# **UNIT-I**

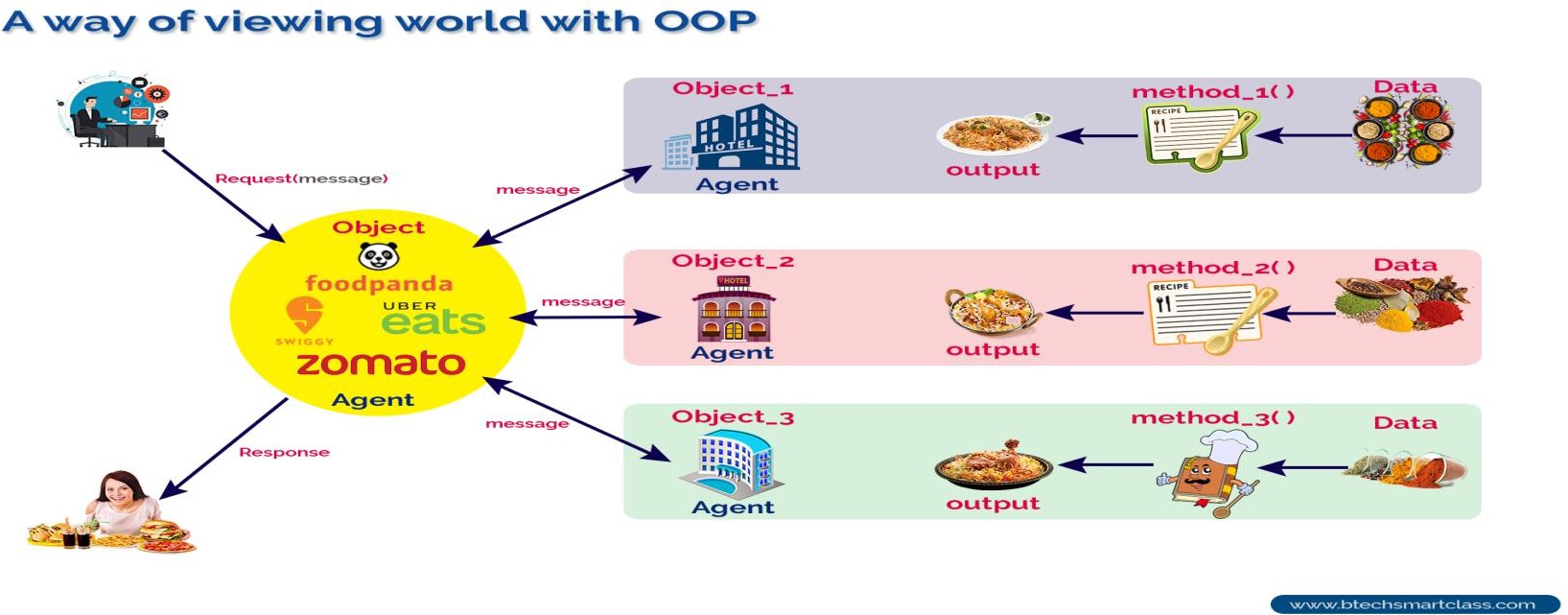
# Object oriented thinking and Java Basics- Need for oop paradigm, summary of oop concepts, coping with complexity, abstraction mechanisms. A way of viewing world – Agents, responsibility, messages, methods, History of Java, Java buzzwords, data types, variables, scope and lifetime of variables, arrays, operators, expressions, control statements, type conversion and casting, simple java program, concepts of classes, objects, constructors, methods, access control, this keyword, garbage collection, overloading methods and constructors, method binding, inheritance, overriding and exceptions, parameter passing, recursion, nested and inner classes, exploring string class.

**A way of Viewing the World**

A way of viewing the world is an idea to illustrate the object-oriented programming concept with an example of a real-world situation.

Let us consider a situation, I am at my office and I wish to get food to my family members who are at my home from a hotel. Because of the distance from my office to home, there is no possibility of getting food from a hotel myself. So, how do we solve the issue?

To solve the problem, let me call zomato (an **agent** in food delevery community), tell them the variety and quantity of food and the hotel name from which I wish to delever the food to my family members. Look at the following image.

**OverView of java**

##### Agents and Communities

To solve my food delivery problem, I used a solution by finding an appropriate agent Zomato) and pass a message containing my request. It is the responsibility of the agent (Zomato) to satisfy my request. Here, the agent uses some method to do this. I do not need to know the method that the agent has used to solve my request. This is usually hidden from me.So, in object-oriented programming, problem-solving is the solution to our problem which requires the help of many individuals in the community. We may describe agents and communities as follows.

**An object-oriented program is structured as a community of interacting agents, called objects. Where each object provides a service (data and methods) that is used by other members of the community.**

In our example, the online food delivery system is a community in which the agents are zomato and set of hotels. Each hotel provides a variety of services that can be used by other members like zomato, myself, and my family in the community.

##### Messages and Methods

To solve my problem, I started with a request to the agent zomato, which led to still more requestes among the members of the community until my request has done. Here, the members of a community interact with one another by making requests until the problem has satisfied.

**In object-oriented programming, every action is initiated by passing a message to an agent (object), which is responsible for the action. The receiver is the object to whom the message was sent. In response to the message, the receiver performs some method to carry out the request. Every message may include any additional information as arguments.**

In our example, I send a request to zomato with a message that contains food items, the quantity of food, and the hotel details. The receiver uses a method to food get delivered to my home.

##### Responsibilities

In object-oriented programming, behaviors of an object described in terms of responsibilities.

In our example, my request for action indicates only the desired outcome (food delivered to my family). The agent (zomato) free to use any technique that solves my problem. By discussing a problem in terms of responsibilities increases the level of abstraction. This enables more independence between the objects in solving complex problems.

##### Classes and Instances

In object-oriented programming, all objects are instances of a class. The method invoked by an object in response to a message is decided by the class. All the objects of a class use the same method in response to a similar message.

In our example, the zomato a class and all the hotels are sub-classes of it. For every request (message), the class creates an instance of it and uses a suitable method to solve the problem

##### Classes Hierarchies

A graphical representation is often used to illustrate the relationships among the classes (objects) of a community. This graphical representation shows classes listed in a hierarchical tree-like structure. In this more abstract class listed near the top of the tree, and more specific classes in the middle of the tree, individuals listed near the bottom.

##### Method Binding, Overriding, and Exception

In the class hierarchy, both parent and child classes may have the same method which implemented individually. Here, the implementation of the parent is overridden by the child. Or a class may provide multiple definitions to a single method to work with different arguments (overloading).

The search for the method to invoke in response to a request (message) begins with the class of this receiver. If no suitable method is found, the search is performed in the parent class of it. The search continues up the parent class chain until either a suitable method is found or the parent class chain is exhausted. If a suitable method is found, the method is executed. Otherwise, an error message is issued

Summary of Object-Oriented Concepts

Alan Kay, considered by some to be the father of object-oriented programming, identi ed the following characteristics as fundamental to OOP [Kay 1993]:

1. Everything is an object.
2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving mes- sages. A message is a request for action bundled with whatever arguments may be necessary to complete the task.
3. Each object has its own memory, which consists of other objects.
4. Every object is an instance of a class. A class simply represents a grouping of similar objects, such as integers or lists.
5. The class is the repository for behavior associated with an object. That is, all objects that are instances of the same class can perform the same actions.
6. Classes are organized into a singly rooted tree structure, called the inheritance hier- archy. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

**History of java**

In 1990, SunMicrosystems has conceived a project to develop a software for consumer electronic devices that could be controlled by remote. This project was called as Green project .

