value is '\u0000' (or 0)
* Maximum value is '\uffff' (or 65,535 inclusive)
* Char datatype 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. They are:

|  |  |
| --- | --- |
| **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) |

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

This will produce the following result:

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

This will produce the following error while compiling it:

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

This will produce the following result:

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:

Development average salary:1000

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

# Java – 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

## Java Access 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).

## Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

**Example**

Variables and methods can be declared without any modifiers, as in the following examples:

String version = "1.5.1";

boolean processOrder() { return true;

}

**Private Access Modifier - Private**

* Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
* Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
* Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
* Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

**Example**

The following class uses private access control:

public class Logger {  
 private String format;  
  
 public String getFormat() {  
 return this.format;  
 }  
  
 public void setFormat(String format) {  
 this.format = format;  
 }  
}

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

**Public Access Modifier - Public**

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

**Example**

The following function uses public access control:

public static void main(String[] arguments) {

// ...

}

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

**Protected Access Modifier - Protected**

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

**Example**

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method:

class AudioPlayer {  
 protected boolean openSpeaker(Speaker sp) {  
 // implementation details  
 return;  
 }  
}  
  
class StreamingAudioPlayer {  
 boolean openSpeaker(Speaker sp) {  
 // implementation details  
 return;  
 }  
}

Here, if we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that’s why we have used *protected* modifier.

**Access Control and Inheritance**

The following rules for inherited methods are enforced:

* Methods declared public in a superclass also must be public in all subclasses.
* Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
* Methods declared private are not inherited at all, so there is no rule for them.

## Java 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.

## The Static Modifier

**Static Variables**

The *static* keyword is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

**Static Methods**

The *static* keyword is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

**Example**

The static modifier is used to create class methods and variables, as in the following example:

public class InstanceCounter {  
 private static int *numInstances* = 0;  
  
 InstanceCounter() {  
 InstanceCounter.*addInstance*();  
 }  
  
 protected static int getCount() {  
 return *numInstances*;  
 }  
  
 private static void addInstance() {  
 *numInstances*++;  
 }  
  
 public static void main(String[] arguments) {  
 System.*out*.println("Starting with " + InstanceCounter.*getCount*() + " instances");  
 for (int i = 0; i < 500; ++i) {  
 new InstanceCounter();  
 }  
 System.*out*.println("Created " + InstanceCounter.*getCount*() + " instances");  
 }  
}

This will produce the following result:

Started with 0 instances Created 500 instances

## The Final Modifier

**Final Variables**

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to a different object.

However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

**Example**

public class Test {  
 // The following are examples of declaring constants:   
 public static final int *BOXWIDTH* = 6;  
 static final String *TITLE* = "Manager";  
 final int value = 10;  
  
 public void changeValue() {  
 value = 12; //will give an error  
 }  
}

**Final Methods**

A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

**Example**

You declare methods using the *final* modifier in the class declaration, as in the following example:

**Final Classes**

public class Test {  
 public final void changeName() {  
// body of method  
 }  
}

The main purpose of using a class being declared as *final* is to prevent the class from being sub classed. If a class is marked as final, then no class can inherit any feature from the final class.

**Example**

public final class Test {  
// body of class  
}

## The Abstract Modifier

**Abstract Class**

An abstract class can never be instantiated. If a class is declared as abstract, then the sole purpose is for the class to be extended.

A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods, then the class should be declared abstract. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

**Example**

abstract class Caravan {  
 private double price;  
 private String model;  
 private String year;  
  
 public abstract void goFast(); //an abstract method  
  
 public abstract void changeColor();  
}

**Abstract Methods**

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

**Example**

public abstract class SuperClass {  
 abstract void m(); //abstract method  
}  
  
class SubClass extends SuperClass {  
 // implements the abstract method  
 void m() {  
 .........  
 }  
}

**The Synchronized Modifier**

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

**Example**

public synchronized void showDetails(){

.......

}

**The Transient Modifier**

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

**Example**

public transient int limit = 55; // will not persist public int b; // will persist

**The Volatile Modifier**

The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

**Example**

public class MyRunnable implements Runnable {  
 private volatile boolean active;  
  
 public void run() {  
 active = true;  
 while (active) { // line 1  
// some code here  
 }  
 }  
  
 public void stop() {  
 active = false; // line 2  
 }  
}

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1, the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

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.

public class className {  
// ...  
}  
 protected static final int *BOXWIDTH* = 42;  
 static final double *weeks* = 9.5;  
 private boolean myFlag;  
  
 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 synchronizedand *volatile* modifiers, which are used for threads.

# Java – 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:

|  |  |
| --- | --- |
| **Sr.No.** | **Operator and Example** |
| 1 | **+ ( Addition )**  Adds values on either side of the operator  **Example:** A + B will give 30 |
| 2 | **- ( Subtraction )**  Subtracts right-hand operand from left-hand operand  **Example:** A - B will give -10 |
| 3 | **\* ( Multiplication )**  Multiplies values on either side of the operator  **Example:** A \* B will give 200 |
| 4 | **/ (Division)**  Divides left-hand operand by right-hand operand  **Example:** B / A will give 2 |
| 5 | **% (Modulus)**  Divides left-hand operand by right-hand operand and returns remainder  **Example:** B % A will give 0 |
| 6 | **++ (Increment)**  Increases the value of operand by 1  **Example:** B++ gives 21 |
| 7 | **-- ( Decrement )**  Decreases the value of operand by 1  **Example:** B-- gives 19 |

**Example**

The following program is a simple example which demonstrates the arithmetic operators. Copy and paste the following Java program in Test.java file, and compile and run this program:

public class Test {  
 public static void main(String args[]) {  
 int a = 10;  
 int b = 20;  
 int c = 25;  
 int d = 25;  
 System.*out*.println("a + b = " + (a + b));  
 System.*out*.println("a - b = " + (a - b));  
 System.*out*.println("a \* b = " + (a \* b));  
 System.*out*.println("b / a = " + (b / a));  
 System.*out*.println("b % a = " + (b % a));  
 System.*out*.println("c % a = " + (c % a));  
 System.*out*.println("a++ = " + (a++));  
 System.*out*.println("b-- = " + (a--));  
// Check the difference in d++ and ++d   
 System.*out*.println("d++ = " + (d++));  
 System.*out*.println("++d = " + (++d));  
 }  
}

This will produce the following result:

a + b = 30  
a - b = -10  
a \* b = 200 b / a = 2  
b % a = 0 c % a = 5  
a++ = 10  
b-- = 11  
d++ = 25  
++d = 27

