Part-A

Overview of Java:
History and Evolution, Byte Code, Buzzwords, Object Oriented Programming-Two Paradigms, Abstraction, Principles of Object Oriented Programming, Structure of Java Program, Java Typical Environment. [4L]

Date Types, Operators and Control Statements:
Data Types in Java, Literals, Variables, Type Casting, Arrays, Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, The Conditional Operator, Operator Precedence, Selection Statements, Control Statements, Recursion V/S Iteration. [8L]

Classes, Objects and Methods:
Class Fundamentals, Declaring Object, Assigning Object Reference Variable, Methods, Constructors, Overloading Methods, Objects as Parameters, Returning Objects, Overloading Constructors, This Keyword, Garbage Collection, Finalize () Method, Access Specifiers, Static , Final, Command Line Arguments. [8L]

Inheritance:
Inheritance Basics, Using Super, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Using Final with Inheritance, Constructor in a Derived Class, Object Class. [4L]

Part-B

Package and Interfaces:
Introducing Package, Package Access Protection, Importing Packages, Interfaces - Defining, Implementing, Nesting, Extending, Default Interface Methods. [4L]

Exception Handling:

Exception Handling Fundamentals, Exception Types, Uncaught Exceptions, Try And Catch, Multiple Catch Clauses, Nested Try Statements, Throw, Finally, Built-In Exceptions, Creating Your Own Exception, Chained Exceptions. [4L]

Multithreaded Programming:
The Java Thread Model, Life Cycle Of Thread, The Main Thread, Creating Thread, Creating Multiple Threads, Using Isalive() And Join(), Thread Priorities, Thread Synchronization, Inter Thread Communications, Suspending, Resuming & Stopping Threads. [4L]

I/O, Applets and Event Handling:
I/O Basics, Reading Console Input, Writing Console Output, Printwriter Class, Reading From and Writing to a File, Introduction to Applet, Applet V/S Application Program, Applet Life Cycle, Two Event Handling Mechanisms, Delegation Event Model, Event Classes, Keyevent Class, Sources of Events, Event Listener Interfaces. [8L]

String Handling:
The String Constructors, String Length, Special String Operations, Character Extraction, String Comparison, Searching String, Modifying String, Data Conversion, Changing the Case of Characters, Stringbuffer. [4L]

JAVA

UNIT 1

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.

History of JAVA

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.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

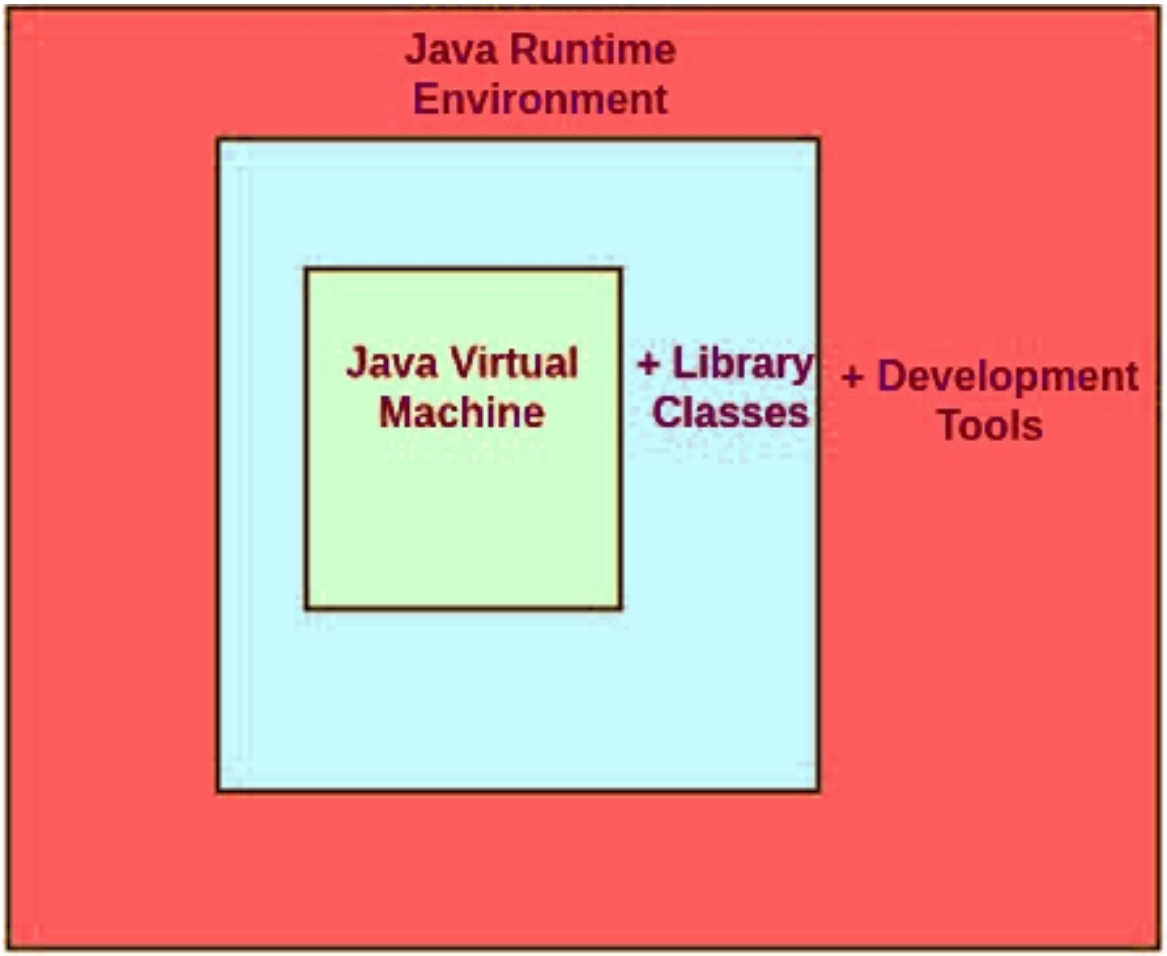
8) Java is an island of Indonesia where first coffee was produced (called java coffee).

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a

JAVA Environment

- Below are the environment settings for both Linux and Windows. JVM, JRE and JDK all three are platform dependent because configuration of each Operating System is different. But, Java is platform independent.
- There are few things which must be clear before setting up the environment
- **JDK**(Java Development Kit) : JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.
- **JRE**(Java Runtime Environment) : JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK. JVM is part of JRE. JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms.