In January 1991, Bill Joy, James Goslings, Patrick Naughton, Mike sheradin met at Aspen to discuss this project. Mike sheradin was to focus on business Development. Ptarick Naughton was to begin work on graphics system and James Gosling was to identify the proper programming language for the project. Gosling though C and C++ could be used to develop the project. But the problem is faced with them is that they were system dependent languages which could not used an various processors which the electronic devices might use. So he started developing a new language which was completely system independent. This language was initially called *Oak*. Since this name was registered by some other company later it was changed to java.

Why the name java? James Gosling and his team members were consuming a lot of coffee while developing this language. They felt that they were able to develop a better language because of good quality coffee they had consumed. So, the coffee also had its own role in developing this language and hence, they fixed the name for the language as java. Thus the symbol for java is *coffee cup and saucer.*

**Features of Object Oriented Programming System**

Some of the features of OOPS are

1. Object.
2. Class.
3. Encapsulation.
4. Abstraction.
5. Inheritance.
6. Polymorphism.

**Object**

Entire OOP methodology has been derived from a single root concept called object. *An object is anything that really exist in the world and can be distinguished from others*. This definition specifies everything is an object for eg: a table, a ball, a car a dog, a person etc. everything will come under objects. Every object has properties and a can perform certain actions.

For example: let us take a example of dog. It got properties like name, height, color, age etc. these properties are represented by variable. Now the object dog have some actions like running, sleeping, eating, barking etc. these are represented by various methods(functions) in our programming. So, we can conclude that objects contains variable and methods.

**Class**

A group of objects exhibiting same properties and actions will come under the same group called a class. A class represents a group name given to several objects.

For example: lets us take an example “*flower is a class”*. But if we take Rose, Lily, Jasmine, there are all objects of flower class. The class flower does not exist physically but its object like Rose, Lily, Jasmine exist physically.

Hence we could define a *class as a model or blueprint for creating the objects.*

**Abstraction**: There may be a lot of data, a class contains and the user does not need the entire data. The user requires only some part of the available data. In this case we can hide the unnecessary data from the user and expose only that data that is of interest to the user. This is called abstraction.

For example A good example for abstraction is a car, any car will have some parts like engine, radiator, mechanical and electrical equipment etc. the use of the car (driver) should know how to drive the car and does not require any knowledge of these parts. For eg: driver is never bothered about how the engine is designed and the internal parts of the engine.

The advantage of abstraction is that every user will get his own view of the data according to his requirements and will not get confused with unnecessary data.

**Encapsulation**: Encapsulation is a mechanism where the data (variables) and code (methods) that act on the data will bind together. For eg: if we take a class, we write the variables and methods inside the class. Thus, class is binding them together so, class is an example of encapsulation.

**Inheritance :** The process of creating a new class from existing class is called inheritance. There exist a parent-child relationship among the classes. When a class inherits another class, it has all the properties of parent class and it adds some new properties of its own. Inheritance aids to **reusability**

For eg:

1. A good example for inheritance in nature is parents producing the children and children inheriting the qualities of the parents. This concept is important because it supports the concept of hierarchical classification.
2. We categorize vehicles into different subclass as shown below

Vehicle

Auto Bike

Bus

Water Vehicle

Air Vehicle

Road Vehicle

Boat

Aeroplane

**Polymorphism:** the word polymorphism came from two greek words ***“poly”*** means many and **“morphos”** meaning forms. Thus, polymorphism represents the ability to assume several different forms. Polymorphism provides flexibility in writing programs in such a way that the programmer uses same method call to perform different operations depending on the requirement.

**Java Buzz words**

Sun Microsystems described java language as java is a simple, Object-oriented, interpreted, Robust, Secure, architecture Neutral, Portable, high –performance, Multithreaded, Dynamic, Distributed language. These are known as Buzz words

1. **Java is Simple :** Although mush of the syntax of java is based on the earlier object-oriented language C++, the java is considerably simpler than C++.

Java made simple by eliminating some of the complicated concepts from C/C++. Some of them are

1. Many of the keywords have been eliminated.
2. There is no preprocessor.
3. Operator overloading, global variables, the goto statement, structures, pointer are eliminated.
4. **Java is Object-Oriented**: Java is a true object oriented language. Almost everything in java is object oriented. The only unit of programming is the class description. Unlike other languages, java has no functions, and no variables that can exist outside of class boundaries. Thus, all java programs must be built out of objects.
5. **Java is Robust**: Robust means strong. Java programs are strong and they don’t crash easily like a C or C++ program. There are two reasons for this. Firstly, java has got excellent inbuilt Exception Handling features. An Exception is an error that occurs at run-time. If an exception occurs, the program terminates abruptly giving rise to problems like loss of data. Overcoming such problems is called exception handling. This, means that even though an exception occurs in a java program no harm will happen.

Another reason, why java is robust lies in its memory management features. Most of the C and C++ programs crash in the middle because of not allocating sufficient memory or forgetting the memory to be freed in a program. Such problems will not occur in java because the user need not allocate or deallocate the memory. In java everything will be taken care by JVM only. Suppose a variable or an object is created in memory and is not referenced for long time. Then after sometime it is automatically removed by Garbage Collector of JVM by using some algorithm.

1. **Java is Portable:** Java is portable means we can easily move java programs from one system to another system at anytime and from anywhere. Changes and upgrades in OS, Processors and system resources will not force any changes in java programs.

Java ensures portability in **two ways:**

First java compiler generates byte code instructions that can be implemented on any machine. Secondly the size of the primitive data types is machine independent.

1. **Java is Architecture Neural**: one of the main problems facing programmers that no guarantee exists that if you write a program today, it will run tomorrow. Even on the same machine. Operating system upgrades and changes in core system resources can all combine to make program malfunction.

The java designers made several hard decisions in the java language and the JVM in an attempt to alter this situation. Their goal was **“Write once, run anywhere, anytime, forever**”. To a great extent, this goal was accomplished.

1. **Java is secure**: java is secure in the following ways.
2. By **eliminating pointers,** the java language removes the most common source of programming errors, such as overwriting memory locations that are being addressed by pointers with improperly set values.
3. The java language also insists that **array index** values are checked for validity before they are referenced and that all variables must be assigned a value before being used
4. Security problems like eavesdropping, tampering, impersonation and virus threats eliminated or minimized by using java on internet.
5. **Java is multithreaded**: java supports multithreaded programming which allows us to write a program that do many thing simultaneously means we need not wait for the application to complete one task before beginning another.

for eg: we can check our mails and also play audio at the back. This feature gives interactive perform of graphical application.

The java runtime comes with the tools the supports multiprocess synchronization and construct smoothly running interactive system.

1. **Java is interpreted**: Java programs are compiled to generate the Byte code. This code(bye code) can be executed on any system that implements the JVM(Java Virtual machine) or java interpreter. If we take any other language, only a interpreter or a compiler is used to execute the programs. But in java, we use both compiler and interpreter for the execution.
2. **High Performance**: the problem with interpreter inside the JVM is that it is slow. Because of this, java a programs used to run slow. To overcome this problem, along with the interpreter, javasoft people have introduced JIT(Just In Time) complier, which enhance the speed of execution, so now in JVM both interpreter and JIT compiler work together to run the program.
3. **Java is Distributed**: Java is designed for the distributed environment of internet because it handles TCP,IP Protocols. Java also supports RMI(Remote Method Invocation) this feature enables a program to invoke methods across network.
4. **Java is Dynamic**: Before the development of java only static text used to be displayed in the browser. But when James Gosling demonstrated an animated atomic molecule where the rays are moving and stretching, the viewer were dumbstruck. This animation was done using an applet program, which are the dynamically interacting programs on internet.