## The Relational Operators

There are following relational operators supported by Java language. Assume variable A holds 10 and variable B holds 20, then:

|  |  |
| --- | --- |
| **Sr. No.** | **Operator and Description** |
| 1 | **== (equal to)**  Checks if the values of two operands are equal or not, if yes then condition becomes true.  **Example:** (A == B) is not true. |
| 2 | **!= (not equal to)**  Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.  **Example:** (A != B) is true. |
| 3 | **> (greater than)**  Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.  **Example:** (A > B) is not true. |
| 4 | **< (less than)**  Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.  **Example:** (A < B) is true. |
| 5 | **>= (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.  **Example:** (A >= B) is not true. |
| 6 | **<= (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.  **Example:** (A <= B) is true. |

**Example**

The following program is a simple example that demonstrates the relational operators. Copy and paste the following Java program in Test.java file and compile and run this program.

public class Test {  
  
 public static void main(String args[]) {  
 int a = 10;  
 int b = 20;  
 System.*out*.println("a == b = " + (a == b));  
 System.*out*.println("a != b = " + (a != b));  
 System.*out*.println("a > b = " + (a > b));  
 System.*out*.println("a < b = " + (a < b));  
 System.*out*.println("b >= a = " + (b >= a));  
 System.*out*.println("b <= a = " + (b <= a));  
 }  
}

This will produce the following result:

a == b = false a != b = true a > b = false a < b = true b >= a = true b <= a = false

## 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:

|  |  |
| --- | --- |
| **Sr. No.** | **Operator and Description** |
| 1 | **& (bitwise and)**  Binary AND Operator copies a bit to the result if it exists in both operands.  **Example:** (A & B) will give 12 which is 0000 1100 |
| 2 | **| (bitwise or)**  **Binary OR Operator copies a bit if it exists in either operand.**  **Example: (A | B) will give 61 which is 0011 1101** |
| 3 | **^ (bitwise XOR)**  **Binary XOR Operator copies the bit if it is set in one operand but not both.**  **Example: (A ^ B) will give 49 which is 0011 0001** |
| 4 | **~ (bitwise compliment)**  **Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.**  **Example: (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.** |
| 5 | **<< (left shift)**  **Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.**  **Example: A << 2 will give 240 which is 1111 0000** |
| 6 | **>> (right shift)**  **Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.**  **Example: A >> 2 will give 15 which is 1111** |
| 7 | **>>> (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.**  **Example: A >>>2 will give 15 which is 0000 1111** |

**Example**

The following program is a simple example that demonstrates the bitwise operators. Copy and paste the following Java program in Test.java file and compile and run this program:

public class Test {  
 public static void main(String args[]) {  
 int a = 60; /\* 60 = 0011 1100 \*/  
 int b = 13; /\* 13 = 0000 1101 \*/  
 int c = 0;  
  
 c = a & b; /\* 12 = 0000 1100 \*/  
 System.*out*.println("a & b = " + c);  
  
 c = a | b; /\* 61 = 0011 1101 \*/  
 System.*out*.println("a | b = " + c);  
  
 c = a ^ b; /\* 49 = 0011 0001 \*/  
 System.*out*.println("a ^ b = " + c);  
  
 c = ~a; /\*-61 = 1100 0011 \*/  
 System.*out*.println("~a = " + c);  
  
 c = a << 2; /\* 240 = 1111 0000 \*/  
 System.*out*.println("a << 2 = " + c);  
  
 c = a >> 2; /\* 15 = 1111 \*/  
 System.*out*.println("a >> 2 = " + c);  
  
 c = a >>> 2; /\* 15 = 0000 1111 \*/  
 System.*out*.println("a >>> 2 = " + c);  
 }  
}

This will produce the following result:

a & b = 12 a | b = 61 a ^ b = 49

~a = -61

a << 2 = 240

a >> 15

a >>> 15

## The Logical Operators

The following table lists the logical operators:

Assume Boolean Variables A holds true and variable B holds false, then:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| 1 | **&& (logical and)**  Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.  **Example:** (A && B) is false. |
| 2 | **|| (logical or)**  Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.  **Example:** (A || B) is true. |
| 3 | **! (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.  **Example:** !(A && B) is true. |

**Example**

The following simple example program demonstrates the logical operators. Copy and paste the following Java program in Test.java file and compile and run this program:

public class Test {  
 public static void main(String args[]) {  
 boolean a = true;  
 boolean b = false;  
  
 System.*out*.println("a && b = " + (a && b));  
  
 System.*out*.println("a || b = " + (a || b));  
  
 System.*out*.println("!(a && b) = " + !(a && b));  
 }  
}

This will produce the following result:

a && b = false a || b = true

!(a && b) = true

## The Assignment Operators

Following are the assignment operators supported by Java language:

|  |  |
| --- | --- |
| **Sr. No.** | **Operator and Description** |
| 1 | **=**  Simple assignment operator. Assigns values from right side operands to left side operand.  **Example:** C = A + B will assign value of A + B into C |
| 2 | **+=**  **Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.**  **Example: C += A is equivalent to C = C + A** |
| 3 | **-=**  **Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.**  **Example:C -= A is equivalent to C = C – A** |
| 4 | **\*=**  **Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.**  **Example: C \*= A is equivalent to C = C \* A** |
| 5 | **/=**  **Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.**  **Example: C /= A is equivalent to C = C / A** |
| 6 | **%=**  **Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.**  **Example: C %= A is equivalent to C = C % A** |
| 7 | **<<=**  **Left shift AND assignment operator.**  **Example: C <<= 2 is same as C = C << 2** |
| 8 | **>>=**  **Right shift AND assignment operator**  **Example: C >>= 2 is same as C = C >> 2** |
| 9 | **&=**  **Bitwise AND assignment operator.**  **Example: C &= 2 is same as C = C & 2** |
| 10 | **^=**  **bitwise exclusive OR and assignment operator.**  **Example: C ^= 2 is same as C = C ^ 2** |
| 11 | **|=**  **bitwise inclusive OR and assignment operator.**  **Example: C |= 2 is same as C = C | 2** |