JDK = JRE + Development Tool

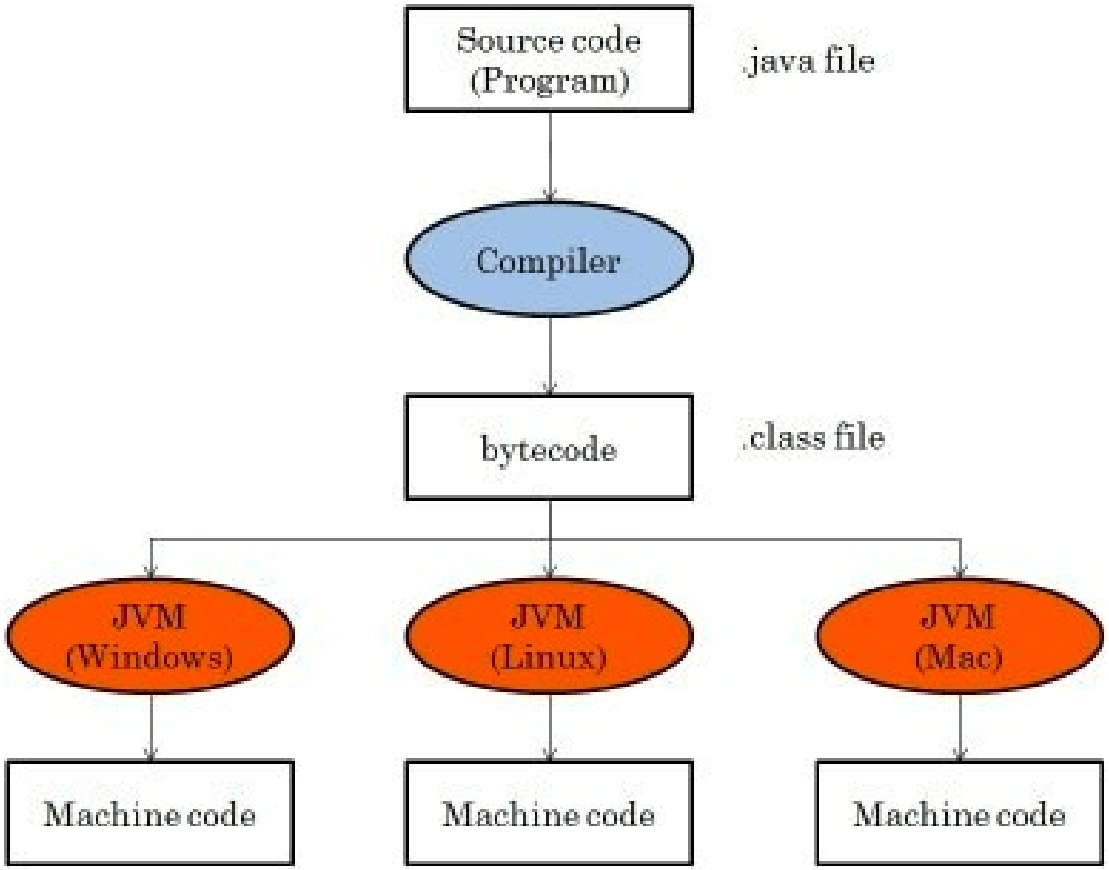
JRE = JVM + Library Classes

Java's Magic: The Bytecode

- Bytecode is the intermediate representation of Java programs.
- As soon as a java program is compiled, java bytecode is generated in the form of a .class file. With the help of java bytecode we achieve platform independence in java.

How does it works?

- When we write a program in Java, firstly, the compiler compiles that program and a bytecode is generated for that piece of code.
- When we wish to run this .class file on any other platform, we can do so. After the first compilation, the bytecode generated is now run by the **Java Virtual Machine(JVM)** and not the processor in consideration. This essentially means that we only need to have basic java installation on any platforms that we want to run our code on.
- Resources required to run the bytecode are made available by the Java Virtual Machine, which calls the processor to allocate the required resources.


Source code
(Program)
.java file
Compiler
bytecode
.class file
JVM
(Windows)
JVM
(Linux)
JVM
(Mac)
Machine code
Machine code
Machine code

Java Buzzwords

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral/portable/ platform independent
- Interpreted + compiled
- High performance
- Distributed

Simple

- Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master.
- If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object oriented features of C++, most programmers have little trouble learning Java.

Secure

- The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

Portability/Platform independence

- Solution is Bytecode.(already discussed, Refer to slides 8,9,10)

Object-Oriented

- Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Robust

- Java is robust because :
- It is a **Strictly typed l**anguage. Some kind of explicit type conversion would be necessary if the data types are not related which in other languages is done implicitly.

```
a  = 9
b = "9"
c = concatenate(a, b)   // produces "99"
d = add(a, b)           // produces 18
```

In a **strongly typed language**, on the other hand, the last two statements would raise type exceptions. To avoid these exceptions, some kind of explicit type conversion would be necessary, like this.

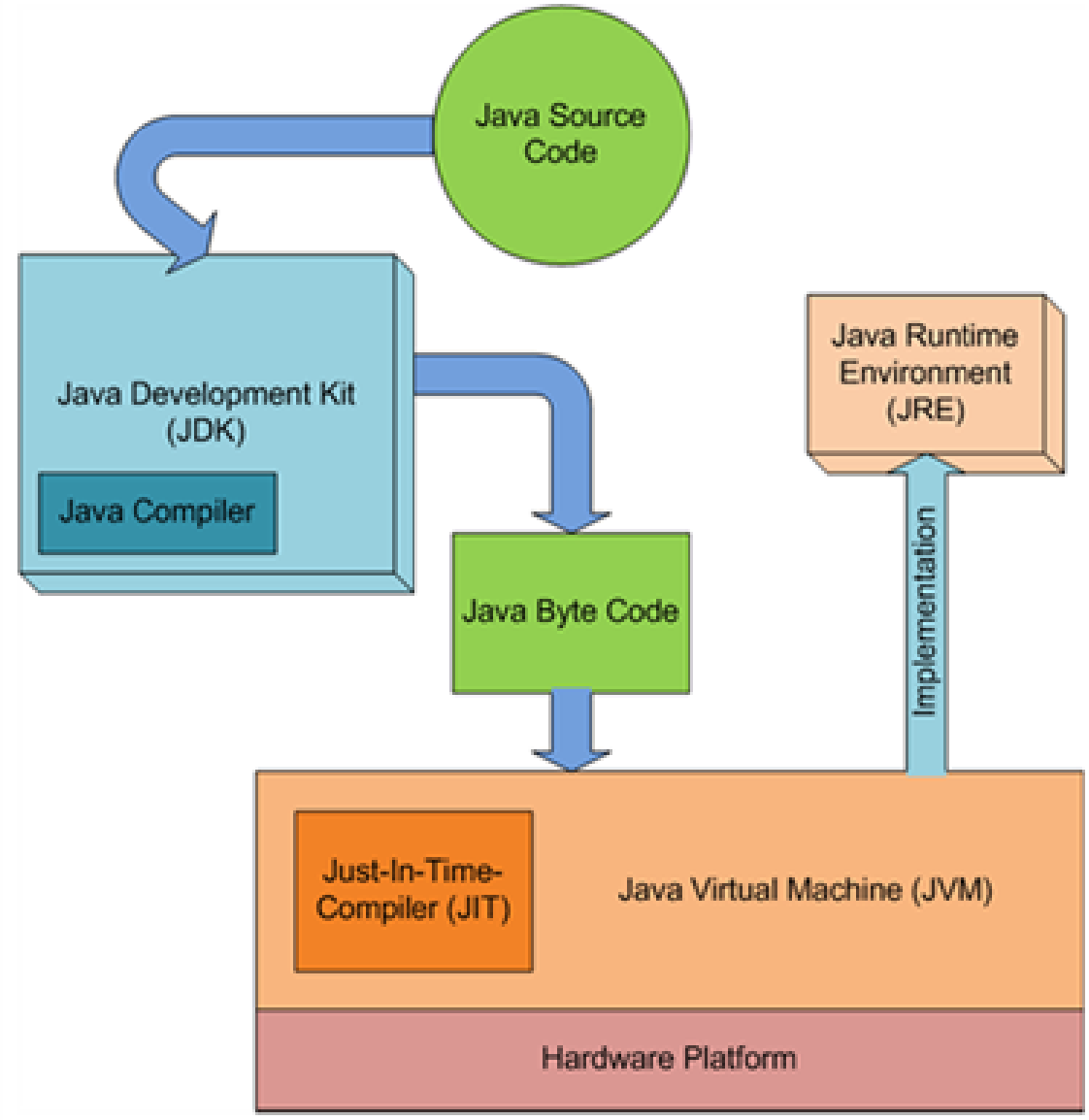
```
a  = 9
b = "9"
c = concatenate(  str(a),  b)
d = add(a,  int(b)  )
```

Multithreaded

- **Multithreading in java** is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.

High Performance

- As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. 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 bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a **just-in-time(JIT)** compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.


Java Source Code
Java Development Kit (JDK)
Java Compiler
Java Byte Code
Java Runtime Environment (JRE)
Implementation
Just-In-Time-Compiler (JIT)
Java Virtual Machine (JVM)
Hardware Platform

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.

Object-Oriented Programming

- Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.
- **Simula** is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.
- **Smalltalk** is considered the first truly object-oriented programming language.
- The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.
- The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

OOP: Two paradigms

- Procedure oriented
- Object oriented
- Difference?

Object-Oriented Programming

- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:
- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.
- **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

- *Collection of objects* is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.
- Class doesn't consume any space.