**Data Types**

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types. These can be put in four groups:

• **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued

signed numbers.

• **Floating-point numbers** This group includes **float** and **double**, which represent

numbers with fractional precision.

• **Characters** This group includes **char**, which represents symbols in a character set, like

letters and numbers.

• **Boolean** This group includes **boolean**, which is a special type

for representing true/false values.

**Integers**

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive

and negative values. Java does not support unsigned, positive-only integers

|  |  |  |
| --- | --- | --- |
| **Name** | **Width** | **Range** |
| long | 64 | –9,223,372,036,854,775,808 **to** 9,223,372,036,854,775,807 |
| int | 32 | –2,147,483,648 **to** 2,147,483,647 |
| short | 16 | –32,768 **to** 32,767 |
| byte | 8 | –128 **to** 127 |

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127.Variables of type **byte** are especially useful when you’re working with a stream of data from a network or file. Byte variables are declared by use of the **byte** keyword.

For example, the following declares two **byte** variables called **b** and **c**:

byte b, c;

**short**

**short** is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are some examples of **short** variable declarations:

short s; short t;

**int**

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range

from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays.

**long**

**long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large

enough to hold the desired value. The range of a **long** is quite large. This makes it useful

when big, whole numbers are needed. For example, here is a program that computes the

number of miles that light will travel in a specified number of days.

**Floating-Point Types**

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

|  |  |  |
| --- | --- | --- |
| **Name** | **Width in Bits** | **Approximate Range** |
| double | 64 | 4.9e–324 to 1.8e+308 |
| float | 32 | 1.4e–045 to 3.4e+038 |

**float**

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don’t require a large degree of precision.

**double**

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double

precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

**Characters**

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human

languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits.

**Booleans**

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

**Variable**

**Variable**:- The name given to computer memory location is called variable. The purpose of variable is to store data.

Syntax:-

**Type identifier[=value][,identifier [=value]…];**

Rules to be followed in naming a variable

1. Variable name may consists of alphabets (A-Z , a-z) ,digits and special symbols $ , \_ are allowed.
2. The first character must not be digit.
3. Variables names are case sensitive.
4. Keywords should not be used as variable names.
5. Variable name can be of any length but, it is better to take length to be reasonable.

There are different types of variables. Some of them are

1. Instance variable.
2. Static variable.
3. Local variable.

**Instance variable**

1. If the value of the variable is varied from one object to another object is called instance variable.
2. For every object a separate copy of instance will be created.
3. Instance variables are created at object creation and destroyed at the time of object destruction.
4. Instance variables should be created within class.
5. JVM supplies default values for instances variables.

class sample

{

int a ,b; // instance variables;

void display()

{

}

-----

-----

}

**Static variables**

1. if the value of variable is not varied from object to object is called static variable.
2. For static variables a single copy will created and shared by all objects of that class.
3. Static variables are declared by means of static modifier.
4. We can access the static members either by class-name (or) object reference. Usage of class-name is recommended.
5. Within the same class it is not required to use class-name , we can access directly.
6. JVM will provide default value for static variable.

**Local variables**

1. The variables which are declared inside a method such type of variables are called local variables.
2. Local variables are also called as stack variables or temporary variables or automatic variables.
3. JVM won’t provide any default value.

class sample

{

void display()

{

int a ,b; // local variables;

}

-----

-----

}

**Arrays**

**Definition**:- An array is an indexed collection of fixed number of homogeneous data elements.

(or)

An array is a group of like-typed variables that are referred to by a common name.

**Arrays offers a convenient means of grouping related information.**

Advantages

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

Disadvantages

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

**Types of arrays**

Arrays are generally categorized into two parts

1. One-Dimensional arrays( or 1D arrays)
2. multi dimensional arrays(or 2D, 3D.. arrays)

In java obtaining an array is a two step process.

1. We must declare a variable of desired array type.(**Array Declaration**)
2. We must allocate the memory that will hold the array using new, and assign it to the array variable(**Array Construction**).

**Array Declaration**

***Syntax:* type var-name[];**

Here type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus the base type for the array determines what type of data the array will hold.

For example

int arr[ ];

🡪 this declaration tells that **arr** is an array variable, no array actually exists. In fact value of **arr**  is set to **null**(not existing/absence of value), which represents an array with no values.

**Array Construction**

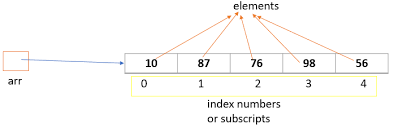
Every array is an object hence we can construct by using **new**  operator. For every array type internally the corresponding classes are variable but these classes are not available to the programmer level.

General form of array construction using new operator

***array-var=new type[size];***

Here type specifies the type of data being allocated. Size specifies the number of elements in the array and ***array-var*** is the array variable that is linked to the array. The elements in the array allocated by new will automatically be initialized to **zero.**

arr=new int[5];



We combine array declaration and array construction in single statement like

**type var-name[]=new type[size];**

we know that in array the first element is stored at 0th index and last element is stored at (n-1)th  index. Where ‘n’ is the size of the array.

**Some important points.**

1. It is legal to have array with zero size i.e int a[]=new int[0];
2. The allowed data types for array size are byte, short, char, int.

int [] a=new int[‘a’]; //valid becoz ASCII code of ‘a’ is equal 97

int [] a=new int[10.5]; //invalid

int [] a=new int[10L]; //invalid

1. The java runtime system will check to be sure that all array index are in the correct range. If we try to access elements outside the range of the array (negative numbers or numbers greater than length of the array).it will cause run-time error.
2. As arrays are internally objects , it is having a property **length** which tells the size of the array.

For example int [] a=new int[6];

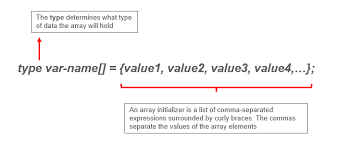
System.out.println(a.length) // output🡪 6

1. Sometimes we can declare array without name. Such type of arrays are called **anonymous arrays**. The main purpose of anonymous arrays is just for instant use. We can create anonymous arrays as follows

**new int[] {1,2,3,4,5};**

**Array Declaration**, **Construction and initialization in a single line**

Arrays can be initialized when they are declared. An array intializer is a list of comma-separated expressions surrounded by curly braces. The comma separates the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array intializer.