**Example**

The following program is a simple example that demonstrates the assignment operators. Copy and paste the following Java program in Test.java file. Compile and run this program:

public class Test {  
 public static void main(String args[]) { int a = 10;  
 int b = 20; int c = 0;  
 c = a + b;  
 System.*out*.println("c = a + b = " + c );  
  
 c += a ;  
 System.*out*.println("c += a = " + c );  
  
 c -= a ;  
 System.*out*.println("c -= a = " + c );  
  
 c \*= a ;  
 System.*out*.println("c \*= a = " + c );  
 a = 10;  
 c = 15;  
 c /= a ;  
 System.*out*.println("c /= a = " + c );  
  
 a = 10;  
 c = 15;  
 c %= a ;  
 System.*out*.println("c %= a = " + c );  
  
 c <<= 2 ;  
 System.*out*.println("c <<= 2 = " + c );  
  
 c >>= 2 ;  
 System.*out*.println("c >>= 2 = " + c );  
  
 c >>= 2 ;  
 System.*out*.println("c >>= a = " + c );  
  
 c &= a ;  
 System.*out*.println("c &= 2 = " + c );  
  
 c ^= a ;  
 System.*out*.println("c ^= a = " + c );  
  
 c |= a ;  
 System.*out*.println("c |= a = " + c );  
 }  
}

This will produce the following result:

c = a + b = 30 c += a = 40

c -= a = 30

c \*= a = 300

c /= a = 1 c %= a = 5

c <<= 2 = 20

c >>= 2 = 5

c >>= 2 = 1

c &= a = 0

c ^= a = 10

c |= a = 10

## 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:

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:

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:

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:

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.

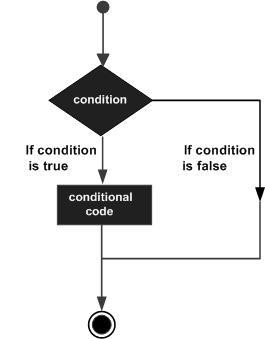
Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] . (dot operator) | Left toright |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | >> >>> << | Left to right |
| Relational | > >= < <= | 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 |

# Java – 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.

|  |  |
| --- | --- |
| **Statement** | **Description** |
| **if statement** | An **if statement** consists of a Boolean expression followed by one or more statements. |
| **if...else statement** | An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false. |
| **nested if statements** | You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |
| **switch statement** | A **switch** statement allows a variable to be tested for equality against a list of values. |

## If Statement in Java

An **if** statement consists of a Boolean expression followed by one or more statements.

**Syntax**

Following is the syntax of an if statement:

if(Boolean\_expression)

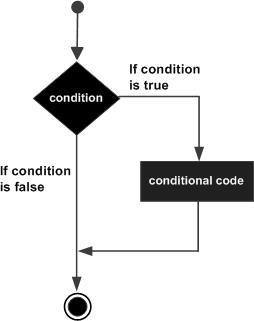
{

//Statements will execute if the Boolean expression is true

}

If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

**Flow Diagram**



**Example**

public class Test {  
 public static void main(String args[]) {  
 int x = 10;  
 if (x < 20) {  
 System.*out*.print("This is if statement");  
 }  
 }  
}

This will produce the following result:

This is if statement.

## If-else Statement in Java

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

**Syntax**

Following is the syntax of an if...else statement:

if(Boolean\_expression){

//Executes when the Boolean expression is true

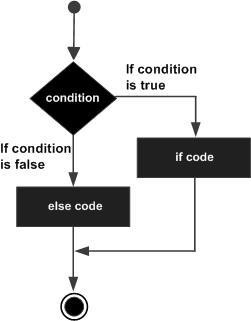
}else{

//Executes when the Boolean expression is false

}

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

**Flow Diagram**



**Example**

public class Test {  
 public static void main(String args[]) {  
 int x = 30;  
 if (x < 20) {  
 System.*out*.print("This is if statement");  
 } else {  
 System.*out*.print("This is else statement");  
 }  
 }  
}

This will produce the following result:

This is else statement

## The if...else if...else Statement

An if statement can be followed by an optional *else if...else* statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are a few points to keep in mind.

* An if can have zero or one else's and it must come after any else if's.
* An if can have zero to many else if's and they must come before the else.
* Once an else if succeeds, none of the remaining else if's or else's will be tested.

**Syntax**

Following is the syntax of an if...else statement:

if(Boolean\_expression 1){

//Executes when the Boolean expression 1 is true

}else if(Boolean\_expression 2){

//Executes when the Boolean expression 2 is true

}else if(Boolean\_expression 3){

//Executes when the Boolean expression 3 is true

}else {

//Executes when the none of the above condition is true.

}

**Example**

public class Test {  
 public static void main(String args[]) {  
 int x = 30;  
 if (x == 10) {  
 System.*out*.print("Value of X is 10");  
 } else if (x == 20) {  
 System.*out*.print("Value of X is 20");  
 } else if (x == 30) {  
 System.*out*.print("Value of X is 30");  
 } else {  
 System.*out*.print("This is else statement");  
 }  
 }  
}

This will produce the following result:

Value of X is 30

## Nested if Statement in Java

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

**Syntax**

The syntax for a nested if...else is as follows:

if(Boolean\_expression 1){

//Executes when the Boolean expression 1 is true if(Boolean\_expression 2){

//Executes when the Boolean expression 2 is true

}

}

You can nest **else if...else** in the similar way as we have nested *if* statement.

**Example**

public class Test {  
 public static void main(String args[]) {  
 int x = 30;  
 int y = 10;  
  
 if (x == 30) {  
 if (y == 10) {  
 System.*out*.print("X = 30 and Y = 10");  
 }  
 }  
 }  
}

This will produce the following result:

X = 30 and Y = 10

## Switch Statement in Java

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

**Syntax**

The syntax of enhanced for loop is:

switch(expression){ case value :

//Statements break; //optional

case value :

//Statements break; //optional

//You can have any number of case statements. default : //Optional

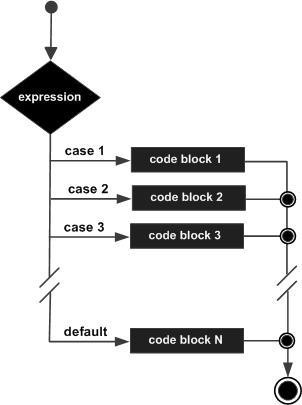
//Statements

}

The following rules apply to a **switch** statement:

* The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
* You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
* When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
* When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
* Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
* A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

**Flow Diagram**



**Example**

public class Test {  
 public static void main(String args[]) {  
 //char grade = args[0].charAt(0);  
 char grade = 'C';  
  
 switch (grade) {  
 case 'A':  
 System.*out*.println("Excellent!");  
 break;  
 case 'B':  
 case 'C':  
 System.*out*.println("Well done");  
 break;  
 case 'D':  
 System.*out*.println("You passed");  
 case 'F':  
 System.*out*.println("Better try again");  
 break;  
 default:  
 System.*out*.println("Invalid grade");  
 }  
 System.*out*.println("Your grade is " + grade);  
 }  
}

Compile and run the above program using various command line arguments. This will produce the following result:

$ java Test Well done

Your grade is a C

$

## 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.

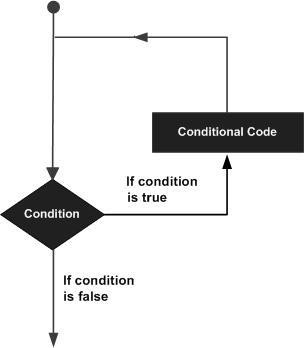
* If the value of exp1 is true, then the value of Exp2 will be the value of the whole expression.
* If the value of exp1 is false, then Exp3 is evaluated and its value becomes the value of the entire expression.