Inheritance

- *When one object acquires all the properties and behaviors of a parent object,* it is known as inheritance.
- It provides code reusability.
- It is used to achieve runtime polymorphism.

Polymorphism

- If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
- In Java, we use method overloading and method overriding to achieve polymorphism.
- Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

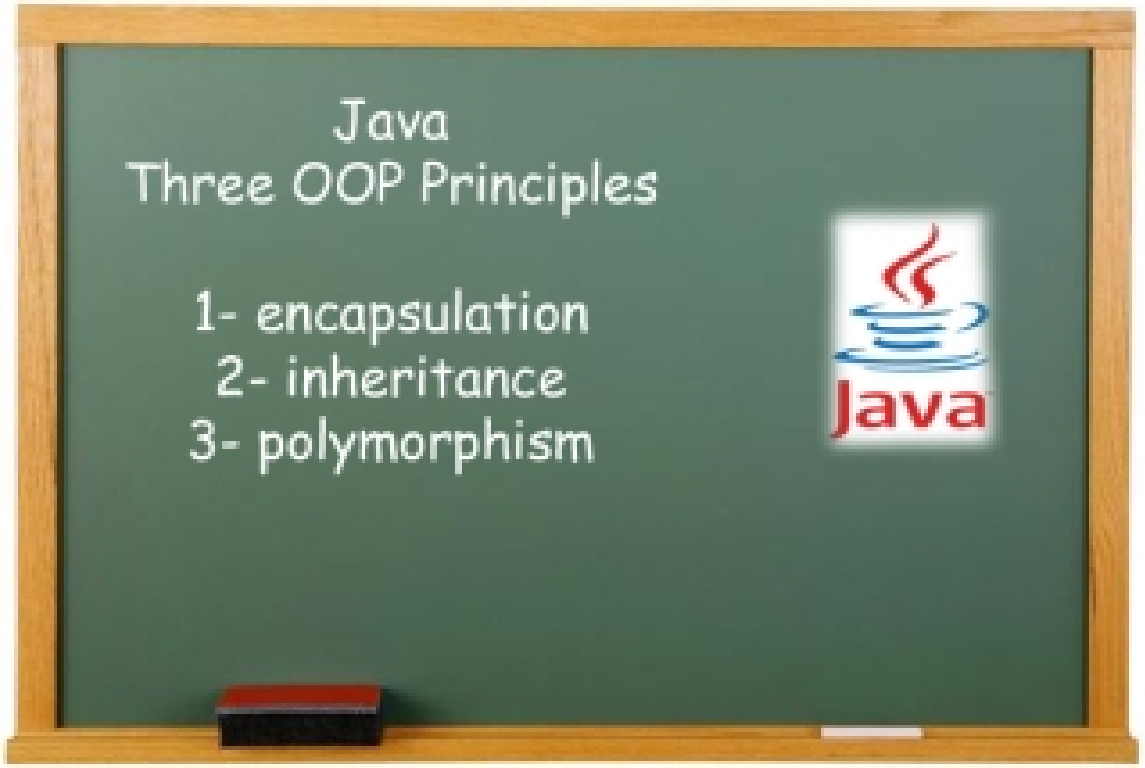
Abstraction

- *Hiding internal details and showing functionality* is known as abstraction. For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

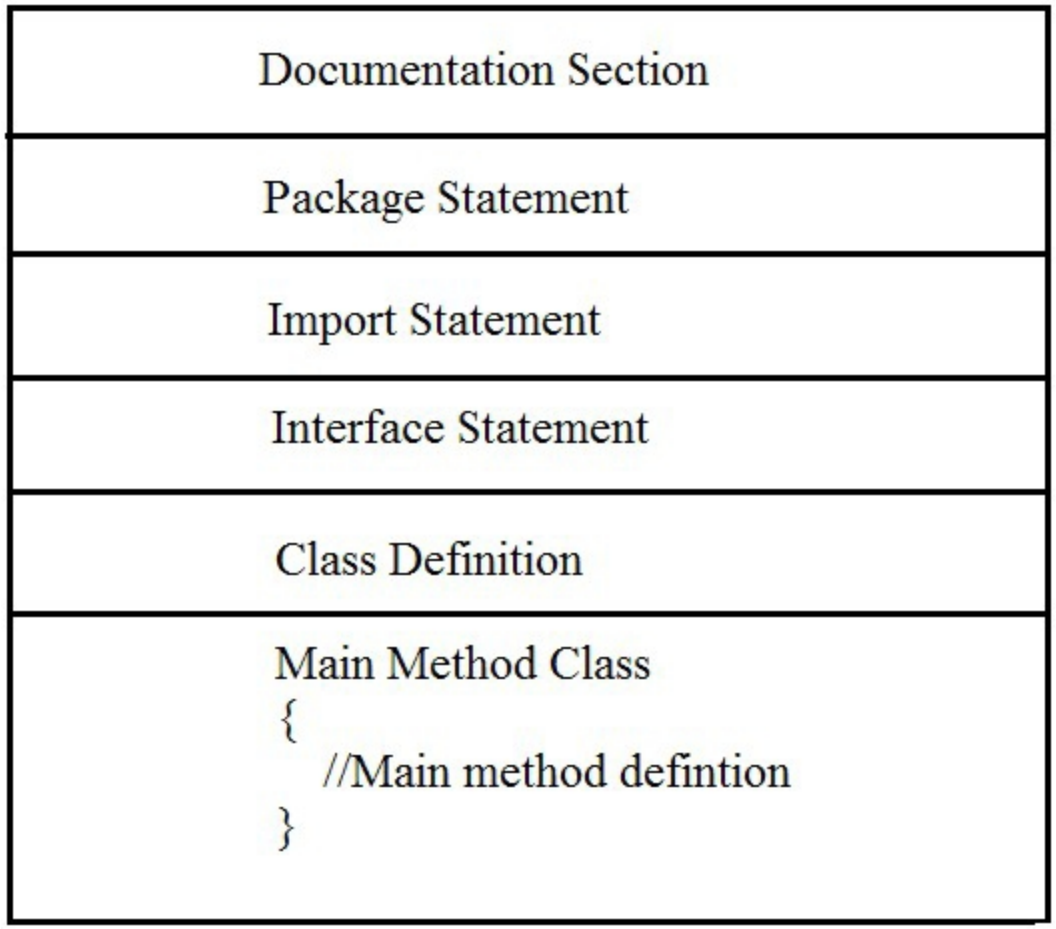
Encapsulation

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation*. For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

3 OOP principles



JAVA: Program structure



JAVA: Program structure

1. Documentation Section:

- It includes the comments that improve the readability of the program.
- A comment is a non-executable statement that helps to read and understand a program especially when your programs get more complex. It is simply a message that exists only for the programmer and is ignored by the compiler.
- A good program should include comments that describe the purpose of the program, author name, date and time of program creation. This section is optional and comments may appear anywhere in the program.

Package Statement

- Java allows you to group classes in a collection known as package. A package statement includes a statement that provides a package declaration. It must appear as the first statement in the source code file before any class or interface declaration. This statement is optional. For example: Suppose you write the following package declaration as the first statement in the source code file.
- package employee;
- This statement declares that all classes and interfaces defined in this source file are part of the employee package. Only one package declaration can appear in the source file.

Import Statement

- Java contains many predefined classes that are stored into packages. In order to refer these standard predefined classes in your program, you need to use fully qualified name (i.e. Packagename.className). But this is a very tedious task as one need to retype the package path name along with the classname. So a better alternative is to use an import statement.
- An import statement is used for referring classes that are declared in other packages. The import statement is written after a package statement but before any class definition. You can import a specific class or all the classes of the package. For example : If you want to import Date class of java.util package using import statement then write

import java.util.Date;

Interface Section

- In the interface section, we specify the interfaces. An interface is similar to a class but contains only constants and method declarations.
- Interfaces cannot be instantiated. They can only be implemented by classes or extended by other interfaces. It is an optional section and **is used when we wish to implement multiple inheritance feature in the program.**
- **interface** stack {
  void push(int item); // Insert item into stack
  int pop(); // Delete an item from stack
  }

Class Section

- The Class section describes the information about user-defined classes present in the program. A class is a collection of fields (data variables) and methods that operate on the fields.
- Every program in Java consists of at least one class, the one that contains the main method. The main () method which is from where the execution of program actually starts and follow the statements in the order specified.
- The main method can create objects, evaluate expressions, and invoke other methods and much more. On reaching the end of main, the program terminates and control passes back to the operating system.
- The class section is mandatory.
- After discussing the structure of programs in Java, we shall now discuss a program that displays a string Hello Java on the screen.