**For example 🡪 int [ ] a={1,2,3,4,5};**

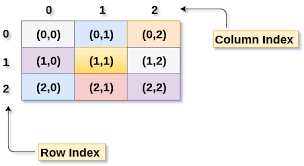
**Multidimensional Arrays**

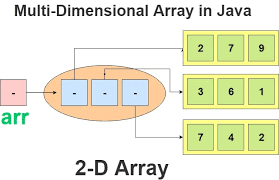
In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

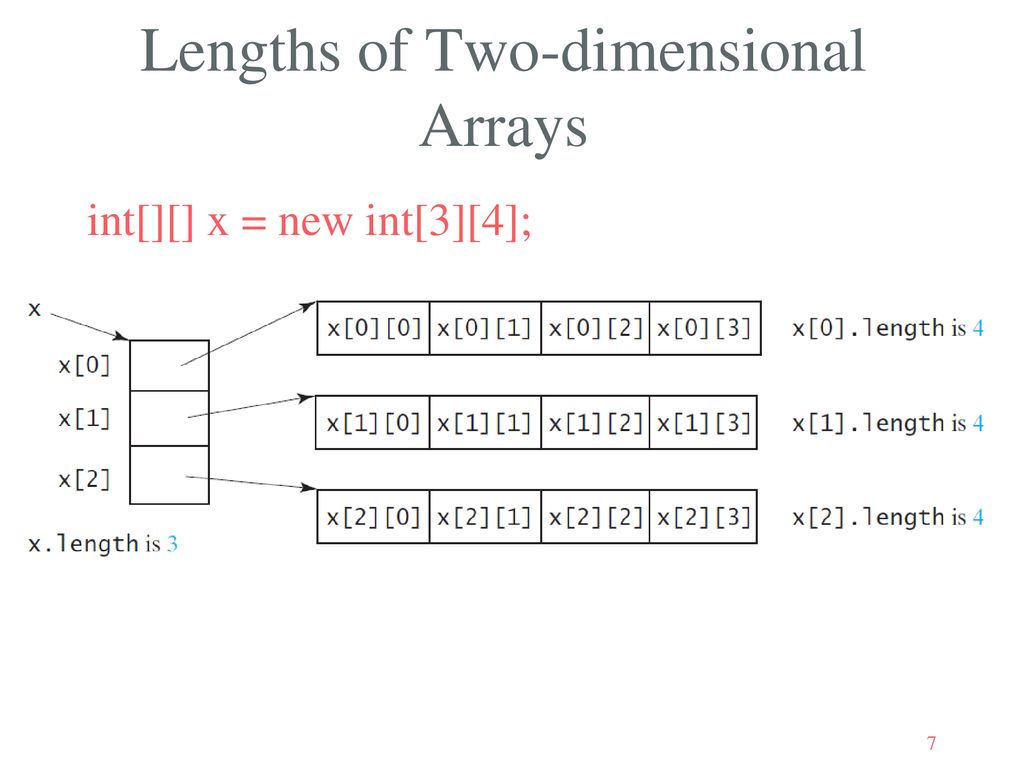
For example, the following declares a two dimensional array variable called arr.

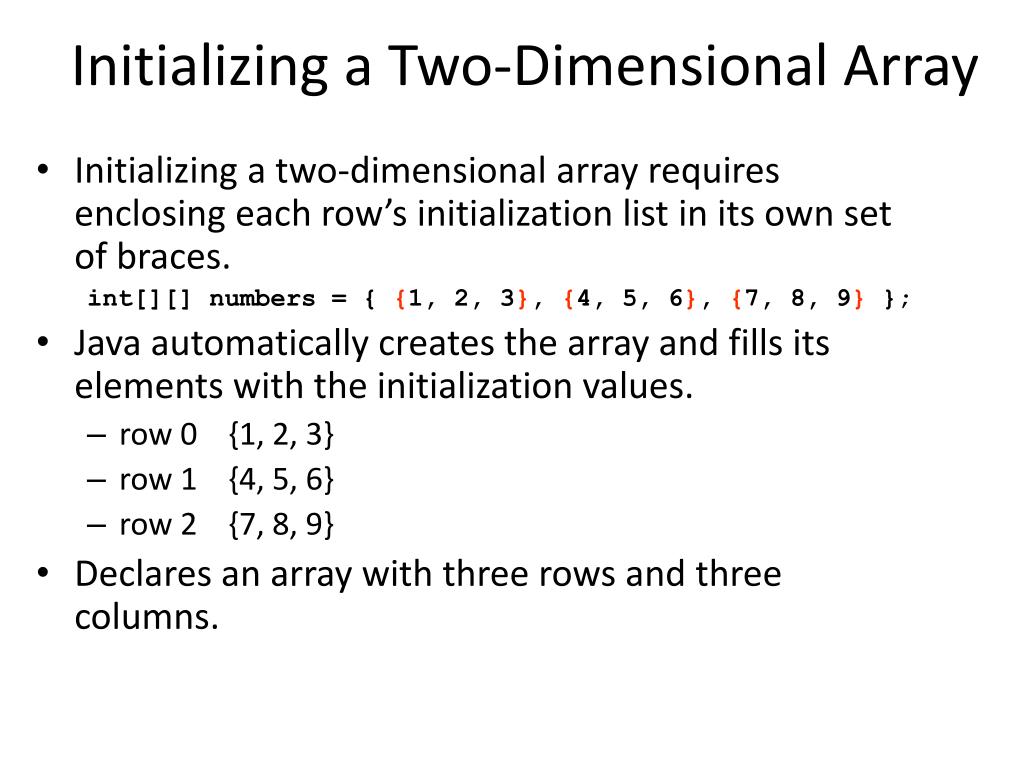
int arr[][] = new int[3][3];

This allocates a 3 by 3 array and assigns it to **arr** Internally this matrix is implemented as an *array* of *arrays* of **int**.



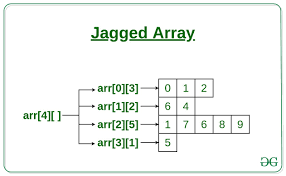




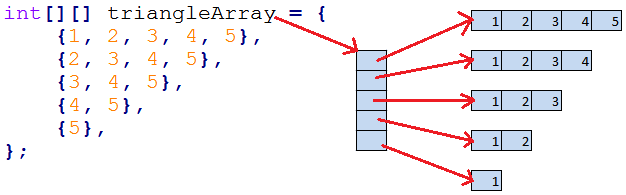


**Ragged arrays**

 Ragged array is **an array of arrays such that member arrays can be of different sizes**, i.e., we can create a 2-D array but with a variable number of columns in each row. These types of arrays are also known as Jagged arrays.



|  |
| --- |
| int[][] arr = new int[4][];  // this will initialize the array elements by 0  arr[0] = new int[3];  arr[1] = new int[2];  arr[2] = new int[5];  arr[3] = new int[1]; |

**Initialization the 2D array**

**Programs on Arrays**

* 1. WAP to demonstrate 1D array.
  2. WAP to demonstrate 2D arrays
  3. WAP to display the following output(Ragged array)

1

2 3

4 5 6

7 8 9 10

4. WAP to display the following output

1 0 0

0 1 0

0 0 1

5. WAP to display the following output

1 2 2

3 1 2

3 3 1

1. **Solution**

import java.util.\*;

class OneArray

{

public static void main(String args[])

{

int []a=new int[5];

Scanner s=new Scanner(System.in);

System.out.println("Enter 5 array elelments");

for(int i=0;i<a.length;i++)

a[i]=s.nextInt();

System.out.println("Entered Array elements are");

for(int i=0;i<a.length;i++)

System.out.print(a[i]+" ");}}

1. **Solution**

import java.util.\*;

class TwoArray

{

public static void main(String args[])

{

int [][]a=new int[3][3];

Scanner s=new Scanner(System.in);

System.out.println("Enter 9 array elelments");

for(int i=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++)

a[i][j]=s.nextInt();