# Java – 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.

|  |  |
| --- | --- |
| **Loop Type** | **Description** |
| **while loop** | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| **for loop** | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| **do...while loop** | Like a while statement, except that it tests the condition at the end of the loop body. |

## While Loop in Java

A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

**Syntax**

The syntax of a while loop is:

while(Boolean\_expression)

{

//Statements

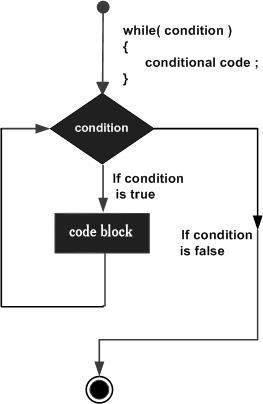
}

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value.

When executing, if the *boolean\_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

**Flow Diagram**



Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

**Example**

public class Test {  
  
 public static void main(String args[]) {  
 int x = 10;  
  
 while (x < 20) {  
 System.*out*.print("value of x : " + x);  
 x++;  
 System.*out*.print("\n");  
 }  
 }  
}

This will produce the following result:

value of x : 10 value of x : 11 value of x : 12 value of x : 13 value of x : 14 value of x : 15 value of x : 16 value of x : 17 value of x : 18 value of x : 19

## for Loop in Java

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A **for** loop is useful when you know how many times a task is to be repeated.

**Syntax**

The syntax of a for loop is:

for(initialization; Boolean\_expression; update)

{

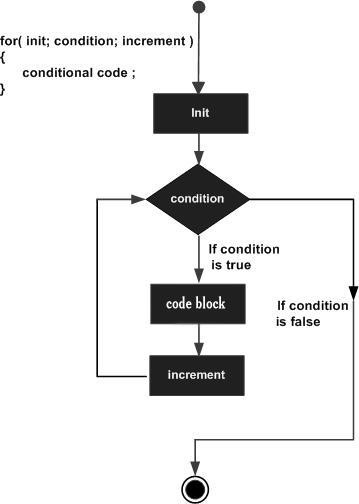
//Statements

}

Here is the flow of control in a **for** loop:

* The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
* Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
* After the body of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.
* The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

**Flow Diagram**



**Example**

Following is an example code of the for loop in Java.

public class Test {  
 public static void main(String args[]) {  
 for (int x = 10; x < 20; x = x + 1) {  
 System.*out*.print("value of x : " + x);  
 System.*out*.print("\n");  
 }  
 }  
}

This will produce the following result:

value of x : 10 value of x : 11 value of x : 12 value of x : 13 value of x : 14 value of x : 15 value of x : 16 value of x : 17 value of x : 18 value of x : 19

## Do While Loop in Java

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

**Syntax**

Following is the syntax of a do...while loop:

do

{

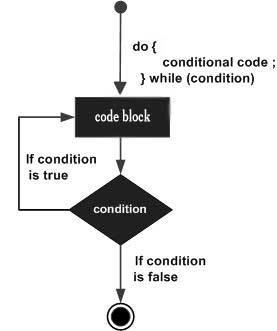
//Statements

}while(Boolean\_expression);

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

**Flow Diagram**



**Example**

public class Test {  
 public static void main(String args[]) {  
 int x = 10;  
  
 do {  
 System.*out*.print("value of x : " + x);  
 x++;  
 System.*out*.print("\n");  
 } while (x < 20);  
 }  
}

This will produce the following result:

value of x : 10 value of x : 11 value of x : 12 value of x : 13 value of x : 14 value of x : 15 value of x : 16 value of x : 17 value of x : 18 value of x : 19

## 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.

|  |  |
| --- | --- |
| **Control Statement** | **Description** |
| **break statement** | Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| **continue statement** | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |

## Break Statement in Java

The **break** statement in Java programming language has the following two usages:

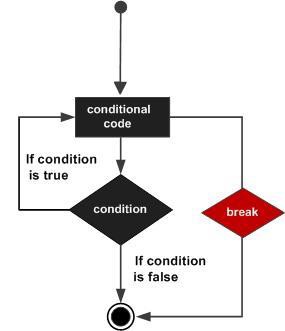
* When the break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
* It can be used to terminate a case in the **switch** statement (covered in the next chapter).

**Syntax**

The syntax of a break is a single statement inside any loop:

break;

**Flow Diagram**



**Example**

public class Test {  
 public static void main(String args[]) {  
 int[] numbers = {10, 20, 30, 40, 50};  
 for (int x : numbers) {  
 if (x == 30) {  
 break;  
 }  
 System.*out*.print(x);  
 System.*out*.print("\n");  
 }  
 }  
}

This will produce the following result:

10

20

## Continue Statement in Java

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

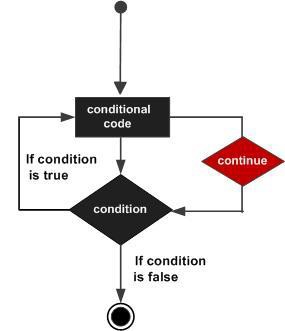
* In a for loop, the continue keyword causes control to immediately jump to the update statement.
* In a while loop or do/while loop, control immediately jumps to the Boolean expression.

**Syntax**

The syntax of a continue is a single statement inside any loop:

continue;

**Flow Diagram**



**Example**

public class Test {  
 public static void main(String args[]) {  
 int[] numbers = {10, 20, 30, 40, 50};  
 for (int x : numbers) {  
 if (x == 30) {  
 continue;  
 }  
 System.*out*.print(x);  
 System.*out*.print("\n");  
 }  
 }  
}

This will produce the following result:

10

20

40

50

## 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(",");  
 }  
 System.*out*.print("\n");  
 String[] names = {"James", "Larry", "Tom", "Lacy"};  
 for (String name : names) {  
 System.*out*.print(name);  
 System.*out*.print(",");  
 }  
 }  
}

This will produce the following result:

10,20,30,40,50,

James,Larry,Tom,Lacy,

# Java – 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 tutorial 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:

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:

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};

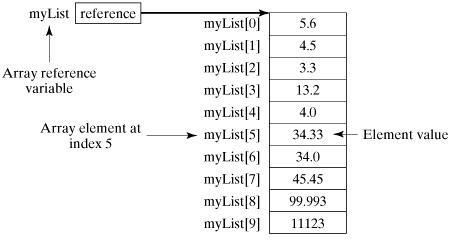
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

**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:

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:

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:

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.

|  |  |
| --- | --- |
| **Sr. No.** | **Methods with 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.)** |

# Java – 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:

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:

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 **max()**. 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. Let’s consider an example:

System.out.println("This is example.com!");

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

int result = sum(6, 9);

**Example**

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

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:

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 *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown in the following example.