Sample program

- // Program to display message on the screen

```
Package hello;    // optional
 class HelloJava
 {
public static void main(String args[]){
   System.out.println("Hello Java");
}
}
```

Public

- **Public:** It is an *Access modifier*, which specifies from where and who can access the method.
- Making the *main()* method public makes it globally available.
- It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

Static

- **Static:** It is a *keyword* which is when associated with a method, makes it a class related method.
- The *main()* method is static so that JVM can invoke it without instantiating(without creating objects) the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the *main()* method by the JVM

Void

- **Void:** It is a keyword and used to specify that a method doesn't return anything.
- As *main()*method doesn't return anything, its return type is *void*.
- As soon as the *main()* method terminates, the java program terminates too. Hence, it doesn't make any sense to return from *main()* method as JVM can't do anything with the return value of it.

Main()

- **main:** It is the name of Java main method. It is the identifier that the JVM looks for as the starting point of the java program. It's not a keyword.

String[] args

- **String[] args:** It stores Java *command line arguments* and is an array of type *java.lang.String*class.
- Here, the name of the String array is *args* but it is not fixed and user can use any name in place of it.
- Javac a.java
- Java a this is d3it

System.out.println

- System is a predefined class that provides access to the system which is introduced inside java lang.package.
  System class is a default class.
- Out is a data type of print stream class that is connected to the console.
- Println ( ) displays the string which is passed to it.
- Method System.out.println displays its argument in the command window followed by a new-line character to position the output cursor to the beginning of the next line.

Whitespace

- Java is a free-form language. This means that you do not need to follow any special indentation rules.
- For example, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator.
- In java, whitespace is a space, tab, or new line.

Identifiers

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers or the underscore and dollar sign characters.
- They must not begin with a number
- Again, java is case-sensitive, so **VALUE** is a different identifier the **Value.**
- Some examples of valid identifiers are:

AvgTemp count a4 $test thi s_is_ok

- Invalid variable names include:

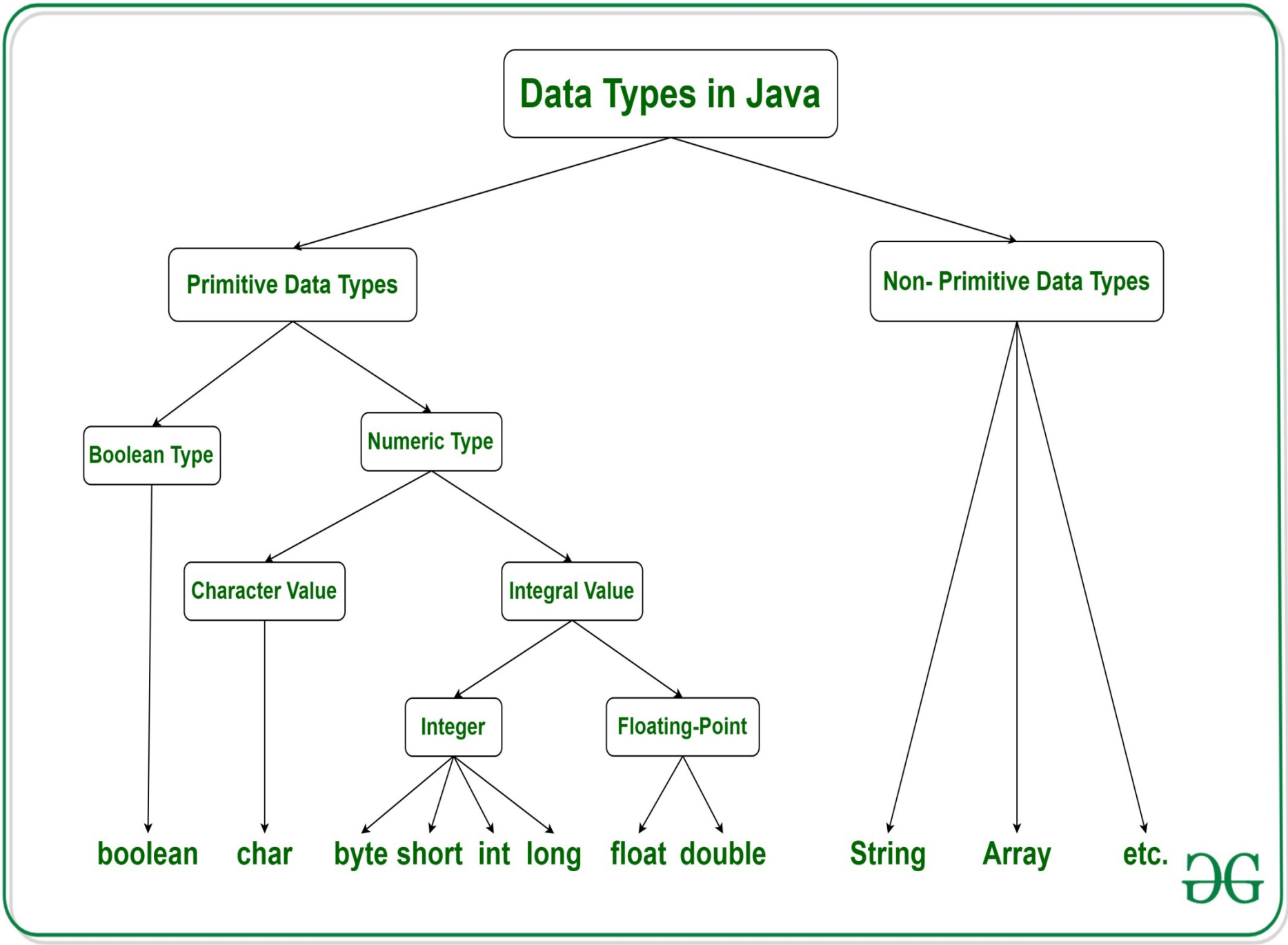
2count high-temp Not/ok

Comments

- There are two types of comments defined by java.
- single-line and multiline.
- The single line comment begins with a //.
- The multiline comment begins with /* and ends with */.

JAVA

UNIT 2(A)


Data Types in Java
Primitive Data Types
Non- Primitive Data Types
Boolean Type
Numeric Type
Character Value
Integral Value
Integer
Floating-Point
boolean
char
byte
short
int
long
float
double
String
Array
etc.

Data types

- Two types: Primitive and Non- Primitive.
- **Primitive** can be categorized into four groups:
- *Integers*
- *Characters*
- *Floating-point numbers*
- *Boolean*

INTEGERS

- There are four **integer** types in Java- ***byte*, *short*, *int* and *long***.
- ***Byte***: It has the least size range (8-bit) integer. Also, it is used when working with raw binary data.
  Declaration example: byte a;
- ***Short***: It is bigger than byte but shorter than int and long. It is 16-bit integer. It can be used when dealing with small numeric values.
  Declaration example: short a;
- ***Int***: It is the most commonly used integer data type which is bigger than a byte and short but smaller than long. It is 32-bit integer.
  Declaration example: int a;
- ***Long***: It is the largest integer data type and is useful when dealing with big, whole numbers. It is 64-bit integer.
  Declaration example: long a;

Range of numeric data types

Type	Size	Range
byte	8 bits	-128 .. 127
short	16 bits	-32,768 .. 32,767
int	32 bits	-2,147,483,648 .. 2,147,483,647
long	64 bits	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
float	32 bits	$3.40282347 \times 10^{38}$, $1.40239846 \times 10^{-45}$
double	64 bits	$1.7976931348623157 \times 10^{308}$, $4.9406564584124654 \times 10^{-324}$

CHARACTERS

- The char data type is a single 16-bit Unicode character.
- **Declaration example**: char ch='x';
- **char** can also be used to perform arithmetic operations.
- For example, **ch++** is a valid statement in Java which increments the *ASCII(Unicode)* value of the character stored.