}

System.out.println("Entered Array elements are");

for(int i=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++)

System.out.print(a[i][j]+" ");

System.out.println();

}}}

1. **Solution**

import java.util.\*;

class RaggedArray

{

public static void main(String args[])

{

int [][]a=new int[4][];

a[0]=new int[1];

a[1]=new int[2];

a[2]=new int[3];

a[3]=new int[4];

Scanner s=new Scanner(System.in);

System.out.println("Enter 9 array elelments");

for(int i=0,k=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++,k++)

a[i][j]=k;

}

System.out.println("Entered Array elements are");

for(int i=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++)

System.out.print(a[i][j]+" ");

System.out.println();

}}}

1. **Solution**

import java.util.\*;

class Test

{

public static void main(String args[])

{

int [][]a=new int[3][3];

for(int i=0,k=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++,k++)

{

if(i==j)

a[i][j]=1;

else

a[i][j]=0;

}

}

for(int i=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++)

System.out.print(a[i][j]+" ");

System.out.println();

}}}

1. **Solution**

import java.util.\*;

class Test1

{

public static void main(String args[])

{

int [][]a=new int[3][3];

for(int i=0,k=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++,k++)

{

if(i==j)

a[i][j]=1;

else if(i<j)

a[i][j]=2;

else

a[i][j]=3;

}}

for(int i=0;i<a.length;i++)

{

for(int j=0;j<a[i].length;j++)

System.out.print(a[i][j]+" ");

System.out.println();

}}}

**Operators**

**Operator:-**  Operator is a symbol that tells the compiler to perform mathematical operation and returns the resultant value.

Java provides a rich set of operator. Most of its operators are divided into the following four groups:

1. Arithmetic Operators.
2. Bitwise Operators.
3. Relational Operators, and
4. Logical Operators.

**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:

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| + | Addition | 6+5=11 |
| – | Subtraction (also unary minus) | 6-5=1 |
| \* | Multiplication | 6\*5=30 |
| / | Division | 6/5=1 |
| % | Modulus | 6%5=1 |
| ++ | Increment |  |
| += | Addition assignment | a+-=1 🡪a=a+1 |
| –= | Subtraction assignment | a-=1 🡪a=a-1 |
| \*= | Multiplication assignment | a\*=1 🡪a=a\*1 |
| /= | Division assignment | a/=1 🡪a=a/1 |
| %= | Modulus assignment | a%=1 🡪a=a%1 |
| – – | Decrement |  |

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

**The Bitwise Operators**

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| ~ | Bitwise unary NOT or Bitwise one’s complement operator | ~(4) 🡪 5 |
| & | Bitwise AND | 42 & 15 🡪10 |
| | | Bitwise OR | 42 | 15 🡪 47 |
| ^ | Bitwise exclusive OR | 42 ^ 15 🡪 37 |
| >> | Shift right | 42 >> 2 🡪10 |
| >>> | Shift right zero fill | -1 >>> 24 🡪 255 |
| << | Shift left | 42<<2 🡪168 |
| &= | Bitwise AND assignment |  |
| |= | Bitwise OR assignment |  |
| ^= | Bitwise exclusive OR assignment |  |
| >>= | Shift right assignment |  |
| >>>= | Shift right zero fill assignment |  |
| <<= | Shift left assignment |  |

**The Bitwise Logical Operators**

The bitwise logical operators are **&**, **|**, **^**, and **~**. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **A | B** | **A & B** | **A ^ B** | **~A** |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

**Relational Operators**

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Result** | **Example** |
| == | Equal to | 5 == 5 🡪 True |
| != | Not equal to | 5 != 6 🡪True |
| > | Greater than | 5 > 6 🡪False |
| < | Less than | 5 < 6 🡪True |
| >= | Greater than or equal to | 6 > = 5🡪 True |
| <= | Less than or equal to | 5 < =6 🡪 True |

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

**Boolean Logical Operators**

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

|  |  |
| --- | --- |
| **Operator** | **Result** |
| & | Logical AND |
| | | Logical OR |
| ^ | Logical XOR (exclusive OR) |
| || | Short-circuit OR |
| && | Short-circuit AND |
| ! | Logical unary NOT |
| &= | AND assignment |
| |= | OR assignment |
| ^= | XOR assignment |
| == | Equal to |
| != | Not equal to |
| ?: | Ternary if-then-else |

The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true = false** and **!false = true**. The following table shows the effect of each logical operation:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **A | B** | **A & B** | **A^ B** | **!A** |
| False | False | False | False | False | True |
| True | False | True | False | True | False |
| False | True | True | False | True | True |
| True | True | True | True | False | False |

**The Assignment Operator**

You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, **=**. The assignment operator works in Java much as it does in any other computer language. It has this general form:

*var = expression*;

Here, the type of *var* must be compatible with the type of *expression.* The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

int x, y, z;

x = y = z = 100; // set x, y, and z to 100

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement

**The ? Operator**

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the **?** and it works in java much like it does in C and C++.

The **?:** has this general form:

***expression1* ? *expression2* : *expression3***

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can’t be **void**.

**Expressions:**

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.

int **a = 0**;

arr[0] = 100**;**

System.out.println**(**"Element 1 at index 0: " + arr[0])**;**

int result **=** 1 + 2**;** // result is now 3

if(value1 == value2**)**

System.out.println**(**"value1 == value2")**;**

The data type of the value returned by an expression depends on the elements used in the expression. The expression a= 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, cadence is an int. As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String.

For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

x + y / 100 // ambiguous

You can specify exactly how an expression will be evaluated using balanced parenthesis rewrite the expression as

(x + y) / 100 // unambiguous, recommended

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

x + y / 100

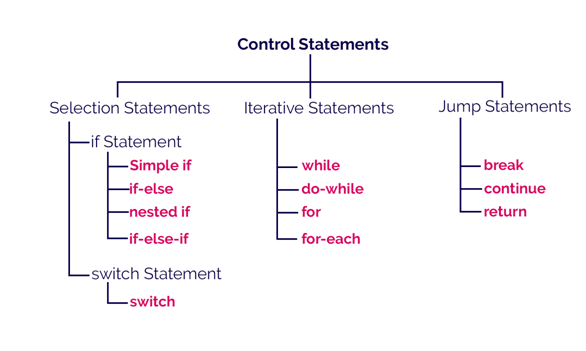
x + (y / 100) // unambiguous, recommended

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first.

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](https://www.javatpoint.com/java-tutorial) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.



Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

**if**(condition) {

statement 1; //executes when condition is true

}

### 2) if-else statement

The [if-else statement](https://www.javatpoint.com/java-if-else) is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

**Syntax:**

**if**(condition) {

statement 1; //executes when condition is true

}

**else**{

statement 2; //executes when condition is false

}

### 3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

**syntax of if-else-if statement is given below.**

**if**(condition 1) {

statement 1; //executes when condition 1 is true

}

**else** **if**(condition 2) {

statement 2; //executes when condition 2 is true

}

**else** {

statement 2; //executes when all the conditions are false

}

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

**Syntax of Nested if-statement is given below.**

**if**(condition 1) {

statement 1; //executes when condition 1 is true

**if**(condition 2) {

statement 2; //executes when condition 2 is true

}

**else**{

statement 2; //executes when condition 2 is false

}

}

### Switch Statement:

In Java, [Switch statements](https://www.javatpoint.com/java-switch) are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

* The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
* Cases cannot be duplicate
* Default statement is executed when any of the case doesn't match the value of expression. It is optional.
* Break statement terminates the switch block when the condition is satisfied.  
  It is optional, if not used, next case is executed.
* While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

**The syntax to use the switch statement is given below.**

**switch** (expression){

**case** value1:

     statement1;

**break**;

    .

