# Unit 1 Introduction to Object oriented programming

## UNIT-I

***For Micro Notes by the Student***

**Introduction to Object Oriented Programming:**

The C language is structured, efficient and high level language. Whenever a new programming language is designed some trade-offs are made, such as ease-of-use verses power

- Safety verses efficiency
- Rigidity versus extensibility
- Prior to the c language, there are many languages:

  FORTRAN, which is efficient for scientific applications, but is not good for system code.

  BASIC, which is easy to learn, but is not powerful, it is lack of structure.
- Assembly Language, which is highly efficient, but is not easy to learn.
- These languages are designed to work with GOTO statement.

As a result programs written using these languages tend to produce "spaghetti code", which is impossible to understand. Dennis Ritchie, during his implementation of UNIX operating system, he developed C language, which similar to an older language called BCPL, which is developed by Martin Richards.

The BCPL was influenced by a language called B, invented by Ken Thomson. C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted. During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is complexity. The program complexity is increasing it demanded the better way to manage the complexity. When computers were first programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The 1960s gave birth to structured programming. This is the method of programming championed by languages such as C. The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming

***For Micro Notes by the Student***

methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. To solve this problem, a new way to program was invented, called object-oriented programming (OOP).
OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.

**The Three OOP Principles:**
All object-oriented programming languages provide mechanisms that help you implement the object-oriented model.
They are encapsulation, inheritance, and polymorphism.

**Encapsulation** is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

In Java, the basis of encapsulation is the class. A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class. The objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods. The members can be public or private. When a member is made public any code outside the class can access them. If the members are declared, then only the members of that class can access its members.

**Inheritance:** Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. The Class from which the properties are acquired is called

*For Micro Notes by the Student*

super class, and the class that acquires the properties is called subclass. This is mainly used for Method overloading and Code reusability.

**Polymorphism:** Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action (that is, method) as it applies to each situation.

**3. Procedural language Vs OOP**

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around "what is happening" and others are written around "who is being affected." These are the two paradigms that govern how a program is constructed. The first way is called the process oriented model or procedural language. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. To manage increasing complexity, the second approach, called object-oriented programming, was designed. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

**4. Applications of OOP:**

There are mainly 4 types of applications that can be created using java programming:

1) **Standalone Application:** It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.
2) **Web Application:** An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, etc. technologies are used for creating web applications in java.
3) **Enterprise Application:** An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security,

load balancing and clustering. In java, EJB is used for creating enterprise applications.

**4) Mobile Application:** An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

***For Micro Notes by the Student***

**5. History of JAVA**

1. Brief history of Java
2. Java Version History Java history is interesting to know.

The history of java starts from Green Team. Java team members (also known as Green Team), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc. For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape. Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major point that describes the history of java.

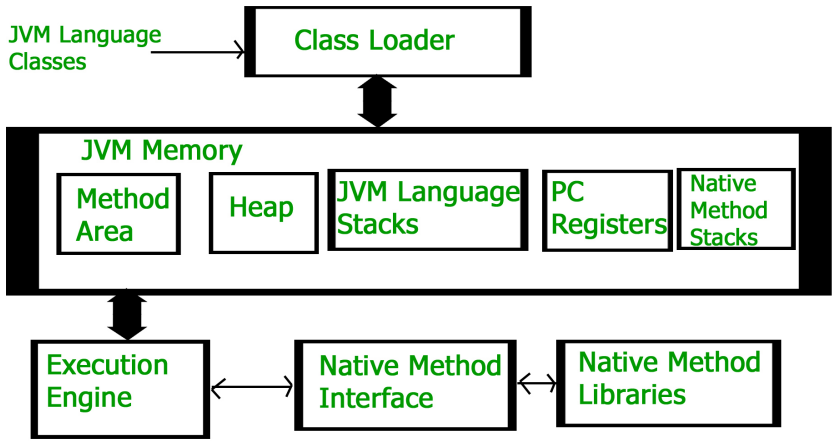
1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.
3) Firstly, it was called "Greentalk" by James Gosling and file extension was .gt. 4) After that, it was called Oak and was developed as a part of the Green project. Why Oak name for java language?
5) Why Oak? Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc. During 1991 to 1995 many people around the world contributed to the growth of the Oak, by adding the new features. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the original prototype.
6) In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies. Why Java name for java language?
7) Java is an island of Indonesia where first coffee was produced (called java coffee).

Java Version History There are many java versions that has been released. Current stable release of Java is Java SE 8.

*For Micro Notes by the Student*

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

**6. Java Virtual Machine:**




**7. Features of Java:**

There is given many features of java. They are also known as **java buzzwords.**

The Java Features given below are simple and easy to understand.

1. Simple
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multithreaded
7. Architecture-neutral
8. Interpreted
9. High performance
10. Distributed

11. Dynamic

1. **Simple** Java was designed to be easy for the professional programmer to learn and use effectively. According to Sun, Java language is simple because: syntax is based on C++ (so easier for programmers to learn it after C++). Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. No need to remove unreferenced objects because there is Automatic Garbage Collection in java.
2. **Secure** Once the byte code generated, the code can be transmitted to other computer in the world with knowing the internal details of the source code.
3. **Portable** The byte code can be easily carried from one machine to other machine.
4. **Object Oriented** Everything in Java is an Object. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.
5. **Robust** The multi-plat-formed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. Java also frees from having worry about many errors. Java is Robust in terms of memory management and mishandled exceptions. Java provides automatic memory management and also provides well defined exception handling mechanism.
6. **Multithreaded** Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.
7. **Architecture-neutral** The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished. Interpreted and
8. **High Performance** Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java byte code was carefully designed so

***For Micro Notes by the Student***

*For Micro Notes by the Student*

that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

9. **Distributed** Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.
10. **Dynamic** Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

**Java Program Structures:**

Simple Java Program

Example.java

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

- Entering the Program We can use any text editor such as "notepad' or "dos text editor".
- The source code is typed and is saved with ".java" as extension.
- The source code contains one or more class definitions.
- The program name will be same as class name in which main function is written.

**Compiling the Program** To compile the program, first execute the compiler, "javac", specifying the name of the source file on the command line, as shown below:

**c:\>javac Example.java**

The javac compiler creates the file called "Example.class", which contains the byte code version of the source code. This byte code is the intermediate representation of the source code that contains the instructions that the Java Virtual Machine (JVM) will execute. Thus the output of the javac is not the directly executable code.

To actually run the program, we must use Java interpreter, called "java".

This is interpreter the "Example.class" file given as input.
When the program is run with java interpreter.

***For Micro Notes by the Student***

**output:**
Hello World

**Description of the every line of the program:**

- The first line contains the keyword class and class name, which actually the basic unit for encapsulation, in which data and methods are declared.
- Second line contains "{" which indicates the beginning of the class.
- Third line contains the public static void main(String args[])
- Where public is access specifier, when a member of a class is made public it can be accessed by the outside code also. The main function is the beginning of from where execution starts.
- Java is case-sensitive. "Main" is different from the "main". In main there is one parameter, String args, which is used to read the command line arguments. Fourth line contains the "{", which is the beginning of the main function.
- Fifth line contains the statement System.out.println("Hello World");
- Here "System" is the predefined class, which provides access to the system, and out is the output stream that is used to connect to the console. The println(), is used to display string passed to it. This can even display other information to.