FLOATING-POINT NUMBERS

- It is the *real* number used where the user requires fractional precision like square roots.
- ***Float***: Float type number have *single precision* and is faster on some processors due to less storage size, although it might become imprecise when dealing with very large or very small numbers. It is a 32 bit number.
- **Declaration example**:
  float a;
  float b=10.9f;
  In the case of initializing float variables the value must be followed by a letter f.

FLOATING-POINT NUMBERS

- ***Double***: Double type number has *double precision* and is faster on modern processors and is optimized for high-speed mathematical calculations.
- It is useful in cases where we need high accuracy or when we have to deal with large-valued numbers. All transcendental mathematical functions like sin(), cos(), etc return double values in Java. .
- It is a 64 bit number.
- **Declaration example**:
  double a;

BOOLEAN

- **Boolean** is another primitive data type in Java which deals with logical values- *true* or *false*. All relational operators return a Boolean value.
- This Boolean value is used to control the *if statement* and the *conditional loop statement*. It can have either of the two values, i.e., either *true* or *false*.
- **Declaration example**: boolean a = true;

Non-primitive data types

- *The non-primitive Java data types are created by the programmer during the coding process, they are known as the "reference variables" or "object variables" as they refer to a location where data is stored.*
- The **non-primitive** data types include
- Classes
- Arrays
- Strings
- Interfaces :An **interface** is like a dashboard or control panel for a class. It has the buttons, but the function is elsewhere.

Type conversion/casting

- When you assign value of one data type to another, the two types might not be compatible with each other.
- If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly.

Type conversion/casting

- Widening Casting (**Implicit**) – Automatic Type Conversion
- Narrowing Casting (**Explicit**) – Need Explicit Conversion

Type conversion/casting

Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

Automatic Type Conversion/Widening

- The two data types are **compatible.**(e.g. int and float are compatible, char and boolean are not compatible with each other.)
- When we assign value of a **smaller data type to a bigger data type**.(int can be converted into double but reverse is not possible. It needs to be done explicitly)

Example

```java
class Test
{
    public static void main(String[] args)
    {
        int i = 100;

        //automatic type conversion
        long l = i;

        //automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Output:

```
Int value 100
Long value 100
Float value 100.0
```

Explicit Conversion/Narrowing

- if we want to assign a value of larger data type to a smaller data type we perform explicit type casting.
- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

Example

```java
//Java program to illustrate explicit type conversion
class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Long value "+l);

        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

Output:

```
Double value 100.04
Long value 100
Int value 100
```

JAVA

UNIT 2(B)

```java
//Java program to illustrate Conversion of int and double to byte
class Test
{
    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("Conversion of int to byte.");

        //i%256
        b = (byte) i;
        System.out.println("i = " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");

        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b= " + b);
    }
}

```

Output:

```
Conversion of int to byte.
i = 257 b = 1

Conversion of double to byte.
d = 323.142 b = 67
```

Type promotion

- While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:
- Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
- If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

Example

```java
//Java program to illustrate Type promotion in Expressions
class Test
{
    public static void main(String args[])
    {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;

        // The Expression
        double result = (f * b) + (i / c) - (d * s);

        //Result after all the promotions are done
        System.out.println("result = " + result);
    }
}
```

Output:

```
Result = 626.7784146484375
```

Arrays

- **Java array** is an object which contains elements of a similar data type.
- Additionally, The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on

Array Declaration

- An array declaration has two components: the type and the name. *type* declares the element type of the array.
- The element type determines the data type of each element that comprises the array. Like array of int type. Int arrayA[];
- we can also create an array of other primitive data types like char, float, double etc. or user defined data type(objects of a class)

Array Initialization

- int intArray[];
  intArray = new int[20];

or

- int[] intArray = new int[20];

Types of Array in java

- There are two types of array.
- Single Dimensional Array
- Multidimensional Array

One dimensional array

```java
// Java program to illustrate creating an array
// of integers,  puts some values in the array,
// and prints each value to standard output.

class GFG
{
    public static void main (String[] args)
    {
      // declares an Array of integers.
      int[] arr;

      // allocating memory for 5 integers.
      arr = new int[5];

      // initialize the first elements of the array
      arr[0] = 10;

      // initialize the second elements of the array
      arr[1] = 20;

      //so on...
      arr[2] = 30;
      arr[3] = 40;
      arr[4] = 50;

      // accessing the elements of the specified array
      for (int i = 0; i < arr.length; i++)
         System.out.println("Element at index " + i +
                                        " : "+ arr[i]);
    }
```

Multidimensional Arrays

- Multidimensional arrays are **arrays of arrays** with each element of the array holding the reference of other array. These are also known as Jagged Arrays.
- A multidimensional array is created by appending one set of square brackets ([]) per dimension.
- Examples:
- Int intArray [][] = new int[10][20]; //a 2D array
- int intArray [][][] = new int[10][20][10]; //a 3D array

Multidimensional Arrays

```java
class multiDimensional
{
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[][] = { {2,7,9},{3,6,1},{7,4,2} };

        // printing 2D array
        for (int i=0; i< 3 ; i++)
        {
            for (int j=0; j < 3 ; j++)
                System.out.print(arr[i][j] + " ");

            System.out.println();
        }
    }
}
```

Output:

```
2 7 9
3 6 1
7 4 2
```

Operators

- https://www.geeksforgeeks.org/operators-in-java/
- 1000: 8 /4
- >>2
- 0010: 2

Operator Precedence

- https://introcs.cs.princeton.edu/java/11precedence/

Selection and Jump statements

- https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/

Iteration statements

- https://www.geeksforgeeks.org/loops-in-java/

Recursion vs Iteration

PROPERTY	RECURSION	ITERATION
Definition	Function calls itself.	A set of instructions repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code Size	Smaller code size	Larger Code Size.
Time Complexity	Very high(generally exponential) time complexity.	Relatively lower time complexity(generally polynomial-logarithmic).

```java
// Java program to find factorial of given number
class GFG {

    // ----- Recursion -----
    // method to find factorial of given number
    static int factorialUsingRecursion(int n)
    {
        if (n == 0)
            return 1;

        // recursion call
        return n * factorialUsingRecursion(n - 1);
    }

    // ----- Iteration -----
    // Method to find the factorial of a given number
    static int factorialUsingIteration(int n)
    {
        int res = 1, i;

        // using iteration
        for (i = 2; i <= n; i++)
            res *= i;

        return res;
    }

    // Driver method
    public static void main(String[] args)
    {
        int num = 5;
        System.out.println("Factorial of " + num
                           + " using Recursion is: "
                           + factorialUsingRecursion(5));

        System.out.println("Factorial of " + num
                           + " using Iteration is: "
                           + factorialUsingIteration(5));
    }
```

Java

UNIT 3(a)

Class

- A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:
- **Modifiers** : A class can be public or has default access
- **Class name:** The name should begin with a initial letter (capitalized by convention).
- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.

Object declaration

- Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way.