**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:

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 remain 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);  
  
 // Invoke the swap method swapFunction(a, b);  
 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:

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:

public class ExampleOverloading {  
  
 public static void main(String[] args) {  
 int a = 11;  
 int b = 6;  
 double c = 7.3;  
 double 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:

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:

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

args[6]: -100

## The 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 startup 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.

**Example**

Here is a simple example that uses a constructor without parameters:

// A simple constructor.  
class MyClass {  
 int x;  
  
 // Following is the constructor  
 MyClass() {  
 x = 10;  
 }  
}

You will have to 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.x + " " + t2.x);  
 }  
}

## Parameterized Constructor

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 with a parameter:

// A simple constructor.   
class MyClass {  
 int x;  
  
 // Following is the constructor   
 MyClass(int i) {  
 x = i;  
 }  
}

You will need to call a 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);  
 }  
}

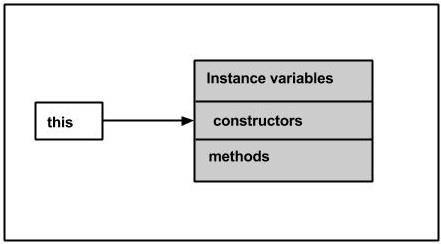
This will produce the following result:

10 20

## 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 (parametrized 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 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 parametrized constructor  
 This\_Example obj2 = new This\_Example(30);  
 //Invoking the print method again   
 obj2.print();  
 }  
  
 public void greet() {  
 System.*out*.println("Hi Welcome to Example");  
 }  
  
 public void print() {  
 //Local variable num   
 int num = 20;  
 //Printing the instance variable  
 System.*out*.println("value of local variable num is : " + num);  
 //Printing the local variable  
 System.*out*.println("value of instance variable num is : " + this.num);  
 //Invoking the greet method of a class this.greet();  
 }  
}

This will produce the following result:

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 Example

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 Example

## 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:

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.

# Java – 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 occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation; 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.

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 *ArrayIndexOutOfBoundsExceptionexception* 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.

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.

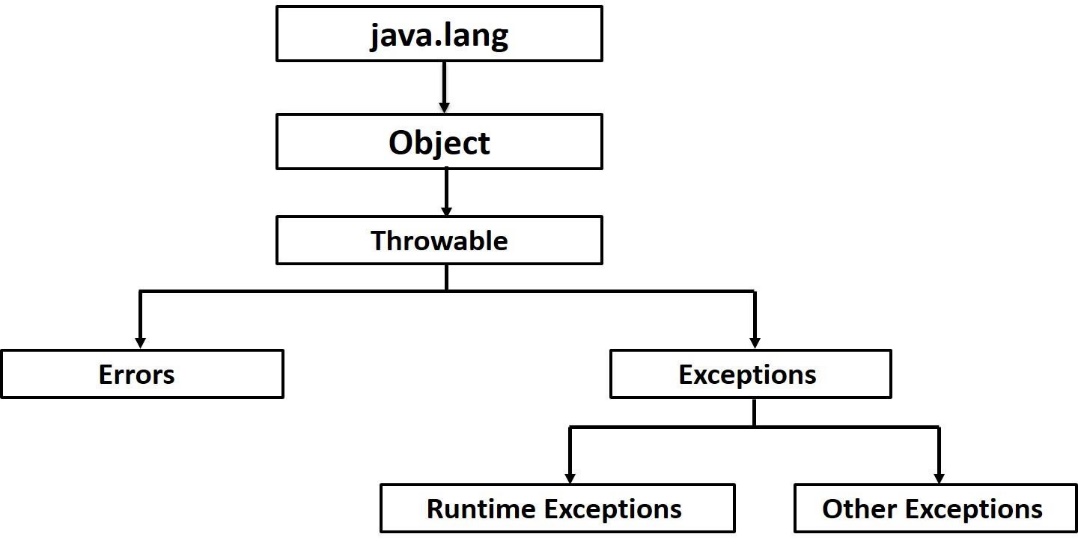
Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by theruntime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.

Following is a list of most common checked and unchecked Java's Built-in Exceptions.

## Built-in Exceptions

Java defines several exception classes inside the standard package **java.lang**.



The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

|  |  |
| --- | --- |
| **Exception** | **Description** |
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with the current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| UnsupportedOperationException | An unsupported operation was encountered. |

Following is the list of Java Checked Exceptions Defined in java.lang.

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

## Exceptions Methods

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

|  |  |
| --- | --- |
| **Sr. No.** | **Methods with 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:

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:

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:

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.

**Example**

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:

## 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.

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

A finally block appears at the end of the catch blocks and has the following 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:

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.

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.

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 deceleration 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.

**Example**

// 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.

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:

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 catergories 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.

# Java – Inner Classes

## 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.

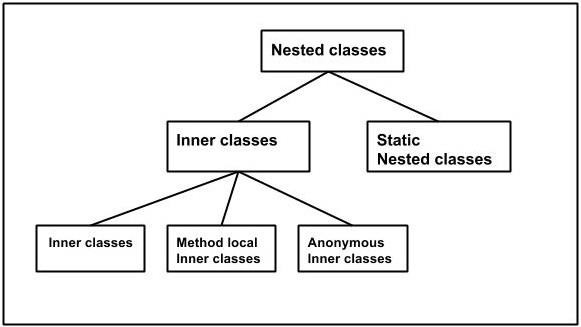
class Outer\_Demo{ class Nested\_Demo{

}

}

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;  
  
 //Accessing he inner class from the method within  
 void display\_Inner() {  
 Inner\_Demo inner = new Inner\_Demo();  
  
 //inner class  
 private class Inner\_Demo {  
 public void print() {  
 System.*out*.println("This is an inner class");  
 }  
 }  
 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.

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.

The value of num in the class Test is: 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 {  
  
 public static void main(String args[]) {  
 Outerclass outer = new Outerclass();  
 outer.my\_Method();  
 }  
  
 //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();  
 }  
}

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

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:

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.

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.