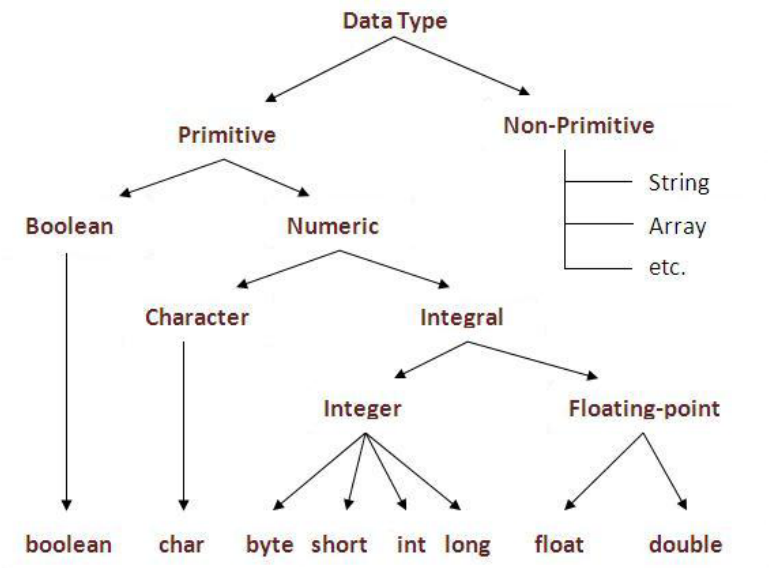
## JAVA BASICS

**Data Types:**
Java is strongly typed language. The safety and robustness of the Java language is in fact provided by its strict type.
There are two reasons for this:
First, every variable and expression must be defined using any one of the type.
Second, the parameters to the method also should have some type and also verified for type compatibility.

**8 primitive data types in Java:**

***For Micro Notes by the Student***

The primitive data types are:
char, byte, short, int, long, float, double, boolean.
These are again grouped into 4 groups.




1. Integer Group: The integer group contains byte, short, int, long. These data types will need different sizes of the memory. These are assigned positive and negative values.

The width and ranges of these values are as follow:

**byte:** 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. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. 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. We can store byte and short values in an int.

Example

int x=12;

**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. Example long x=123456;

**2. Floating-Point Group**

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. These are used with operations such as square root, cosine, and sine etc.

There are two types of Floating-Point numbers:

float and double.

The float type represents single precision and double represents double precision.

Their width and ranges are as follows:

**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. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.

Example:

float height, price;

**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 the math functions, such as sin( ), cos( ), and sqrt( ), return double values.

**Example:**

***For Micro Notes by the Student***

*For Micro Notes by the Student*

```
double area,pi;
```

**Example program to calculate the area of a circle**

```
import java.io.*;
class Circle
{
public static void main(String args[])
{
double r,area,pi;
r=12.3;
pi=3.14;
area=pi*r*r;
System.out.println("The Area of the Circle is:"+area);
}
}
```

**3. Characters Group:**

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. Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

Here is a program that demonstrates char variables:

```
// Demonstrate char data type.
class CharDemo
{
public static void main(String args[])
{
char ch1, ch2;
ch1 = 88; // code for X
ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " + ch2);
}
}
```

**4. Booleans:**

*For Micro Notes by the Student*

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. Here is a program that demonstrates the boolean type:

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[])
{
boolean b;
b = false;
System.out.println("b is " + b);
b = true; System.out.println("b is " + b);
// a boolean value can control the if statement
if(b)
System.out.println("This is executed.");
b = false;
if(b)
System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

**Identifiers:**

Identifiers are used for class names, method names, and variable names.

**Rules of an Identifier:**

1. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
2. They must not begin with a number, lest they be confused with a numeric literal.
3. Again, Java is case-sensitive, so VALUE is a different identifier than Value.

**valid identifiers :**

**Ex:**

Average, Height, A1, Area_Circle .

**Invalid Identifiers:**

*For Micro Notes by the Student*

Ex:
2types, Area-circle, Not/ok.

**Variables:**

The variable is the basic unit of storage in a Java program.

A variable is defined by the combination of an identifier, a type, and an optional initializer.

In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable In Java, all variables must be declared before they can be used.

The basic form of a variable declaration is shown here:

**type identifier [ = value][, identifier [= value] ...] ;**

Here the type is any primitive data types, or class name.

The identifier is the name of the variable. We can initialize the variable by specifying the equal sign and value.

**Example**

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing // d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; //the variable x ahs the value 'x'
```

**Dynamic Initialization of the variable:**

We can also assign a value to the variable dynamically as follow:

```
int x=12;
int y=13;
float z=Math.sqrt(x+y);
```

**The Scope and Lifetime of Variables:**

Java allows, to declare a variable within any block.

- A block begins with opening curly brace and ended with end curly brace. Thus, each time we start new block, we create new scope.
- A scope determines what objects are visible to parts of your program.
- It also determines the life time of the objects.
- Many programming languages define two scopes: Local and Global
- As a general rule a variable defined within one scope, is not visible to code defined outside of the scope.

- Scopes can be also nested. The variable defined in outer scope are visible to the inner scopes, but reverse is not possible.

***For Micro Notes by the Student***

**Example code**

```
void function1()
    {
       //outer block
       int a;
    //here a,b,c are visible to the inner scope
    int a=10;
    if(a==10)
    {
    // inner block
    int b=a*20;
    int c=a+30;
    }//end of inner block
    b=20*2; // b is not known here, which declared in inner scope
    }//end of the outer block
```

**Literals:**

A constant value can be created using a literal representation of it. Here are some literals:

```
int x=25;
char ch=88;
flaot f=12.34;
byte b=12;
```

**Comments:**

In java we have three types of comments:

1. single line comment,
2. Multiple line comment, and
3. document type comment.

1. Single line comment is represented with // (two forward slashes),
2. Multiple comment lines represented with /*……*/ (slash and star),
3. document comment is represented with /**……*/.

Separators In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have

*For Micro Notes by the Student*

seen, it is used to terminate statements.

The separators are shown in the following table:

**The Java Keywords**

There are 50 keywords currently defined in the Java language (see Table below).

| | | | | |
|---|---|---|---|---|
| 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 |

Some noteworthy points regarding Java keywords:

- o const and goto are reserved words but not used.
- o true, false and null are literals, not keywords.
- o All keywords are in lower-case.

These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.

These keywords cannot be used as names for a variable, class, or method.

## OPERATORS

- An operator is a symbol that tells the compiler to perform a specific mathematical, logical or other manipulation.
- Java has 4 general classes of operators:
  1. Arithmetic operator
  2. Relational Operator
  3. Logical Operator
  4. Bitwise Operator

**Arithmetic Operator (+,-,*,/,%,++,--)**

Arithmetic operators are used in mathematical expressions in the same way they are used in algebra.

| Operator | Example | Meaning |
|---|---|---|

*For Micro Notes by the Student*

| | | |
|---|---|---|
| + | a+b | Adds values on either side of operator |
| - | a-b | Subtracts right hand operand from left hand operand |
| * | a*b | Multiplies values on either side of the operator |
| / | a/b | Divides left hand operand by right hand operand |
| % (modulus) | a%b | Divides left hand operand by right hand operand and returns remainder |
| ++ (increment) | a++ | Increases the value of the operand by 1 |
| --(decrement) | a-- | Decreases the value of operand by 1 |

**Increment and decrement operator:**

- ++ and – are java increment and decrement operators.
- Increment operator increases the value of operand by 1 and decrement operator decreases the value of operand by 1.
- Both increment and decrement operator can either precede (prefix) or follow(postfix) the operand.
- Increment and decrement operator applicable only for variables but not for constant values.
- If you are using prefix form then increment/decrement will be done before the rest of the expression.

  Example: x=10;

  Y= ++x;

  Y will be set to 11 and x increments to 11.

- If you are using postfix form, then increment/decrement will be done after the complete expression is evaluated.

  Example: x=10;

  Y=x++;

  Y will be set to 10, x increments to 11.
- In both the cases, X will be set to 11 but the difference is when it happens.

**Relational Operator:**

- Relational operator produce true or false. Relational operator supported by Java:

| Operator | Meaning |
|---|---|

*For Micro Notes by the Student*

| | |
|---|---|
| == (equal to) | Checks if the value of two operands are equal or not. If yes then condition becomes true. |
| != (not equal to) | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. |
| > (Greater than) | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. |
| < (less than) | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. |
| >= (Greater than or equal to) | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true |
| <= (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. |

**Logical operator:**

- The operand must of type Boolean and the result of a logical operation is of type Boolean.
- The logical operators are &, |, ^, and !

**Short circuit logical operator:**

- In simple terms, short circuit means stopping evaluation once you know that the answer can no longer change.
- Example: **a&b**, the **AND** operator results in false when **a** is false, no matter what b is.[it is not necessary to know what the right side] similarly **a|b**, the **OR** operator results in true when a is true, no matter what **b** is.
- The short circuit AND operator is &&, ant the short circuit OR operator is ||. Their normal counterpart are & and | respectively.
- The && and || don't evaluate the right hand side if it isn't necessary. The & and | operator always evaluate both sides.

**Assignment Operator:**

- The assignment operator is the single equal sign, =
- General form of assignment operator:

  **Var=expression;**

  Here type of var must be compatible with the type of expression.

- **Simple assignment:** int x=10; we are assigning 10 to the int variable.
- **Chained assignment:** a=b=c=d=e=10;
- **Compound assignment:** Assignment mixed with some other operator. Ex: a+=20;

***For Micro Notes by the Student***

**Bitwise operator:**

- Bitwise operator can be applied to values of the long, int, char, short, byte.
- Bitwise operations can't be used in Boolean, float, or double.
- They are called bitwise operator because they are used to test, set, or shift the individual bit that makes up a value.

**The bitwise AND, OR, XOR and NOT operator**

- & ( AND) returns true if and only if both argument are true.
- | (OR) returns true if and only if atleast one argument is true.
- ^ (XOR) returns true , if and only if both argument are different and it returns false if both argument are same.

```
public class bitwsiedemo {
    public static void main(String[] args) {
            System.out.println(true&false);
            System.out.println(true|false);
            System.out.println(true^false);
            System.out.println(10&11);
            System.out.println(10|11);
            System.out.println(10^11);
      }
  }
```

**Output:**

```
false
true
true
10
11
1
```

**Bitwise complement(~)**

Bitwise complement applicable only for integer type but not for Boolean.

*For Micro Notes by the Student*

```
public class bitwi {
    public static void main(String[] args) {
        System.out.println(~4);
    }
}
```

**Output: -5**

**Conditional operator: (?:)**

- Conditional operator is called as ternary operator.

  Example: **int x=(15<20)?80:90);**

  **System.out.println(x);**

  (2) **int x=(15<10)?80: ((50>70):100:120);**

  **System.out.println(x);**

**Type conversion and casting:**

- It is common to assign a value of one type to a variable of another type. If the two types are compatible, then java will perform the conversion automatically.

  Example: You might want to assign an int value to a float variable,

  ```
  int  i;
  flaot f;
  i=10;
  f=i;
  ```

  the value of i is converted into a float & then assigned to f.

- When compatible types are mixed in an assignment, the value of the right side is automatically converted to the type of the left side.

  Example: int and float are compatible.

  Boolean and int are not compatible.

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if:
  1) The two types are compatible.
  2) The destination type is larger than the source type.

  double>float>long>int>short>byte
- When we assign smaller data type value to bigger data type variable i.e., keeping small value in the big container is known as auto widening or upcasting.
- There is no loss of information in implicit type casting.

***For Micro Notes by the Student***

- Areas where implicit type casting is performed:
  byte -> short -> int -> long -> float -> double

**Explicit type casting/casting incompatible type:**

- To create a conversion between two incompatible types, we must use a cast.
- A cast is performed by placing the desired type in the parenthesisto the left of the value to be converted.
- **General form:** *(target_type)* value
  *target_type* specifies the desired type.

Example: if we want to convert the type of the expression x/y to int

```
double x,y;
// …
int z=(int) (x/y);
```

x and y are of type double, the cast converts the outcome of the expression to int.

- When we are converting data from big sized data type to small sized data type i.e. keeping big value in small place/container is known as narrowing or down casting.
- When a cast involves a narrowing conversion, information might be lost.
  Example: int x=130;

```
byte b=(byte) x;
System.out.println(b);  //Output:-126
```

**Enumeration (enum):**

- An enumeration is a list of constants.
- If we want to represent a group of constant then we go for enumeration.
- An enumeration is created using the keyword **enum**.

**Enum declaration and uses:**

```
enum Phone
{
        iphone, Samsung galaxy, Motorola, Lg, Sony;
}
```

- iphone, Samsung_galaxy, Motorola, Lg, Sony are called enumeration constants.
- These constants are implicitly public static final.
- These constants represent an object of the type month. iphone is one type of Phone object, Samsung_galaxy is one type of Phone object.
- Once we have defined enumeration, we can create a variable of that

*For Micro Notes by the Student*

type.

Example: Phone MyPhone;

- MyPhone is of type Phone, the only values it can be assigned are those defined by the enumeration.

Example: MyPhone=Phone.iphone;

System.out.println(MyPhone); // prints iphone

**Enumeration value can also be used to control switch statement**

```
enum month
{
    jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec;
}
public class ENUMDEMO {

        public static void main(String[] args)
        {
                month m10=month.oct;

        switch(m10)
                {
                        case jan:System.out.println("31 days");
                                                break;
                        case feb:System.out.println("28 days");
                                        break;
                        case oct:System.out.println("31 days");
                                        break;
                        case nov:System.out.println("30 days");
                                        break;
                        case dec:System.out.println("31 days");
                                        break;
                }
        }
}
```

**OUTPUT:**

31 days

**Control Flow Statements:**

Java Program control statement can be put into the following categories:

1) **Selection** statements allow your program to choose different path of

*For Micro Notes by the Student*

execution based upon the outcome of an expression. Java Support two selection statements: **if** and **switch**

2) **Iteration** statements enable program execution to repeat one or more statements. Java's iteration statements are **for, while** and **do-while**
3) **Jump** statements transfer control to another part of your program. Java support three jump statement: **break, continue** and return.

**Java's Selection Statement**

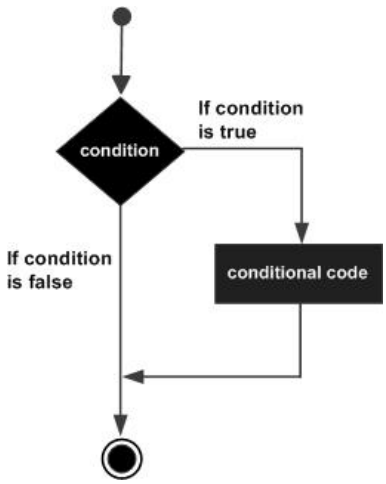
**if Statement:** if statement tests the condition. The general form of if statement:

```
if(condition)
{
    Statement1;
}
Else
{
    Statement2;
```

Here, condition is an expression that evaluates to either true of false. if the condition is true, then statement1 is executed. If condition is false, then statement2 is executed.