Object declaration

```java
// Java program to illustrate creation of Object
// using new keyword
public class NewKeywordExample
{
    String name = "GeeksForGeeks";
    public static void main(String[] args)
    {
        // Here we are creating Object of
        // NewKeywordExample using new keyword
        NewKeywordExample obj = new NewKeywordExample();
        System.out.println(obj.name);
    }
}
```

Output:

```
GeeksForGeeks
```

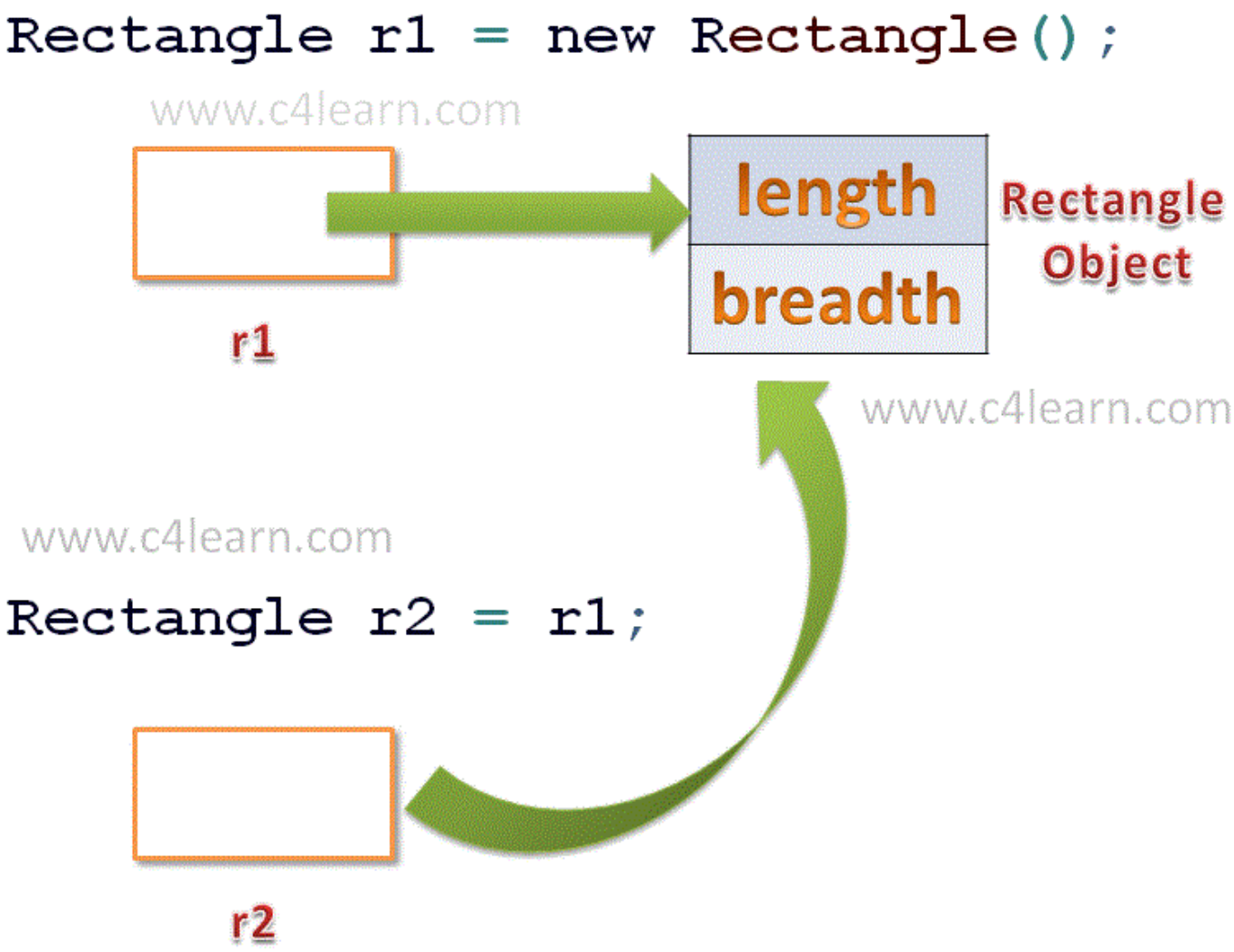
Object reference variable

- We can assign value of reference variable to another reference variable.
- Reference Variable is used to store the address of the variable.
- Assigning Reference will not create distinct copies of Objects.
- All reference variables are referring to same Object.

Object reference variable

- Synytax:
  Rectangle r1 = new Rectangle();
  Rectangle r2 = r1;
- r1 is reference variable which contain the address of Actual Rectangle Object.
- r2 is another reference variable
- r2 is initialized with r1 means – "**r1 and r2**" both are referring same object , thus it does not create duplicate object , nor does it allocate extra memory.

Object reference variable



Object reference variable

Assigning Object Reference Variables does not-

- Create Distinct Objects.
- Allocate Memory
- Create duplicate Copy

Object reference variable

- Example:

```
class Rectangle
 { double length;
double breadth;
public static void main(String args[])
{ Rectangle r1 = new Rectangle();
Rectangle r2 = r1;
r1.length = 10;
r2.length = 20;
System.out.println("Value of R1's Length : " + r1.length);
System.out.println("Value of R2's Length : " + r2.length); } }
```

Output: Value of R1's Length : 20.0

Value of R2's Length : 20.0

Methods

- A method is a collection of statements that perform some specific task and return the result to the caller.
- Methods allow us to **reuse** the code without retyping the code.
- Methods are **time savers** and help us to **reuse** the code without retyping the code.

Method Declaration

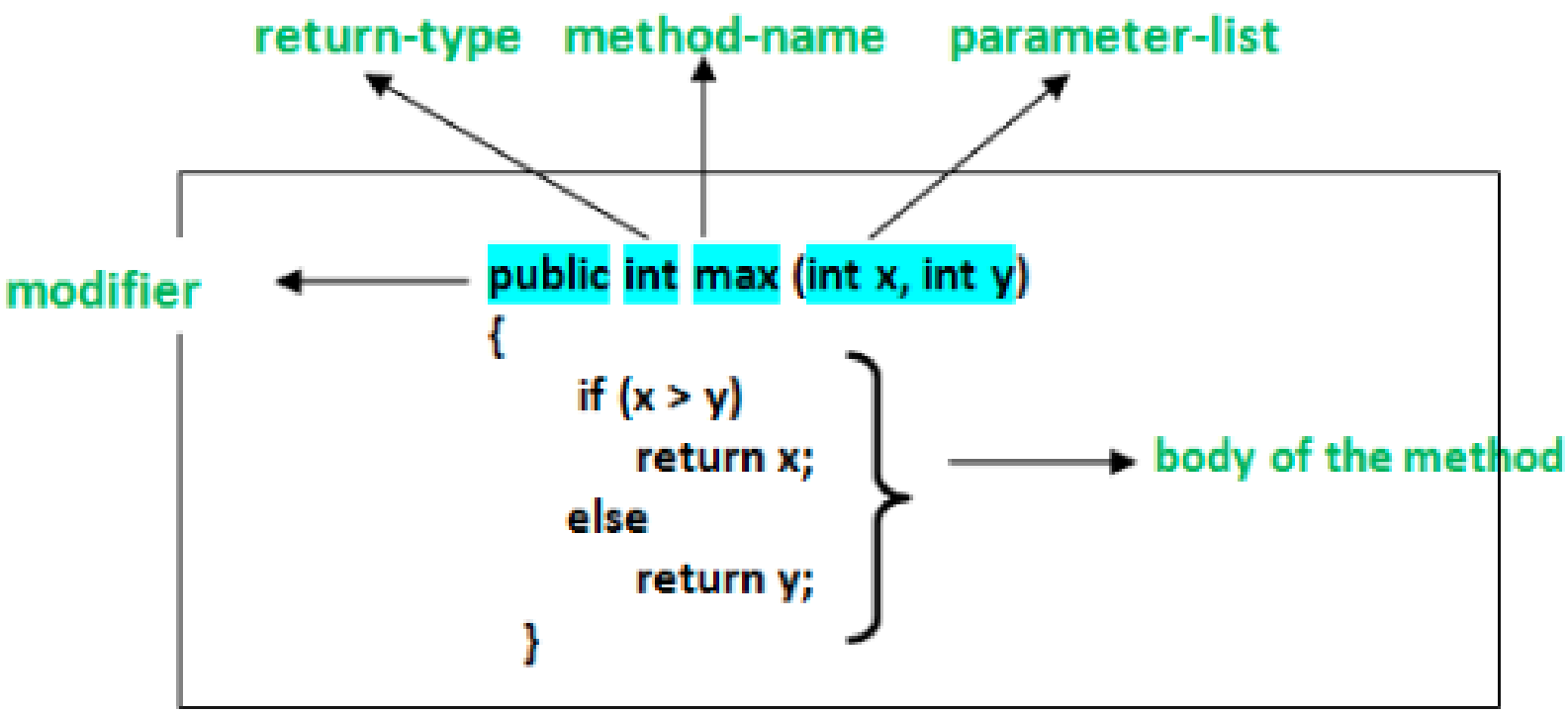
- In general, method declarations has six components :

1. **Modifier**–: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
   - public: accessible in all class in your application.
   - protected: accessible within the class in which it is defined and in its **subclass(es)**
   - private: accessible only within the class in which it is defined.
   - default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.