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 Message {  
 String greet();  
}  
  
public class My\_class {  
 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";  
 }  
 });  
 }  
  
 //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 calss as an argument");  
 }  
}

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

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:

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 {  
 public static void main(String args[]) {  
 Outer.Nested\_Demo nested = new Outer.Nested\_Demo();  
 nested.my\_method();  
 }  
  
 static class Nested\_Demo {  
 public void my\_method() {  
 System.*out*.println("This is my nested class");  
 }  
 }  
}

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

This is my nested class

# Java – 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.

To have more detail about this interface, check [The Enumeration](http://www.tutorialspoint.com/java/java_enumeration_interface.htm).

### The Enumeration Interface

The Enumeration 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. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

The methods declared by Enumeration are summarized in the following table:

|  |  |
| --- | --- |
| **Sr. No.** | **Methods with Description** |
| 1 | **boolean hasMoreElements( )**  When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated. |
| 2 | **Object nextElement( )**  This returns the next object in the enumeration as a generic Object reference. |

### Example

Following is an example showing usage of Enumeration.

import java.util.Vector;  
import java.util.Enumeration;  
  
public class EnumerationTester {  
  
 public static void main(String args[]) {  
 Enumeration days;  
 Vector dayNames = new Vector();  
 dayNames.add("Sunday");  
 dayNames.add("Monday");  
 dayNames.add("Tuesday");  
 dayNames.add("Wednesday");  
 dayNames.add("Thursday");  
 dayNames.add("Friday");  
 dayNames.add("Saturday");  
 days = dayNames.elements();  
 while (days.hasMoreElements()) {  
 System.*out*.println(days.nextElement());  
 }  
 }  
}

This will produce the following result:

Sunday Monday Tuesday Wednesday Thursday Friday Saturday

## 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.

For more details about this class, check The BitSet.

### The BitSet Class

The BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits. This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.

The BitSet defines the following two constructors.

|  |  |
| --- | --- |
| **Sr. No.** | **Constructor and Description** |
| 1 | **BitSet( )**  This constructor creates a default object |
| 2 | **BitSet(int size)**  This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero |

### Example

The following program illustrates several of the methods supported by this data structure:

import java.util.BitSet;  
  
public class BitSetDemo {  
  
 public static void main(String args[]) {  
 BitSet bits1 = new BitSet(16);  
 BitSet bits2 = new BitSet(16);  
  
 // set some bits  
 for (int i = 0; i < 16; i++) {  
 if ((i % 2) == 0) bits1.set(i);  
 if ((i % 5) != 0) bits2.set(i);  
 }  
 System.*out*.println("Initial pattern in bits1: ");  
 System.*out*.println(bits1);  
 System.*out*.println("\nInitial pattern in bits2: ");  
 System.*out*.println(bits2);  
 // AND bits bits2.and(bits1);  
 System.*out*.println("\nbits2 AND bits1: ");  
 System.*out*.println(bits2);  
  
 // OR bits bits2.or(bits1);  
 System.*out*.println("\nbits2 OR bits1: ");  
 System.*out*.println(bits2);  
  
 // XOR bits bits2.xor(bits1);  
 System.*out*.println("\nbits2 XOR bits1: ");  
 System.*out*.println(bits2);  
 }  
}

This will produce the following result:

Initial pattern in bits1:

{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:

{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:

{2, 4, 6, 8, 12, 14}

bits2 OR bits1:

{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:

{}

## 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.

For more details about this class, check The Vector.

### The Vector Class

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

* Vector is synchronized.
* Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

|  |  |
| --- | --- |
| **Sr. No.** | **Constructor and Description** |
| 1 | **Vector( )**  This constructor creates a default vector, which has an initial size of 10 |
| 2 | **Vector(int size)**  This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size |
| 3 | **Vector(int size, int incr)**  This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward |
| 4 | **Vector(Collection c)**  This constructor creates a vector that contains the elements of collection c |

### Example

The following program illustrates several of the methods supported by this collection:

import java.util.\*;  
  
public class VectorDemo {  
  
 public static void main(String args[]) {  
 // initial size is 3, increment is 2  
 Vector v = new Vector(3, 2);  
 System.*out*.println("Initial size: " + v.size());  
  
 System.*out*.println("Initial capacity: " + v.capacity());  
 v.addElement(new Integer(1));  
 v.addElement(new Integer(2));  
 v.addElement(new Integer(3));  
 v.addElement(new Integer(4));  
 System.*out*.println("Capacity after four additions: " + v.capacity());  
 v.addElement(new Double(5.45));  
 System.*out*.println("Current capacity: " + v.capacity());  
 v.addElement(new Double(6.08));  
 v.addElement(new Integer(7));  
 System.*out*.println("Current capacity: " + v.capacity());  
 v.addElement(new Float(9.4));  
 v.addElement(new Integer(10));  
 System.*out*.println("Current capacity: " + v.capacity());  
 v.addElement(new Integer(11));  
 v.addElement(new Integer(12));  
 System.*out*.println("First element: " +  
 (Integer) v.firstElement());  
 System.*out*.println("Last element: " +  
 (Integer) v.lastElement());  
 if (v.contains(new Integer(3)))  
 System.*out*.println("Vector contains 3.");  
 // enumerate the elements in the vector.  
 Enumeration vEnum = v.elements();  
 System.*out*.println("\nElements in vector:");  
 while (vEnum.hasMoreElements())  
 System.*out*.print(vEnum.nextElement() + " ");  
 System.*out*.println();  
 }  
}

This will produce the following result:

Initial size: 0

Initial capacity: 3

Capacity after four additions: 5 Current capacity: 5

Current capacity: 7

Current capacity: 9

First element: 1

Last element: 12

Vector contains 3.

Elements in vector:

1 2 3 4 5.45 6.08 7 9.4 10 11 12

## 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.

For more details about this class, check The Stack.

**The Stack Class**

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

Stack( )

Apart from the methods inherited from its parent class Vector, Stack defines the following methods:

### Example

The following program illustrates several of the methods supported by this collection:

import java.util.\*;  
  
public class StackDemo {  
  
 static void showpush(Stack st, int a) {  
 st.push(new Integer(a));  
 System.*out*.println("push(" + a + ")");  
 System.*out*.println("stack: " + st);  
 }  
  
  
 static void showpop(Stack st) {  
 System.*out*.print("pop -> ");  
 Integer a = (Integer) st.pop();  
 System.*out*.println(a);  
 System.*out*.println("stack: " + st);  
 }  
  
  
 public static void main(String args[]) {  
 Stack st = new Stack();  
 System.*out*.println("stack: " + st);  
 *showpush*(st, 42);  
 *showpush*(st, 66);  
 *showpush*(st, 99);  
 *showpop*(st);  
 *showpop*(st);  
 *showpop*(st);  
 try {  
 *showpop*(st);  
 } catch (EmptyStackException e) {  
 System.*out*.println("empty stack");  
 }  
 }  
}

This will produce the following result:

stack: [ ] push(42) stack: [42] push(66) stack: [42, 66] push(99)

stack: [42, 66, 99]

pop -> 99

stack: [42, 66]

pop -> 66

stack: [42]

pop -> 42 stack: [ ]

pop -> empty stack

## 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.

For more details about this class, check The Dictionary.