    .

    .

**case** valueN:

     statementN;

**break**;

**default**:

**default** statement;

}

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

**Looping/Iterative/Repetitive Control statement**

In looping control instruction, a sequence of statements are executed repeatedly until some condition for termination of loop are satisfied.

Java language supports four different types of looping statements. They are

1. While Loop
2. Do-While Loop
3. For Loop
4. For-each Loop (used for arrays and Collections)

**The While Statement**

The while statement is used to carry out looping operations.

The General Form of while statement is

**while (condition)**

**{**

**//body of loop**

**}**

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

**The do-While Statement**

When a loop is constructed using the while statement, the test for continuation of the loop is carried out at the beginning of each pass. Sometimes , however ,it is desirable to have a loop with the test for continuation at the end of each pass. This can be accomplished by means of the **do-while** statement. The general form of the Do-While statement is

**do**

**{**

**// body of loop**

**} while(condition);**

The included statements in the loop will be executed repeatedly, as long as the value of conditional expression true. Here the statements will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The statements can be either simple or compound, though most applications will require it to be a compound statement. When condition becomes false, control passes to the next line of code immediately following the loop.

For most applications it is more natural to test for continuation of a loop at the beginning rather than at the end of the loop. For this reason, the Do-while statement is used less frequently than the while statement.

**The For statement**

The *for statement* is the third and perhaps the most commonly used looping statement in java language. This statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued and a third expression that allows the index to the modifies at the end of each pass.

The general form of for statement is

**for(initialization; condition; increment/decrement)**

**{**

**//body of loop**

**}**

The execution of the *for* statement is as follows:

1. Initialization of the control variable is done first, using assignment statement such as i=1 or count=0. The variable i, count are known as loop-control variables.
2. The value of the control variable is tested using the condition. The condition is a relational expression , such as i<10 that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, control is transferred back to the *for* statement after evaluating the last statement in the loop. Now the control variable is incremented using an assignment statement such as i=i+1 and new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

**Differences between while and do-while**

In while loop first the condition is checked whether true or false, if the condition is true then the control enters inside the block and executes the statements. Because the testing of the condition is done at the beginning it is also called as *pretest loop. In case of do-while loop the testing of the condition is done at the end after completing the execution of the statements. Hence it is called as post test loop.*

**Nested loops**

Loops can be nested (i.e., embedded) one within another. The inner and outer loops need not be generated by the same type of control structure. It is essential, however that one loop be completely embedded within the other-there can be no overlap. Also, each loop must be controlled by different index.

examples

1. while(condition-1)

{

Statements;

while(condition-2)

{

Statements;

}

}

2. for(initialization; condition-1; increment/decrement)

{

Statements;

for(initialization; condition-2; increment/decrement)

{

Statements;

}}

**Jump Statements**

java language permits a jump from one statement to another within a loop as well as a jump out of a loop. The following are the jump statements.

1. Break.
2. Break with a label.
3. Continue.
4. Continue with a label.
5. Return.

**Break statement**

The break statement is used to terminate loops or to exit from a switch. It can be used within a *while, do-while, for* or a *switch* statement.

The break statement is written simply as

*break;*

without any embedded expressions or statements.

When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop. Usually we will use the break statement with if statement.

**Break with label**

Break with a label is used in the case where we can force the control to come out from named block.

The labeled break statement is written simply as

*Break label;*

Here , label is the name of a label that identifies a block of code. When this form of break executes, control is transferred out of the named block of code. The labeled block of code must enclose the break statement, but it does not need to be the immediate enclosing block. This means that you can use a labeled break statement to exit from a set of nested blocks.

To name a block, put a label at the start of it. A label is nay valid java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a break statement.

**/\* Demo program on Break with Label\*/**

class sample1

{

public static void main(String args[])

{

L1:

for(int i=0;i<3;i++)

{

for(int j=0;j<3;j++)

{

if(i==j)

break L1;

System.out.println(i+"-----"+j);

}}}}

**Continue statement**

The continue statement is used to skip the current iteration i.e, on executing continue statement the continue is transferred to the beginning of the loop. In case of while or do-while the control is transferred to the conditional part. Whereas in for loop the control is transferred to the increment (or) decrement part. Usually we will use the continue statement with if statement.

The continue statement is written simply as

***continue;***

**continue with label :**

this is similar to labeled break but instead of making the control to come out from loop , the control is transferred to labeled loop by skipping the current iteration and proceeding with rest of the iterations in the loop.

**Return Statement**

The last control statement is *return*. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus the return statement immediately terminates the method in which it is executed.

**/\* This is a simple Java program. Call this file "Example.java".\*/**class Example {  
// Your program begins with a call to main().  
public static void main(String args[]) {  
System.out.println("This is a simple Java program.");  
}}

**Introduction to classes**

The class is at the core of java. It is the logical construct upon which entire java language is built. The importance of class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus class is a template from an object, and an object is an instance of a class.

**General Form of class declaration**

class classname

{

type instance\_variable1;

type instance\_variable 2;

……………

type instance\_variable n;

type methodname\_1(parameter list)

{

// body of the method

}

type methodname\_2(parameter list)

{

// body of the method

}

type methodname\_n(parameter list)

{

// body of the method

}

}

The data or variables defined within a class are called instance variables. The code is contained within the method. Collectively, the methods and variables defined within a class are called members of the class. Variables defined within a class are called instance variables because each instance of the class (that is each object of the class) contains its own copy of these variables. Thus the data for one object is separate and unique from the data for other.

**Declaring Object**

When we create a class, you are creating a new data type. We can use this type to declare objects of that type. Creating objects of a class is a two-step process

1. We must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.

***classname class\_variable;***

1. We must acquire an actual physical copy of the object and assign it to the variable. We can do this using the new operator. The new operator dynamically allocates memory for an object and returns a reference to it. This reference is more or less, the address in memory of the object allocated by new. This reference is then stored in the variable.

***class\_variable=new classname();***

Here, class\_var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parenthesis specifies the constructor for the class. Two steps can be combined into one steps i.e.

**classname class\_variable=new classname();**

**Introducing methods**

Classes usually consists of two things: instance variables and methods.

***General form of defining method***

*type name*(*parameter*-*list*)

{

// body of method

}

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name.* This can be any legal identifier other than those already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement;

**return *value*;** Here, *value specifies the va*lue to be returned to the calling function

**Method Overloading**

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java’s most exciting and useful features.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

While overloading method resolution, if the compiler unable to find the method with required argument, it won’t raise compile-time error immediately. First compiler promotes the argument to the next level and checks whether matched method is available or not. If it is not available then compiler once again promotes arguments to the next level and checks for the matched method. This process will be continued for all possible promotion. Still if the compiler unable to find the matched method then only it will raise compile-time error.