```
}
```

**Flow chart:**




**Task 1:**

Write a Java Program to find the given number is even or odd.

**Solution:**

***For Micro Notes by the Student***

**Nested if:** Nest if-statements, which means we can use one if or else if statement inside another if or else if statement.

Syntax for nested if:

```
If(condition1)
{
    Statement1;
    If(condition2)
    {
       Statement2;
       Statement3;
    }
}
```

An else statement always associates with the nearest if statement in the same block.

**Task 2:**

Write a Java Program to find the biggest number among three numbers.

**Solution:**

**Task 3:**

Given an integer, n,perform the following conditional actions:

If **n** is odd, print **Weird**

If **n** is even and in the inclusive range of 2 to 5, print **Not Weird**

If **n** is even and in the inclusive range of 6 to 20, print **Weird**

If n is even and greater than 20, print **Not Weird.**

Write the program to print whether or not n is weird.

Input Format:

A single line containing a positive integer, n.

**Constraints**

1<=n<=100

**Solution:**

**Switch Statement:**

- The switch statement provides a multiway branch.
- It enables a program to select among several alternatives.
- The value of an expression is successively tested against a list of constants. When a match is found, the statement sequence associated with that match is executed.
- The general form of switch statement:

**switch(expression)**
**{**
**case value1: Statement1;**
**break;**
**case value2: Statement2;**
**break;**
.
.
.
**case valueN: StatementN;**
**Break;**
**default: default statement;**
**}**

-> The variable used in switch expression can be of type: byte, short, int, char, enum, string.
-> 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.
-> Duplicate case values are not allowed.
-> When matching case is found, the statement following that case will execute until a break
statement is reached.
-> When a break statement is reached the switch terminates.
-> If a case sequence doesn't contain a break, execution continues into the next case sequence until a break is reached.
-> A switch statement can have an optional default case, which must appear at the end of the switch. No break is needed in the default case.

***For Micro Notes by the Student***

***For Micro Notes by the Student***

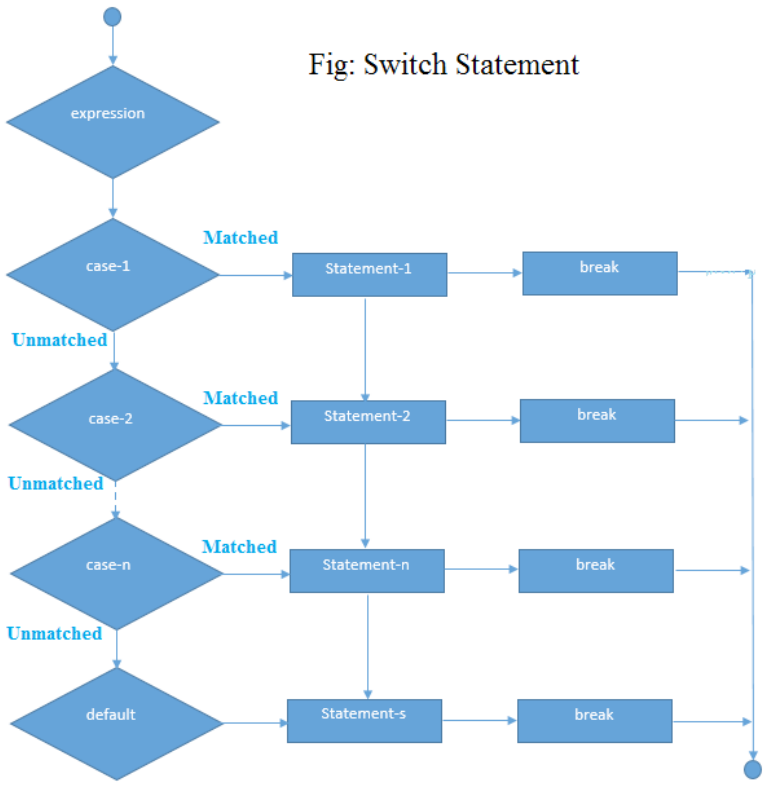



Fig: Switch Statement

**Task 4:**

Write a java program that read three inputs (operator and 2 numbers) from the keyboard and perform calculation based on numbers and operator entered. Then the result is displayed on the screen.

**Solution:**

***For Micro Notes by the Student***

**Java's iteration statements:**

**for loop:**

- **for** loop is used whenever, user want to repeat some statements or block of code for a specific times.
- The general form of for loop:

```
for(initialization;condition;iteration)
{
Statement sequence
}
```

It has 3 parts: initialization, condition & iteration(increment/decrement)

(1) Initialization is executed first, and only once. This step allows us to declare and initialize any loop control variable.

(2) Next, the condition is evaluated. If it is true, the body of the loop is executed. if it is false, the body of the loop doesn't execute and the flow of control jumps to the next statement just after the for loop.

(3) After the body of the for loop executes, the flow of control jumps back up to the increment/decrement statement. This statementallows us to update any loop control variables.

The condition is now evaluated again. If it is true, the loop executes and the process repeats itself.

After the condition becomes false, the **for** loop terminates.

**Variations on the for loop**

(1) **Use of multiple loop control variables:** Java allows two or more variables to control a **for** loop. Java permits you to include multiple statements in both the initialization and iteration portion of the **for**. Each statement is separated from the next by a comma.

```
for(   i=0 , j=1  ;   i<=j  ;  i ++ , j++)
{
        Statements;
}
```

(2) **Infinite loop:** You can intentionally create an infinite loop( loop that never terminates) if you leave all three parts of the **for** empty.

*For Micro Notes by the Student*

```
for(     ;     ;     )
{
        Statement;
}
```

(3) **Missing piece:** Either the initialization or the iteration expression or both may be absent.

```
for(  ; a>b ;  )
{
        Statement;
        a--;
}
```

**Task 5:**

Write a Java program to find the factorial of a given number.

**Solution:**

**While loop:**

- While loop repeats a group of statements as long as a condition is true. Once the condition becomes false, the loop is terminated and control of flow jumps to the next statement just after the for loop.
- **Syntax:**

```
while(condition)
{
        Statements;
}
```

**Task 6:**

Write a Java program to find the given number is prime or not.

***Solution:***

***For Micro Notes by the Student***

**do… while loop:**

do-while loop is used when there is a need to repeatedly execute a group of statement as long as a condition is true. If the condition is false, the repetation will be stopped and the flow of execution comes out of do-whle loop.

**Syntax:**

```
do
{
Statements;
}while(condition);
```

**Task 7:**

In fibonacci series, next number is the sum of previous two numbers for example 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 etc. The first two numbers of fibonacci series are 0 and 1.

**Solution:**

**Difference between while and do… while statement**

| while loop | do… while loop |
|---|---|
| 1. Entry control Loop.<br>2. Condition is checked at the start.<br>3. While loop statements (body) may not be executed even once.<br>4. There is no semicolon at the end of while statement. | 1. Exit control loop.<br>2. Condition is checked at the end of structure.<br>3. Do while loop statements (body) are executed at least once.<br>4. There is semicolon at the end of while statement |

**Jump statements:**

Java support three jump statements: break, continue and return. These statements transfer control to another part of your program.

**Using break:** (to exit)

- Java break is used to break loop or switch statement.
- Break statement has the following two usage:

*For Micro Notes by the Student*

1. 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.
2. It can be used to terminate a case in the switch statement.

- when break statement is used inside a set of nested loops, the break statement will break out of only the innermost loop. The outer loop is unaffected.

**Break as a form of goto:**

- Java doesn't have goto statement , because it provides an unstructured way to alter the flow of program execution.
- Break statement gives you the benefit of a goto .
- The general form of the labeled break statement is :

  **break label;**

  Here, label is the name that identifies a block of code.

  -When this form of break executes, control is transferred out of the named block of code.
- To name a block, put a label at the start of it.
- A label is any valid java identifier followed by colon.
- The following code demonstrates the labeled block or labeled break.