### The Dictionary Class

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.

Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

The abstract methods defined by Dictionary are listed below:

|  |  |
| --- | --- |
| **Sr. No.** | **Methods with Description** |
| 1 | **Enumeration elements( )**  Returns an enumeration of the values contained in the dictionary. |
| 2 | **Object get(Object key)**  Returns the object that contains the value associated with the key. If the key is not in the dictionary, a null object is returned. |
| 3 | **boolean isEmpty( )**  **Returns true if the dictionary is empty, and returns false if it contains at least one key.** |
| 4 | **Enumeration keys( )**  **Returns an enumeration of the keys contained in the dictionary.** |
| 5 | **Object put(Object key, Object value)**  **Inserts a key and its value into the dictionary. Returns null if the key is not already in the dictionary; returns the previous value associated with the key if the key is already in the dictionary.** |
| 6 | **Object remove(Object key)**  **Removes the key and its value. Returns the value associated with the key. If the key is not in the dictionary, a null is returned.** |
| 7 | **int size( )**  **Returns the number of entries in the dictionary.** |

The Dictionary class is obsolete. You should implement the [Map interface](http://www.tutorialspoint.com/java/java_map_interface.htm) to obtain key/value storage functionality.

### Example

Map has its implementation in various classes like HashMap. Following is an example to explain map functionality:

import java.util.\*;  
  
public class CollectionsDemo {  
  
 public static void main(String[] args) {  
 Map m1 = new HashMap();  
 m1.put("Zara", "8");  
 m1.put("Mahnaz", "31");  
 m1.put("Ayan", "12");  
 m1.put("Daisy", "14");  
 System.*out*.println();  
 System.*out*.println(" Map Elements");  
 System.*out*.print("\t" + m1);  
 }  
}

This will produce the following result:

ap Elements

{Mahnaz=31, Ayan=12, Daisy=14, Zara=8}

## 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.

For more detail about this class, check The Hashtable.

### The Hashtable Class

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

However, Java 2 re-engineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Following is the list of constructors provided by the HashTable class.

|  |  |
| --- | --- |
| **Sr. No.** | **Constructor and Description** |
| 1 | **Hashtable( )**  This is the default constructor of the hash table it instantiates the Hashtable class. |
| 2 | **Hashtable(int size)**  This constructor accepts an integer parameter and creates a hash table that has an initial size specified by integer value size. |
| 3 | **Hashtable(int size, float fillRatio)**  **This creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.** |
| 4 | **Hashtable(Map<? extends K, ? extends V> t)**  **This constructs a Hashtable with the given mappings.** |

### Example

The following program illustrates several of the methods supported by this data structure:

import java.util.\*;  
  
public class HashTableDemo {  
  
 public static void main(String args[]) {  
 // Create a hash map  
 Hashtable balance = new Hashtable();  
 Enumeration names;  
 String str;  
 double bal;  
 balance.put("Zara", new Double(3434.34));  
 balance.put("Mahnaz", new Double(123.22));  
 balance.put("Ayan", new Double(1378.00));  
 balance.put("Daisy", new Double(99.22));  
 balance.put("Qadir", new Double(-19.08));  
  
 // Show all balances in hash table.   
 names = balance.keys();  
 while (names.hasMoreElements()) {  
 str = (String) names.nextElement();  
 System.*out*.println(str + ": " + balance.get(str));  
 }  
 System.*out*.println();  
 // Deposit 1,000 into Zara's account  
 bal = ((Double) balance.get("Zara")).doubleValue();  
 balance.put("Zara", new Double(bal + 1000));  
 System.*out*.println("Zara's new balance: " + balance.get("Zara"));  
 }  
}

This will produce the following result:

Qadir: -19.08

Zara: 3434.34

ahnaz: 123.22

Daisy: 99.22

Ayan: 1378.0

Zara's new balance: 4434.34

## 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.

For more detail about this class, check The Properties.

### The Properties Class

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.

Properties define the following instance variable. This variable holds a default property list associated with a Properties object.

Properties defaults;

Following is the list of constructors provided by the properties class.

|  |  |
| --- | --- |
| **Sr. No.** | **Constructors and Description** |
| 1 | **Properties( )**  This constructor creates a Properties object that has no default values. |
| 2 | **Properties(Properties propDefault)**  Creates an object that uses propDefault for its default values. In both cases, the property list is empty. |

### Example

The following program illustrates several of the methods supported by this data structure:

import java.util.\*;  
  
  
public class PropDemo {  
  
 public static void main(String args[]) {  
 Properties capitals = new Properties();  
 Set states;  
 String str;  
  
 capitals.put("Illinois", "Springfield");  
 capitals.put("Missouri", "Jefferson City");  
 capitals.put("Washington", "Olympia");  
 capitals.put("California", "Sacramento");  
 capitals.put("Indiana", "Indianapolis");  
  
 // Show all states and capitals in hashtable.  
 states = capitals.keySet();  
 // get set-view of keys  
 Iterator itr = states.iterator();  
 while (itr.hasNext()) {  
 str = (String) itr.next();  
 System.*out*.println("The capital of " +  
 str + " is " + capitals.getProperty(str) + ".");  
 }  
 System.*out*.println();  
  
  
 // look for state not in list -- specify default   
 str = capitals.getProperty("Florida", "Not Found");  
 System.*out*.println("The capital of Florida is "  
 + str + ".");  
 }  
}

This will produce the following result:

The capital of Missouri is Jefferson City. The capital of Illinois is Springfield.

The capital of Indiana is Indianapolis. The capital of California is Sacramento. The capital of Washington is Olympia.

The capital of Florida is Not Found.

# Java - Object Oriented

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 Parsing

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.

public class Dog {  
 String breed;  
 int ageC  
 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:

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

Java also supports [Singleton Classes](http://www.tutorialspoint.com/java/java_using_singleton.htm) 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.

## How to Use Singleton Class?

The Singleton's purpose is to control object creation, limiting the number of objects to only one. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources, such as database connections or sockets.