The following is the list of all possible automatic promotions

Byte Short

int🡪long🡪float🡪double

Char

*Automatic conversion is followed by compiler*

Byte🡪 short 🡪 int🡪 long🡪 float-🡪 double

char

**Program1:**

**Exact match between function call and function Definition**

class OverLoad1

{

public void m1(int a)

{

System.out.println("int-arg");

}

public void m1(long a)

{

System.out.println("long-arg");

}

public void m1(char a)

{

System.out.println("char-arg");

}

public static void main(String args[])

{

OverLoad1 l1=new OverLoad1();

l1.m1(10);

l1.m1(10l);

l1.m1('a');

}

}

**Program 2:**

**Automatic conversion takes place between function call and function definition**

class OverLoad2

{

public void m1(int a)

{

System.out.println("int-arg");

}

public void m1(float a)

{

System.out.println("float-arg");

}

public static void main(String args[])

{

OverLoad2 l1=new OverLoad2();

l1.m1('a');

l1.m1(10l);

l1.m1(3.5);

}

}

**varible length arguments**

JKDk 1.5 adds a feature that simplifies the creation of methods that need to take a variable number of arguments.

A methods that takes a variable number of arguments is called ***variable-arity*** method or simply a ***varargs method***

**Program 3:**

class VargsMethod

{

static void m1(int ...a)

{

System.out.println("vargs int contain"+ a.length+"Arguments");

for(int x:a)

System.out.println(x+" ");

}

static void m1(boolean ... b)

{

System.out.println("vargs boolean contain"+ b.length+"Arguments");

for(boolean a:b)

System.out.println(a+" ");

}

static void m1(String s,int ...a)

{

System.out.println("vargs String and int contain "+ s+" and "+a.length+ "Arguments");

for(int x:a)

System.out.println(x+" ");

}

public static void main(String args[])

{

m1(10,20,30);

m1(true,false,true,false);

m1("abc",1,2,3,4);

}

}

**Program4: Vargs method**

class OverLoad3

{

public void m1(int a)

{

System.out.println("int-arg");

}

public void m1(int ...a)

{

System.out.println("int ... arg");

}

public static void main(String args[])

{

OverLoad3 l1=new OverLoad3();

l1.m1();

l1.m1(10);

l1.m1(10,20,30);

}

}

**Constructors**

A constructor is similar to a method that is used to initialize the instance variables. The sole purpose of a constructor is to initialize the instance variables.

**Characteristics of constructors**

1. The constructor’s name and class name should be same.
2. A constructor may have or may not have parameters. Parameters are variables to receive data from outside into the constructor. If a constructor does not have any parameters it is called default constructor. If a constructor has 1 or more parameters, it is called parameterized constructor.
3. A constructor does not return any value, not even void.
4. A constructor is automatically called and executed at the time of creating an object. While creating an object, if nothing is passed to the object the default constructor is called and executed. If some value are passed to the object, then the parameterized constructor is called.

**Example** Person p1 =new Person(); //default constructor

Person p2=new Person(“a,”,22); //parameterized constructor

1. A constructor is called and executed only once per object. This means when we create an object the constructor is called. When we create second object, Again the constructor is called second time.
2. Like method overloading we can overload the constructors.

**Difference between default constructor and parameterized constructor.**

|  |  |
| --- | --- |
| **Default constructor** | **Parameterized constructor** |
| Default constructor is useful to initialize all objects with same data. | Parameterized constructor is useful to initialize each object with different data. |
| Default constructor does not have any parameters. | Parameterized constructor will have 1 or more parameters. |
| When data is not passed at the time of creating an object. Default constructor is called. | When data is passed at the time of creating an object, parameterized constructor is called. |

**The this Keyword**

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class’ type is permitted. To better understand what this refers to, consider the following version of Box( ):

// A redundant use of this.

Box(double w, double h, double d) {

this.width = w;

this.height = h;

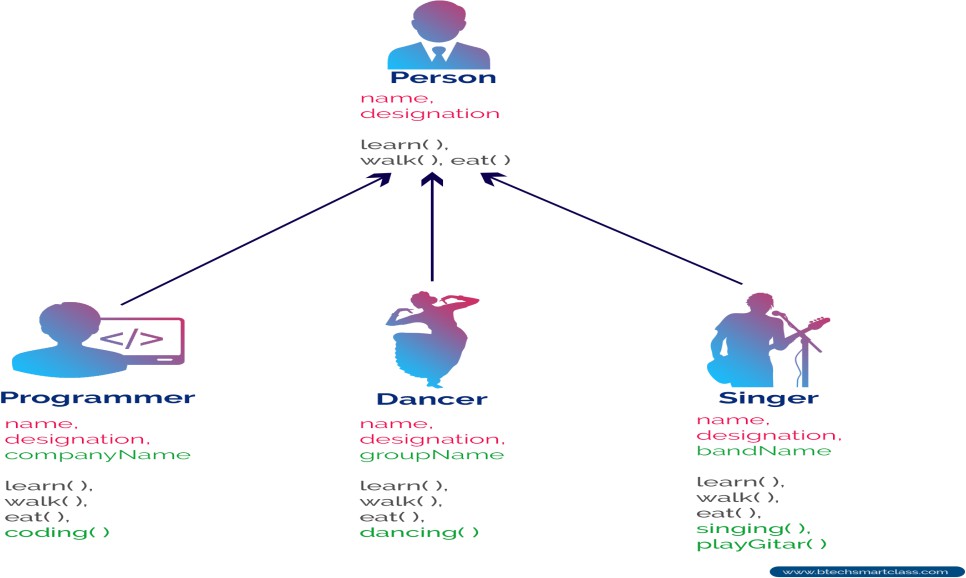
this.depth = d;

}

The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object.

#### Inheritance concept

* The inheritance is a very useful and powerful concept of object-oriented programming. In java, using the inheritance concept, we can use the existing features of one class in another class.
* The inheritance provides a great advantage called code re-usability.
* With the help of code re-usability, the commonly used code in an application need not be written again and again.



The inheritance can be defined as follows.