```
public class breakgoto {
    public static void main(String[] args) {
       boolean x=true;
       bl1:{
          bl2:{
                bl3:{
                      System.out.println("Block3");
                      if(x) break bl2;
                   } //end bl3
              System.out.println("block2");
              } //end block2
           System.out.println("block1");
           } //end block1
       System.out.println("Out of all block");
        }
    }
    OUTPUT
    Block3
    block1
    Out of all block
```

***For Micro Notes by the Student***

**Continue Statement**

- 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 iteration portion of the for statement & then to the conditional expression.
- In a while loop or do-while loop a continue statement causes control to be transferred directly to the conditional expression that controls the loop.
- The syntax of a continue is a single statement.

  **Continue;**
- As with the break statement, continue may specify a label to describe which enclosing loop to continue.

```
public class contexample {

    public static void main(String[] args) {
    int i;
    for(i=0;i<=10;i++)
        {
        if(i<5) continue;
        System.out.println(i);
        }
    }
    }
```

**OUTPUT**

5, 6, 7, 8, 9, 10

**ARRAYS:**

- An array is collection of fixed number of homogenous data element, referred to by a common name.
- There are two type of array: Single dimension array , Two dimensional array, multi dimensional array.

***For Micro Notes by the Student***

- Array in java is indexed based, first element of the array stored at 0 index.
- The main advantage of array is we can represent huge number of elements by using single element.

**Limitations Of Array:**

1) Arrays are fixed in size: Once we have created an array with some size, it is not possible to increase or decrease size based on our requirement.
   To use array concept we should know size in advance, which may not possible always.
2) Arrays can hold only homogenous data type element: Array can hold only homogenous data type element (similar type of data). If we have an int array, we can store only int type of element. If we try to insert any other data type like Boolean, float we get compile time error.

[To overcome these problem, we have collection concept in unit-4. Collection is not fixed in size and collection can accept both homogenous and heterogeneous element]

**Array Creation:**

**One dimensional Array creatation:**

- Array in java is an Object, we can create an array by using new operator.
- Since Arrays are implemented as Object, the creation of an Array is a two **step process.**

**Step1: Declare an Array:** to declare an array below is a general form/Syntax:

**data_type array_name[];**

Example: int simple[];
int[] simple;
int []simple;

**Step2: Array creation/ Allocate memory for the Array:** Array in java is an object, we create an object by using new operator.

To create/ allocate memory for the array element use below **general form:**

**array_name = new type [size];**

Example: sample = new int [10];
sample = new int [0];

- *[Both the above step combined in single line]* You will use this general

form:

**type[] array_name=new type[size];**

Example: int sample = new int [10];

- An individual element within an array is accessed by use of index. An index describes the position of an element within an Array.
- Array index begins with zero that means if you want to access first element of an array use zero as an index.
- Each individual array element is used in the same way as a 'normal variable'.

**Example:** To assign a value 3 to the first element in sample…sample[0]=3;

***For Micro Notes by the Student***

**Task 8:**

Write a java program to find the largest and smallest number in the given array.

**Solution:**

***For Micro Notes by the Student***

**Two dimensional Array creatation:**

In C/C++ we use matrix representation (group of rows and column) to represent 2 dimensional array. But in JAVA we use Array of Arrays approach for 2 dimensional/multi dimensional array.

**Step1: Declare an Array:** For two dimensional array we use 2 square bracket. The general form to declare two dimensional array:

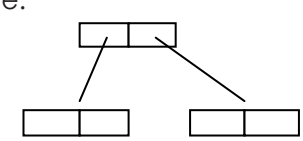
**data_type[][] array_name;**

Example: int[][] simple;

**Step2: Array creation: Two dimensional** arrays are not implemented by using matrix.

Equivalent Java code for this memory structure:

**int simple[][] = new int [2][2];**



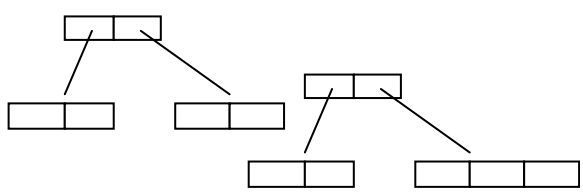
Here, in this memory Structure **base size is 2.**

**int simple1[][] = new int[2][ ];**

Next level, Size is sometimes 2 or 3, not fixed

**simple[0]= new int[2];**

**simple[1] = new int[3];**



**Array Initialization:**

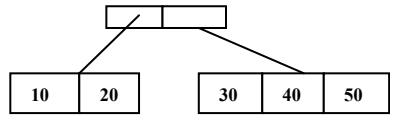
- Once we have created an array automatically every array element is initialized with default values.
- Array can be initialized when they are created. The general form for initializing a one dimensional array is:

**type[] array_name = { val1, val2, val3, … ,valn };**

Here, the values to be inserted into the array are listed inside { … } block.

- Java automatically allocates an array large enough to hold the initializers that you specify. There is no need to explicitly use new operator.
- Example: int[] a={10,20,30,40,50};
  char[] ch={'a', 'b', 'c', 'd', 'e'};
  string[] s={"ACE", "Engg", "College"};
- **Two dimensional array initialization:**

Example: **int[][] x={ {10,20},{30,40,50}};**




***For Micro Notes by the Student***

**Task 9:**

Write a java program to find the addition of two matrices.

**Solution:**

**Task 10:**

Write a java program to find the multiplication of two matrices.

**Solution:**

***For Micro Notes by the Student***

**Strings:**

Generally, String is a sequence of characters. But JAVA implement strings as an object of type string.

**Strings are immutable:** immutable simply means unmodifiable or unchangeable. When you create a string object you are creating a string that can't be changed. i.e once string object is created you can't change the characters that comprise that string.

Each time we need an altered version of an existing string, a new string object is created that contains the modifications. The original string is left unchanged.

**Constructing String:**

1) The easy and direct way to create a string is:

   **String str = "ACE Engineering College";**

   Compiler creates a string object str with its value ACE Engineering College

2) You can create string object by using new keyword and calling the string constructor.

   **String str = new String("ACE Engineering College");**

   This creates a string object called str that contains the character string ACE Engineering College.

3) You can construct a string from another String.

**String str = new String(**"ACE Engineering College"**);**

**String str1 = new String(str);**

*For Micro Notes by the Student*

**Java String class methods**

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

| No. | Method | Description |
|---|---|---|
| 1 | char charAt(int index) | returns char value for the particular index |
| 2 | int length() | returns string length |
| 3 | static String format(String format, Object... args) | returns formatted string |
| 4 | static String format(Locale l, String format, Object... args) | returns formatted string with given locale |
| 5 | String substring(int beginIndex) | returns substring for given begin index |
| 6 | String substring(int beginIndex, int endIndex) | returns substring for given begin index and end index |
| 7 | boolean contains (CharSequence s) | returns true or false after matching the sequence of char value |
| 8 | static String join(CharSequence delimiter, CharSequence... elements) | returns a joined string |
| 9 | static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements) | returns a joined string |
| 10 | boolean equals (Object another) | checks the equality of string with object |
| 11 | boolean isEmpty() | checks if string is empty |
| 12 | String concat(String str) | concatinates specified string |
| 13 | String replace(char old, char new) | replaces all occurrences of specified char value |
| 14 | String replace(CharSequence old, CharSequence new) | replaces all occurrences of specified CharSequence |
| 15 | static String equalsIgnoreCase (String another) | compares another string. It doesn't check case. |

*For Micro Notes by the Student*

| | | |
|---|---|---|
| 16 | String[] split(String regex) | returns splitted string matching regex |
| 17 | String[] split(String regex, int limit) | returns splitted string matching regex and limit |
| 18 | String intern() | returns interned string |
| 19 | int indexOf(int ch) | returns specified char value index |
| 20 | int indexOf(int ch, int fromIndex) | returns specified char value index starting with given index |
| 21 | int indexOf(String substring) | returns specified substring index |
| 22 | int indexOf(String substring, int fromIndex) | returns specified substring index starting with given index |
| 23 | String toLowerCase() | returns string in lowercase. |
| 24 | String toLowerCase(Locale l) | returns string in lowercase using specified locale. |
| 25 | String toUpperCase() | returns string in uppercase. |
| 26 | String toUpperCase(Locale l) | returns string in uppercase using specified locale. |
| 27 | String trim() | removes beginning and ending spaces of this string. |
| 28 | static String valueOf(int value) | converts given type into string. It is overloaded. |

**Task 11:**

Write a java program to count the number of words in a string?

**Solution:**

***For Micro Notes by the Student***

**Task 11:**

Write a java program which demonstrates all string handling functions?

**Solution:**

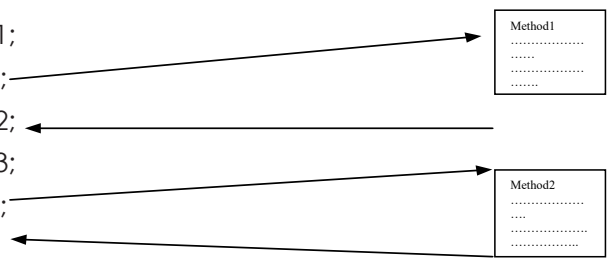
***For Micro Notes by the Student***

**Methods:**

- A simple class contains variable and methods.
- A method is a collection of statement that are grouped together to perform an operation.
- Each method has name, when that name is encountered in a program, the execution of the program branches to the body of that method.
- When the method is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.

**Example:**