For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time.

## Implementing Singletons

### Example 1

The easiest implementation consists of a private constructor and a field to hold its result, and a static accessor method with a name like getInstance().

The private field can be assigned from within a static initializer block or, more simply, using an initializer. The getInstance( ) method (which must be public) then simply returns this instance –

// File Name: Singleton.java  
public class Singleton {  
 private static Singleton *singleton* = new Singleton();  
  
 /\* A private Constructor prevents any other  
 \* class from instantiating.  
 \*/  
 private Singleton() {  
 }  
  
 /\* Static 'instance' method \*/  
 public static Singleton getInstance() {  
 return *singleton*;  
 }  
  
 /\* Other methods protected by singleton-ness \*/  
 protected static void demoMethod() {  
 System.*out*.println("demoMethod for singleton");  
 }  
}

Here is the main program file, where we will create a singleton object:

// File Name: SingletonDemo.java   
public class SingletonDemo {  
 public static void main(String[] args) {  
 Singleton tmp = Singleton.getInstance();  
 tmp.demoMethod();  
 }  
}

This will produce the following result −

demoMethod for singleton

### Example 2

Following implementation shows a classic Singleton design pattern:

public class ClassicSingleton {  
  
 private static ClassicSingleton *instance* = null;  
  
 private ClassicSingleton() {  
 // Exists only to defeat instantiation.  
 }  
  
 public static ClassicSingleton getInstance() {  
 if (*instance* == null) {  
 *instance* = new ClassicSingleton();  
 }  
 return *instance*;  
 }  
}

The ClassicSingleton class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

Here, ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed.

## 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:

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:

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

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

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 above mentioned 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.

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 tutorial, 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:

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

## Java – 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.

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 Substraction(int x, int y) {  
 z = x - y;  
 System.*out*.println("The difference between the given numbers:" + z);  
 }  
}  
  
public class My\_Calculation extends Calculation {  
 public static void main(String args[]) {  
 int a = 20, b = 10;  
 My\_Calculation demo = new My\_Calculation();  
 demo.addition(a, b);  
 demo.Substraction(a, b);  
 demo.multiplication(a, b);  
 }  
  
 public void multiplication(int x, int y) {  
 z = x \* y;  
 System.*out*.println("The product of the given numbers:" + z);  
 }  
}

Compile and execute the above code as shown below.

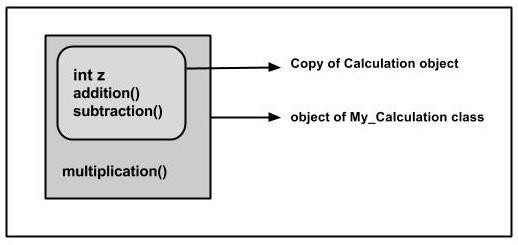
javac My\_Calculation.java java My\_Calculation

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

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 (Calin this case) you cannot call the method **multiplication()**, which belongs to the subclass My\_Calculation.

Calculation cal = 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;  
  
 public static void main(String args[]) {  
 Sub\_class obj = new Sub\_class();  
 obj.my\_method();  
  
 }  
 //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);  
 }  
}

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 −

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 parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a string 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 argd[]) {  
 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 −

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

### IS-ARelationship

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 −

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.

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:

true true true

### HAS-Arelationship

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.

Let’s look into an example −

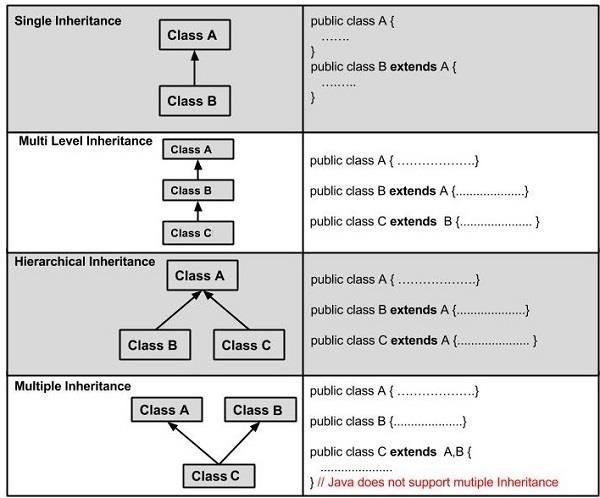
public class Vehicle {  
}  
  
public class Speed {  
}  
  
public class Van extends Vehicle {  
 private Speed sp;  
}

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.



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 had helped Java get rid of the impossibility of multiple inheritance.

## Java – 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:

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:

TestDog.java:30: cannot find symbol symbol : method bark()

location: class Animal

b.bark();

^

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 overridding method in the sub lass 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 uncheck 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.

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:

Animals can move Dogs can walk and run

## Java – 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 refers 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.

/\* 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:

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. Invoking 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 the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby 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.

## Java – 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:

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.

### 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 semi colon (;) 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  
}

## Java – 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.

**Example**

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 void setAge(int newAge) {  
 age = newAge;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String newName) {  
 name = newName;  
 }  
  
 public String getIdNum() {  
 return idNum;  
 }  
  
 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:

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.
* The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

## Java – 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:

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.

/\* 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\  
}

**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.

This will produce the following result:

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.

/\* File name : MammalInt.java \*/  
public class MammalInt implements Animal {  
  
 public static void main(String args[]) {  
 MammalInt m = new MammalInt();  
 m.eat();  
 m.travel();  
 }  
  
 public void eat() {  
 System.*out*.println("Mammal eats");  
 }  
  
 public void travel() {  
 System.*out*.println("Mammal travels");  
 }  
  
 public int noOfLegs() {  
 return 0;  
 }  
}

When implementation 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:

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:

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.

## Java – 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*:

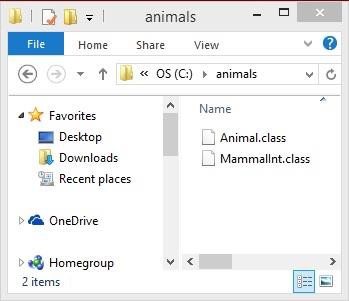
package animals;  
  
/\* File name : MammalInt.java \*/  
public class MammalInt implements Animal {  
  
 public static void main(String args[]) {  
 MammalInt m = new MammalInt();  
 m.eat();  
 m.travel();  
 }  
  
 public void eat() {  
 System.*out*.println("Mammal eats");  
 }  
  
 public void travel() {  
 System.*out*.println("Mammal travels");  
 }  
  
 public int noOfLegs() {  
 return 0;  
 }  
}

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.

$ java animals.MammalInt 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