**The inheritance is the process of acquiring the properties of one class to another class.**

**Inheritance Basics**

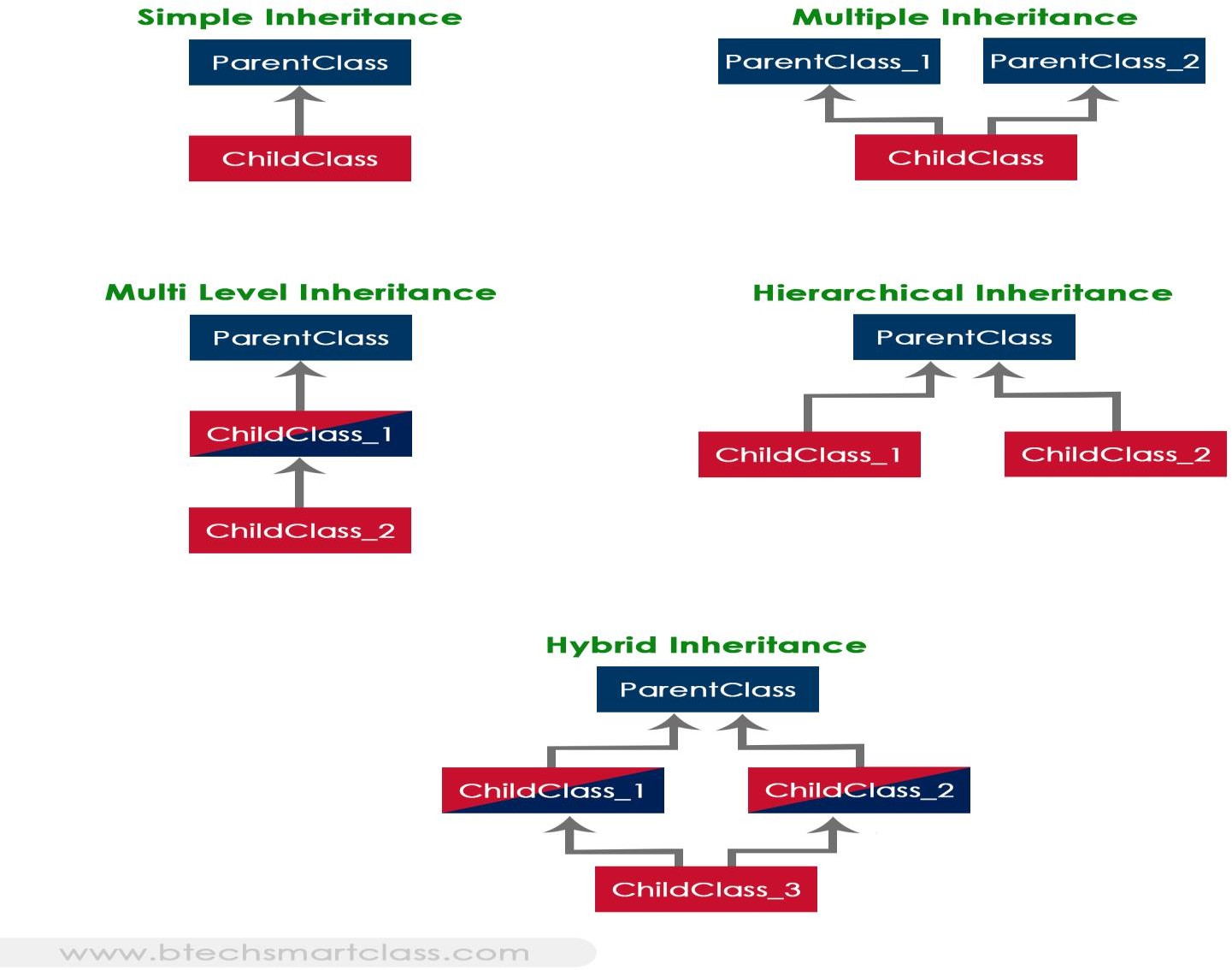
* In inheritance, we use the terms like parent class, child class, base class, derived class, superclass, and subclass.
* The **Parent class** is the class which provides features to another class. The parent class is also known as **Base class** or **Superclass**.
* The **Child class** is the class which receives features from another class. The child class is also known as the **Derived Class** or **Subclass**.

In the inheritance, the child class acquires the features from its parent class. But the parent class never acquires the features from its child class.

There are five types of inheritances, and they are as follows.

* **Simple Inheritance (or) Single Inheritance**
* **Multiple Inheritance**
* **Multi-Level Inheritance**
* **Hierarchical Inheritance**
* **Hybrid Inheritance**

The following picture illustrates how various inheritances are implemented.



The java programming language does not support multiple inheritance type. However, it provides an alternate with the concept of interfaces.

Creating Child Class in java

In java, we use the keyword **extends** to create a child class. The following syntax used to create a child class in java.

Syntax

class <ChildClassName> extends <ParentClassName>{

...

//Implementation of child class

... }

**Example program1**

class Person

{

int id;

String name;

int age;

void getdata(int id,String name,int age)

{

this.id=id;

this.name=name;

this.age=age;

}

void display()

{

System.out.println("My id is"+id);

System.out.println("My name is"+name);

System.out.println("My age is"+age);

}

}

class Student extends Person

{

int m1,m2,m3;

void getmarks(int m1,int m2,int m3)

{

this.m1=m1;

this.m2=m2;

this.m3=m3;

}

void putmarks()

{

System.out.println("My marks are "+m1+" "+m2+" "+m3);

}

}

class employee extends Person

{

float salary;

void getsal(float salary)

{

this.salary=salary;

}

void putsal()

{

System.out.println("My salary is "+salary);

}

}

class Test2

{

public static void main(String args[])

{

Student s=new Student();

s.getdata(1,"abc",20);

s.display();

s.getmarks(10,20,30);

s.putmarks();

employee s1=new employee();

s1.getdata(100,"xyz",40);

s1.getsal(10000);

s1.display();

s1.putsal();

}

}

**Program 2**

class A

{

int i,j;

void showij()

{

System.out.println("i="+i+"j="+j);

}

}

class B extends A

{

int k;

void showk()

{

System.out.println("k="+k);

}

void sum()

{

System.out.println("The sum of i,j, and k is "+(i+j+k));

}

}

class Test1

{

public static void main(String args[])

{

B b1=new B();

b1.i=10;

b1.j=20;

b1.k=30;

b1.showij();

b1.showk();

b1.sum();

}

}

**Constructors in inheritance**

in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass’ constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default or parameterless constructor of each superclass will be executed

class A

{

A()

{

System.out.println(" A class Constructor");

}

}

class B extends A

{

B()

{

System.out.println(" B class Constructor");

}

}

class C extends B

{

C()

{

System.out.println(" C class Constructor");

}

}

class D

{

public static void main(String args[])

{

C c1=new C();

}

}

**Method overriding:**

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

**// demo program on Method overriding.**

class A

{

int i, j;

A(int a, int b)

{

i = a; j = b;

}

// display i and j

void show()

{

System.out.println("i and j: " + i + " " + j)

}

}

class B extends A

{

int k;

B(int a, int b, int c)

{

super(a, b); k = c;

}

// display k – this overrides show() in A

void show()

{

System.out.println("k: " + k);

} }

class Override

{

public static void main(String args[])

{

B subOb = new B(1, 2, 3);

subOb.show(); // this calls show() in B

}

**Difference between Method overloading and overriding.**

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| 1. Writing two or more methods with the same name but with different signatures is called Method overloading | Writing two or more methods with the same name and same type signatures is called Method overriding. |
| 1. Method Overloading is done in same class | Method overriding is done in super and sub classes. |
| 1. In Method Overloading method return type can be same or different | In Method overriding method return types should be same. |
| 1. JVM decides which method is called depending on the differences in the method signatures | JVM decides which method is called depending on the data types(class) of the object used to call the method. |
| 1. Method Overloading is **code refinement**. Same method is refined to perform a different task. | Method overriding is **code replacement**. The sub class method overrides(replaces) the super class method. |

**Garbage Collection**

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach it handles deallocation for you automatically. The technique that accomplishes this is called ***garbage collection****.* It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

**The finalize( ) Method**

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.  
The **finalize( )** method has this general form:

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

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

**Introducing Access Control**

Encapsulation provides another important attribute: *access control.* Through encapsulation,  
you can control what parts of a program can access the members of a class. By controlling   
access, you can prevent misuse. How a member can be accessed is determined by the *access specifier* that modifies its declaration.

**Access Specifiers**

An access modifier is a keyword that specifies how to access the members of a class or a class itself. We can use access modifier before a class and its members. There are four access modifier available in java.

* 1. **Private :-** private members of a class are not accessible any where outside the class. They are accessible only within the class by the methods of that class.
  2. **Public:**  public members of class are accessible every where outside the class. So any other program can read them and use them.
  3. **Protected:** protected members of a class are accessible only to classes in same package but outside the class, but only to classes that subclass your class directly.
  4. **Default:** if no access modifier is written by the programmer then the java compiler uses a default access modifier. Default members are accessible outside the class, but within the same package(directory).