```
public static void main(String args[])
{
statement1;
method1();
statement2;
statement3;
method2();
statement;
....
}
```




- **Elements of method declaration are:**

a) Method return type
b) Method name
c) Pair of parenthesis and a body between braces { }.

- The **general form** of method is:

```
ret_type method_name(parameter_list)
{
//body of method
}
```

Returning from the method:

2 condition causes a method to return to the code that invoke it. When it:

1. Completes all the statement.
2. Reaches a return statement and is executed.

**Task 12:**

Write a Java Program to define a class, define instance methods for setting

and getting values of instance variables and instantiate its object.

***For Micro Notes by the Student***

**Constructor:**

- Constructor is invoked at the time of object creation.
- Constructor is used to initialize data members of a class.
- Constructor name must be same as its class name.
- Constructors have no return type, not even void.
- All class have constructor, whether you define or not.
- If you don't write the constructor, a default constructor will be generated by the compiler.
- Two type of constructor: Default constructor and parameterized constructor.
- A constructor with no parameter is known as default constructor.

**Parameterized constructor:** A constructor that has parameter(s) is known as parameterized constructor.

**Example:**

public class consructorDemo {

***For Micro Notes by the Student***

```
public static void main(String[] args) {
        first obj=new first();
        first obj1=new first(10);
        }
}
class first
{
        first()
        {
        System.out.println("Constructor  0 parameter");
        }
first(int a)
{
System.out.println("Constructor with 1 parameter");
}

}
```

OUTPUT:
Constructor 0 parameter
COnstructor with 1 parameter

**Static Keyword:**

We can apply static keyword to variables, methods and blocks.

**Static variable:**

A static variable can be accessed directly by using class name. We do not require object of class to access static variable.

Static variable can accessed by calling **class_name.variable_name;** without need to create instance for the class.

Value of a static variable will be common for all instances.

All the instance of the class share the same copy of the variable. (that is not unique for each object)

**Static method:**

- A static method belongs to class rather than object. It can be called directly by using the class name: **class_name.method_name();**
- **Restriction:** Static method can access only static variable , it can not access non static variable.

```
class abc
{   int i=10;
    static int j=25;
```

*For Micro Notes by the Student*

```
    public static void show()
    {
    System.out.println("Hiee im static method show");
    //System.out.println(i); Error:i is non ststic variable. Show() can access only
    static variable
    System.out.println(j); // Static method can access static variable j
    }
}
public class staticvarmet
{
        public static void main(String[] args)
        {
        abc.show(); // Static method show() don't need object of class abc,
    only
                require class name
        System.out.println(abc.j); //Static varibale don't need object, can be
            accessed using class name
        }
}
```

**Static Block:**

- A static block is a block of code inside a class that will be executed when a class is first loaded into the JVM.
- Mostly static block is used to initialize static variable. ( we can't use constructor because constructor only works on instance variable)
- A static block is executed when the class is first loaded.

```
public class staticblock
{

static
{
System.out.println("i will still get executed when u don't write anythig in
main()");
}

    public static void main(String[] args)
            {
```

*For Micro Notes by the Student*

```
        }

    }
```

**Point to remember :** static block gets called when class is loaded into JVM before main method

**Task 13:**

Write a java program to count the total number of students admitted in the college.

**Solution:**

***For Micro Notes by the Student***

**Access control/ Access specifier:**

Access specifier are used to specify the scope of a member of class and class itself.

The four access specifier are : **private, protected, public** and **default.**

**Private:**

- Private access specifier can be given to any member of class. When we say any member it would be variable in a class or method in class or constructor as well.

- When we specify private modifier for a member of class it means the member can be accessed only within side of class not outside.

**Protected:**

When we specify protected, any protected member can be accessed within the same class, within subclass/child class whether child class is the same package or in a different package and within non-sub class in the same package.

Private and protected can be given to member of class, not class itself.

**Public:**

Public modifier when specified public member can be accessed from anywhere.

**Default:**

When we have default scope, it means member can be accessible within the same class, within a sub class provided sub class in same package, within the non sub class in the same package.

*For Micro Notes by the Student*

**this** keyword: refers current object

**Uses:**

2. this keyword can be used to refer current class instance variable.
3. this() can be used to invoke current class constructor.

1) **this keyword can be used to refer current class instance variable.** If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

```
class example
{
        int x;
        public example(int x)
        {
                this.x=x;
        }
        public void show()
        {
        System.out.println(x);
        }
}
public class thisdemo
{
public static void main(String[] args)
   {
   example obj = new example(10);
        obj.show();
   }
}
```

2) **this() can be used to invoke current class constructor.** This() constructor call can be used to invoke the current class constructor. This approach is better when we have many constructor in the class and want to reuse that constructor.