Method Declaration

2. **The return type** : The data type of the value returned by the method or void if does not return a value.
3. **Method Name** : the rules for field names apply to method names as well, but the convention is a little different.
4. **Parameter list** : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
5. **Method body** : it is enclosed between braces. The code you need to be executed to perform your intended operations.

Method Declaration



Method overloading

- Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both.
- Overloading is related to compile time (or static) polymorphism.

```java
public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output :

```
30
60
31.0
```

Constructor

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Rules for creating Java constructor

- There are two rules defined for the constructor:

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

- There are two types of constructors in Java:
- Default constructor (no-arg constructor)
- Parameterized constructor

Default constructor

- A constructor that has no parameter is known as default constructor.
- If we don't define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.

Default constructor

```java
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

Test it Now

Output:

```
Bike is created
```

Parameterized Constructor

- A constructor that has parameters is known as parameterized constructor.
- The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.
- If we want to initialize fields of the class with your own values, then use a parameterized constructor.

```
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Test it Now

Output:

```
111 Karan
222 Aryan
```

JAVA

Unit 3(b)

Constructor overloading

- https://www.geeksforgeeks.org/constructor-overloading-java/

Passing Objects in Java

- Although Java is strictly pass by value, the precise effect differs between whether a primitive type or a reference type is passed.
- When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference. Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

Passing Objects in Java

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.

```
class ObjectPassDemo
{
    int a, b;

    ObjectPassDemo(int i, int j)
    {
        a = i;
        b = j;
    }

    // return true if o is equal to the invoking
    // object notice an object is passed as an
    // argument to method
    boolean equalTo(ObjectPassDemo o)
    {
        return (o.a == a && o.b == b);
    }
}

// Driver class
public class Test
{
    public static void main(String args[])
    {
        ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
        ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);

        System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

Output:

```
ob1 == ob2: true
ob1 == ob3: false
```

Java strictly Pass by value?

- https://www.infoworld.com/article/3512039/does-java-pass-by-reference-or-pass-by-value.html#:~:text=Java%20always%20passes%20parameter%20variables,is%20passed%20a%20new%20value.
- https://www.geeksforgeeks.org/g-fact-31-java-is-strictly-pass-by-value/

Access Control in Java

- Access control specify what parts of a program can access the members of a class and so prevent misuse.
- Java's access modifiers are public, private, and protected. Java also defines a default access level.

Access Control in Java

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

THIS Keyword

- https://www.javatpoint.com/this-keyword

Other uses of ‘this’

- https://www.geeksforgeeks.org/this-reference-in-java/

Garbage collection

- https://www.guru99.com/java-garbage-collection.html

finalize() method

- It is a **method** that the **Garbage Collector** always calls just **before** the deletion/destroying the object which is eligible for Garbage Collection, so as to perform **clean-up activity**.
- Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation. Remember it is **not** a reserved keyword.
- Once finalize method completes immediately Garbage Collector de object

Syntax

```
protected void finalize()
```

finalize() method

```
public class JavafinalizeExample1 {
    public static void main(String[] args)
   {
      JavafinalizeExample1 obj = new JavafinalizeExample1();
      System.out.println(obj.hashCode());
      obj = null;
      // calling garbage collector
      System.gc();
      System.out.println("end of garbage collection");

   }
   @Override
   protected void finalize()
   {
      System.out.println("finalize method called");
   }
}
```

Output

```
2018699554
end of garbage collection
finalize method called
```

JAVA

UNIT 4(A)

Inheritance

- Inheritance is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Moreover, you can add new methods and fields in your current class also.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

How to use inheritance

The keyword used for inheritance is **extends**.

Syntax :

```
class derived-class extends base-class
{
   //methods and fields
}
```

Inheritance: Sample program

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

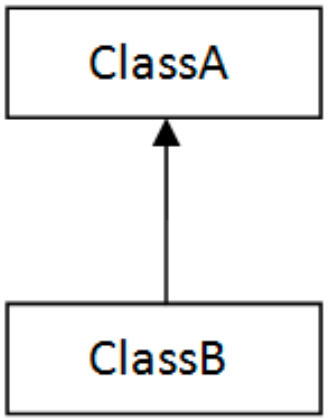
Test it Now

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

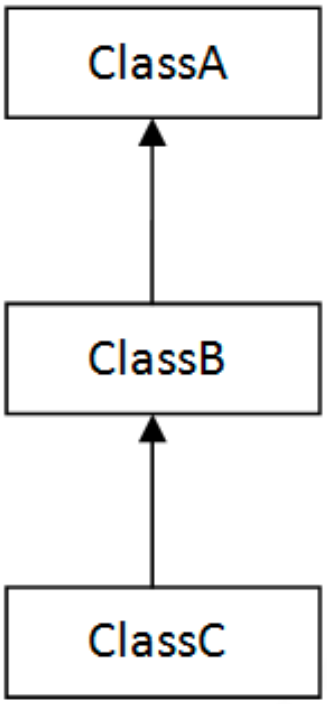
Types of inheritance

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

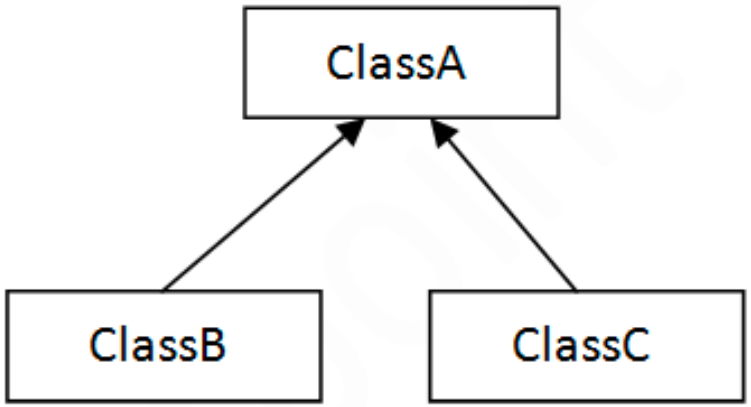
Types of inheritance



1) Single



2) Multilevel



3) Hierarchical

Single Inheritance

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

Super keyword

- The **super** keyword in java is a reference variable that is used to refer parent class objects.
- The keyword "super" came into the picture with the concept of Inheritance.

Use of super

1) To access the data members of parent class when both parent and child class have member with same name
2) To access the method of parent class when child class has overridden that method.
3) To access the parent class constructor.