```
class example
{
        int x;
```

*For Micro Notes by the Student*

```
        example()
        {
        System.out.println("default
                constructor");
        }

        public example(int x)
        {
                this();
                this.x=x;
    }

    public void show()
    {       System.out.println(x);
    }
}

public class thisdemo
{
public static void main(String[] args)
        {
        example obj = new example(10);
                obj.show();
        }
    }
```

**Overloading Methods:**

Two or more methods with the same name but different signature( number or sequence of parameter) is known as Method overloading.

```
        class OverloadDemo
        {

                void test()
                {
                System.out.println("No parameters");
                        }

                void test(int a)
                {
                System.out.println("a: " + a);
                }

                void test(int a, int b)
```

*For Micro Notes by the Student*

```
        {
        System.out.println("a and b: " + a + " " + b);
        }

    }

    class overloadingmethod
    {
        public static void main(String args[])
        {
        OverloadDemo ob = new OverloadDemo();
            ob.test();
            ob.test(10);
            ob.test(10, 20);

        }
    }
```

**Task 14:**

Write a Java Program to define a class, define instance methods and overload them and use them for dynamic method invocation.

**Solution:**

***For Micro Notes by the Student***

**Task 15:**

Write a java program which contains a method fun() such that fun(x) returns $x^2$ and fun(x,y) returns $x^2+y^2$ (x and y are integers).

**Solution:**

**Recursion:**

- Recursion is a technique in which a Function calls itself until certain condition is satisfied.
- A Function, which contain call to itself is known as a recursive function.
- In recursion, the calling function and called function are same.

**Recursive version to compute factorial**

**Factorial:** Product of all the numbers from 1 to that number including itself.

```
import java.util.Scanner;
public class RecFactorial
{
                                    System.out.println(ft.fact(n));
```

```
public static void main(String[] args)
{
Scanner sc=new
Scanner(System.in);
facto ft=new facto();

System.out.println("Enter number");
int n=sc.nextInt();

}
}
```

```
class facto
{
        int fact(int n)
            {
        if(n==0) return 1;
        else
        return n*fact(n-1);
         }
```

***For Micro Notes by the Student***

**Task 16:**

**Recursive Version of Fibonacci numbers**

**Fibonacci:** Each number is the sum of the previous two numbers except the first and second number.

**0, 1, 1, 2, 3 , 5, 8, 13, ….**

**Solution:**

***For Micro Notes by the Student***

**Garbage Collection:**

- Garbage collection one important feature in java.
- In C/C++, Programmer is responsible for allocating memory by using malloc() or calloc() and programmer is responsible for de-allocating memory using free() or delete()
- In JAVA, we can create object by using **new** keyword but we can't remove an object or delete an object once it is done.
- Free an object is not allowed in Java.
- In JAVA, we have the garbage collector which takes care of releasing object that goes out of scope.
- A garbage collector is an application program which always runs in the background.
- What garbage collector does is it collects objects that goes out of scope and remove those unreferenced objects.
- System.gc() method is used to explicitly call garbage collector but it is not guaranteed that garbage collection happens at that time because garbage collection thread priority is very low its priority is zero.

**Inheritance:**

- Inheritance can be defined as the process where one class acquires the properties of another.
- It's a mechanism by which one class acquires all the properties and behavior of an existing class.
- A class that is inherited is called as Super class/ Parent class/ Base class. The class which inherits the properties of another class is called sub class/ child class/ derived class.

*For Micro Notes by the Student*

- **Extends** keyword is used to inherit the properties of a class.

Syntax:

```
class super
{
.....//body of class
....
}

class sub extends super
{
....// body of class
....
}
```

- The extend keyword indicates that you are making a new class derives from an existing class.
- You can specify only one super class for any sub class that you create. Multiple inheritance is not supported in Java.
- **Major advantage of inheritance:** Code reusability: sub class uses all the members from its super class without rewriting the same.

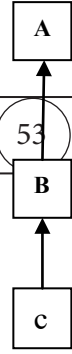
**Types of Inheritance:**

**Single level inheritance:** We have two class, class A and class B. if class B Extends class A it becomes single level inheritance. So we have only one Level. Arrow points to parent class.

```
class A
{

}

class B extends A
{

}
```


**Multilevel Inheritance:**

```
class A
{
```




*For Micro Notes by the Student*

```
}
class B extends A
{

}

class C extends B
{
}
```

**Member access and Inheritance:**

- Often an instance variable of a class is declared private to protect data.
- In inheritance even though a sub class includes all of the member of its super class, it can't access private members of the super class.
- Inheriting a class doesn't over rule the private access restriction.

```
public class memeberaccess
{
        public static void main(String[] args)
        {
        classB obj=new classB();
                obj.show();
        }
}

class classA
{
    int a=10; //default memeber
    private int b=20;
    protected int c=30;
    public int d=50;

    public void show()
    {
       System.out.println(a);
       System.out.println(b);
       System.out.println(c);
       System.out.println(d);
    }
```

*For Micro Notes by the Student*

```
}

class classB extends classA
{
    public void show()
    {
        System.out.println(a);
        System.out.println(b); // Error: classA.b is not visible
        System.out.println(c);
        System.out.println(d);
    }
}
```

**Constructor and inheritance:**

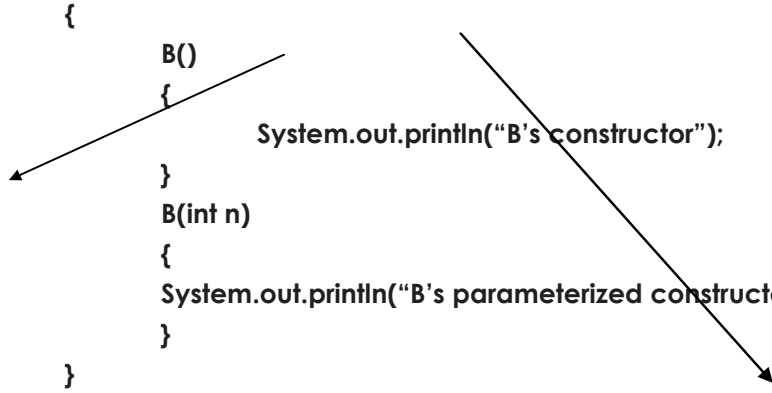
- It is possible for both super class and sub class to have their own constructors.
- When both the super class and the sub class defines constructors, the process is bit more complicated because both the super class and sub class constructor must be executed.

- Whenever we create object of sub class you also get the object of super class.
- Whenever we call constructor of sub class, it will call default constructor of super class (it automatically gets called) and it will call particular constructor.

```
public class inheritcons {

        public static void main(String[] args)
    {
        B obj=new B();
        B obj1=new B(10);
    }
}
class A
{
    A()
```

*For Micro Notes by the Student*

```
        {
                System.out.println("A's constructor");

        }
        A(int n)
        {
        System.out.println(" A parameterized constructor");
        }
}
class B extends A
{
        B()
        {
                System.out.println("B's constructor");
        }
        B(int n)
        {
        System.out.println("B's parameterized constructor");
        }
}
```



**Output:**

A's constructor
B's constructor
A's constructor
B's parameterized constructor

**Task 16:**

Write a java program that implements educational hierarchy using inheritance.

| Office |
|---|
| Empno<br>Empname<br>Salary<br>Getvalue() |

| Office |
|---|


***For Micro Notes by the Student***

*For Micro Notes by the Student*

**Super :** refers immediate parent class object.

**Uses:**

1. super() is used to invoke parent class constructor.
   General form: **super ( arg_list );**
   super() must always be the **first statement inside sub class** constructor.
2. super is used to invoke parent class method.
3. super is used to refer parent class variable.
   General form: **super.member**
   member can be either variable or method

**super() to call parent class constructor**

```
public class inheritcons
{
        public static void main(String[] args)
        {
                B obj1=new B(10);
        }
}


class A
{
        A()
        {
        System.out.println("A's constructor");
        }
        A(int n)
        {
        System.out.println(" A parameterized constructor");
        }
}

class B extends A
{
```

*For Micro Notes by the Student*

```
                B()
                {
                System.out.println("B's constructor");
                }
                B(int n)
                {
                super(n);    //super(n) calls paremetrised constructor of
    parent class
                System.out.println("B's parameterized constructor");
                }
        }
```

**Output:**

A parameterized constructor
B's parameterized constructor

**super to refer parent class variable**

```
        public class supervar
        {
                public static void main(String args[])
                {
                        B obj=new B();
                        obj.show();
                }

        }

        class A
        {
        int i=10;

        }

        class B extends A
        {
                int i=20;
                public void show()
                {
                System.out.println(i);
                System.out.println(super.i); // Super refers parent class
    variable i
                }
```

*For Micro Notes by the Student*

```
        }
```

**Output**

20
10

**super to call parent class method**

```
        public class supermetho {
                public static void main(String args[])
                {
                        B obj=new B();
                        obj.show();
                }
                }

                class A
                {       public void show()
                        {
                        System.out.println("show A");
                        }
                }

                class B extends A
                {
                        public void show()
                        {  super.show();
                        System.out.println("Show B");
                        }
                }
```

Using **final:**

- The final keyword in java is used to restrict the user.
- Final can be used with:

a) **Variable:** final variable can be initialized once. The value of final variable can't be changed.

```
public class finalvar
{
        public static void main(String[] args)
   {
                final int a=10;
                System.out.println(a);
   }
}
```

*For Micro Notes by the Student*

b) **Method:** final method can't be overridden by sub class. Restricts method overriding.

```
public class finalmet
{
   public static void main(String[] args)
   {
            subclass obj=new subclass();
            obj.show();
   }
}

class superclass
{
   final public void show()
   {
      System.out.println("i am in super class");
   }
}

class subclass extends superclass
{
   public void show() //Error:can't override final method from super class
   {
      System.out.println("i am in child class");
   }
}
```

**To prevent method from being overridden, specify final as a modifier at the start of its declaration.**

**The compiler chacks and gives an error if you try to override the method.**

c) **Class:** final class can't be subclassed. Restricts inheritance.

To prevent class being inherited preceed class declaration with final. Final class can't be inherit by other class

```
public class finalclass
{
        public static void main(String[] args)
        {
                derivedclass obj=new derivedclass();
```

*For Micro Notes by the Student*

```
                obj.show();
        }
}

final class baseclass
{
        public void show()
        {
        System.out.println("i am in super class");
        }
}

class derivedclass extends baseclass //Error: type derivedclass can't
be subclass the final class
        {
        public void show()
        {

        System.out.println("i am in child class");
        }
        }
```

**Method Overriding:**

Method overriding forms the basis for one of the JAVAs most powerful concept: Dynamic method dispatch.

Dynamic method dispatch is the mechanism by which function call is resolved at run time rather than compile time.

Defining a method in a sub class which has the same name and same signature as a method in its super class is known as method overriding.

```
public class override
{

    public static void main(String[] args)
{
        child obj=new child();
        obj.show();
    }
}
```

*For Micro Notes by the Student*

```
class parent
{
public void show()
{
System.out.println("Parent class show()");
}
}
class child extends parent
{
public void show()
{
System.out.println("Child class show()");
}
}
```

There are two ways to achieve abstraction in java:

1. **Abstract class** (0 to 100%): it doesn't provide 100% abstraction because abstract class have both abstract method & concrete method(**method** have complete definition)
2. **Interface** (100%): interface is pure abstract class. Interface can have **only abstract** methods.

**Abstract Method:**

- Abstract method just has the name and signature followed by a semicolon.

  Like this: **void call ();**
- An abstract method is a method that is declared without an implementation.

  We use abstract keyword in method declaration.

  **void abstract call();**

  **abstract** keyword is used to declare the method as abstract.

**Abstract Class:**

- Abstract classes contains abstract method and concrete method.

*For Micro Notes by the Student*

Abstract class may or may not contain abstract methods.

- If a class contains at least one abstract methods, then the class itself must be declared as abstract.

```
public abstract class RahulPhone
{
        public  void call( )
        {
        System.out,println("Calling…");
        }
abstract void driveCar( );
abstract void move( );
}
```

- **When an abstract class is sub classed, the subclass must provide the implementations of all the abstract methods** of its parent class. However, if we fail to provide implementation of any one abstract method, then the subclass must be declared as abstract.

```
public abstract class AdityaPhone extends RahulPhone
{
        public void move()
        {
        System.out.println("moving…");
        }

        abstract void driveCar( );
}
```

- **We cannot create instance of abstract class** in java because they are incomplete. Even though, if your abstract class don't contain any abstract method, you can't create instance of it.
  By making a class abstract, we told compiler that, it's incomplete. Java compiler will throw an error, when a code tries to instantiate abstract class
- **Abstract classes can have Constructors.** What is the purpose of constructor, if we can't instantiate abstract class?
  Constructor can only be called during **constructor chaining**, i.e. when you create instance of concrete implementation class.

```
public class MyPhone extends AdityaPhone
{
```

*For Micro Notes by the Student*

```
        public void driveCar()
        {
        System.out.println("waymo… google self driving car");
        }
}


        public class abstactExample
        {
                public static void main(String args[])
                {
                MyPhone np=new MyPhone();
                        np.call();
                        np.move();
                        np.driveCar();
                }
        }
```

**Difference between abstract class and interface**

| Abstract Class | Interface |
|---|---|
| Abstract class can **have both abstract method and concrete methods.** | Interface can have **only abstract methods.** |
| Abstract class **doesn't support multiple inheritance.** | Interface **supports multiple inheritance.** |
| A class can extend **only one abstract class** | A class **can implement any number of interfaces.** |
| The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| **Example:**<br><br>public abstract class MyPhone<br>{<br>public abstract void call();<br>} | **Example:**<br><br>public interface MyPhone<br>{<br>void call();<br>} |

***For Micro Notes by the Student***

**Task 18:**

Write a Java program to create an abstract class named Shape that contains two integers and an empty method named printArea(). Provide three classes named Rectangle, Triangle and Circle such that each one of the classes extends the class Shape. Each one of the classes contains only the method printArea() that prints the area of the given shape.

**Task 19:**

Create an inheritance hierarchy of Rodent: Mouse, Gerbil, Hamster, etc. In the base class, provide methods that are common to all Rodents, and override these in the derived classes to perform different behaviors depending on the specific type of Rodent. Create an array of Rodent, fill it with different specific types of Rodents, and call your base-class methods.

**Solution:**

***For Micro Notes by the Student***

**Task 20:**

Create an abstract class with no methods. Derive a class and add a method. Create a static method that takes a reference to the base class, downcasts it to the derived class, and calls the method. In main( ), demonstrate that it works. Now put the abstract declaration for the method in the base class, thus eliminating the need for the downcast.

**Solution:**

***For Micro Notes by the Student***

**Task 21:**

Add a new method in the base class of shapes. Java that prints a message, but don''t override it in the derived classes. Explain what happens. Now override it in one of the derived classes but not the others, and explain what happens. Finally, override it in all the derived classes.

**Solution:**