To access the variables of parent class

- When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

```
//Parent class or Superclass or base class
class Superclass
{
   int num = 100;
}
//Child class or subclass or derived class
class Subclass extends Superclass
{
   /* The same variable num is declared in the Subclass
    * which is already present in the Superclass
    */
    int num = 110;
    void printNumber(){
        System.out.println(num);
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
    }
}
```

Output:

110

with super

```
class Superclass
{
   int num = 100;
}
class Subclass extends Superclass
{
   int num = 110;
   void printNumber(){
        /* Note that instead of writing num we are
         * writing super.num in the print statement
         * this refers to the num variable of Superclass
         */
        System.out.println(super.num);
   }
   public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
   }
}
```

Output:

100

Java

UNIT 4(b)

To access method of parent class with super

- The super keyword can also be used to invoke parent class method.
- It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Parentclass
{
   //Overridden method
   void display(){
        System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
   void display(){
        System.out.println("Child class method");
   }
   void printMsg(){
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
   }
   public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
   }
}
```

Output:

```
Child class method
Parent class method
```

To access constructor of parent class with super

- When we create the object of sub class, the new keyword invokes the **constructor** of child class, which implicitly invokes the constructor of parent class. So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed.
- It happens because compiler itself adds super()(this invokes the no-arg constructor of parent class) as the first statement in the constructor of child class.

To access constructor of parent class with super

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Test it Now

Output:

```
animal is created
dog is created
```

Method overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.
- It is also called **Dynamic Method**

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Method overriding

```java
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show()
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        // If a Parent type reference refers
        // to a Parent object, then Parent's
        // show is called
        Parent obj1 = new Parent();
        obj1.show();

        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

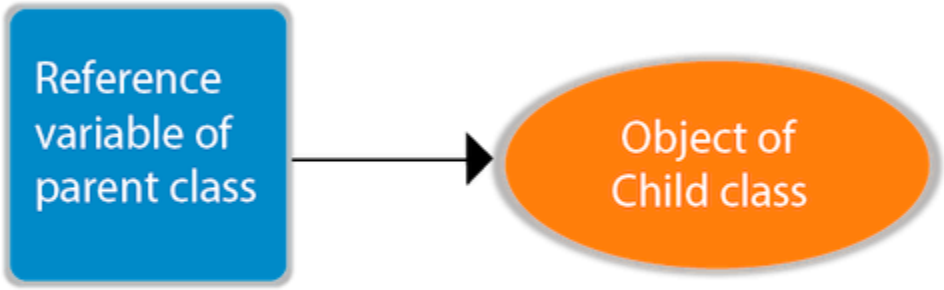
Dynamic Method Dispatch

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:




```
class A{}
class B extends A{}
```

```
A a=new B();//upcasting
```

Dynamic Method Dispatch

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}

  public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Test it Now

Output:

```
running safely with 60km.
```

Abstract class

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- Why abstract class?.....**To achieve abstraction**.
- There are two ways to achieve abstraction in java:
- Abstract class (0 to 100%)
- Interface (100%)

Abstract class

- A class which is declared as abstract is known as an **abstract class**.
- It can have abstract and non-abstract methods.
- It needs to be extended and its method implemented.
- It cannot be instantiated but we can create reference of abstract class.
- An abstract class must be declared with an abstract keyword.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstract Method in Java

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- **Syntax:**

  abstract void printStatus();

// no method body

Abstract class

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Test it Now

```
running safely
```

JAVA

UNIT 5

Package

- **Package** in Java is a mechanism to **encapsulate** a group of classes, sub packages and interfaces. Packages are used for:
- **Preventing naming conflicts**. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making **searching/locating** and usage of classes, interfaces, enumerations and annotations easier

Package

- Packages can be considered as **data encapsulation** (or data-hiding).
- All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program.
- A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.
- We can reuse existing classes from the packages as many time as we need it in our program.

Package

- **Providing controlled access**: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

Access Protection in Packages

- Packages are meant for encapsulating, it works as containers for classes and other sub packages. Class acts as containers for data and methods. There are four categories, provided by Java regarding the visibility of the class members between classes and packages:
- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

Access Protection in Packages

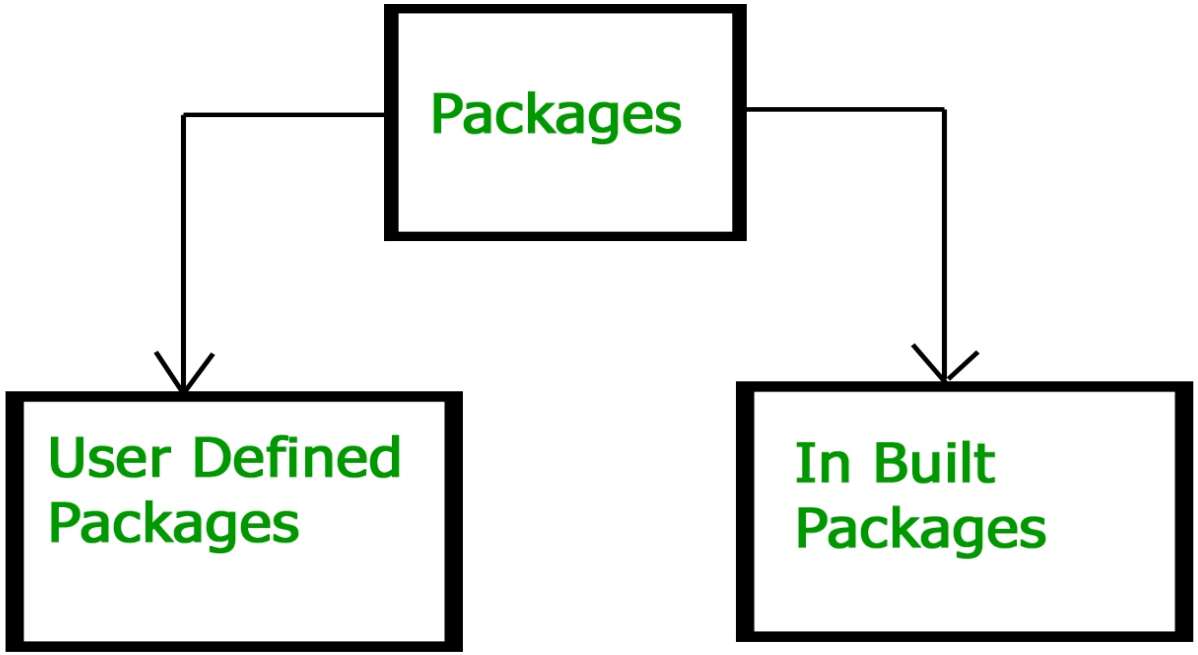
	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
same package subclass	No	Yes	Yes	Yes
same package non - subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package mon-subclass	No	No	No	Yes

Importing packages

```
// import the Vector class from util package.
import java.util.vector;

// import all the classes from util package
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

Types



Built-in Packages

- These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:
  1) **java.lang:** Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
  2) **java.io:** Contains classed for supporting input / output operations.
  3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
  4) **java.applet:** Contains classes for creating Applets.
  5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
  6) **java.net:** Contain classes for supporting networking

User-defined packages

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Interface

- Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).
- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

Interface

- To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default.
- A class that implement interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Note

- Why use interfaces when we have abstract classes?
- The reason is, abstract classes may contain non-final and non-static variables, whereas variables in interface are final, public and static.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Why do we use interface ?

- It is used to achieve **total abstraction**.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve **multiple inheritance** .

JAVA

Unit 5(b)

Interface declaration

```
interface <interface_name> {

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

```
// A simple interface
interface Player
{
    final int id = 10;
    int move();
}
```

Interface implementation

```java
import java.io.*;

// A simple interface
interface in1
{
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}

// A class that implements interface.
class testClass implements in1
{
    // Implementing the capabilities of
    // interface.
    public void display()
    {
        System.out.println("Geek");
    }

    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
        System.out.println(a);
    }
}
```

Output:

```
Geek
10
```

Nested Interface

- https://www.geeksforgeeks.org/interface-nested-class-another-interface/
- **Note:** Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

Default interface methods

- Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface.
- To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

Default interface methods

```java
// A simple program to Test Interface default
// methods in java
interface TestInterface
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
      System.out.println("Default Method Executed");
    }
}

class TestClass implements TestInterface
{
    // implementation of square abstract method
    public void square(int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);

        // default method executed
        d.show();
    }
}
```

Output:

```
16
Default Method Executed
```

JAVA

UNIT 6

Exception handling

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- **What is exception:** An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time, that disrupts the normal flow of the program's instructions.

Error vs Exception

- **Error:** An Error indicates serious problem that a reasonable application should not try to catch. It is not recoverable.
  Exception: Exception indicates conditions that a reasonable application might try to catch. It is recoverable

Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:
- Checked Exception
- Unchecked Exception
- Error

Types of Java Exceptions

1) Checked Exception: Checked exceptions are checked at compile-time. e.g. IOException, SQLException etc.

2) Unchecked Exception: Unchecked exceptions are not checked at compile-time, but they are checked at runtime. e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

3) Error: Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Exception Example

```java
// java program to demonstrate
// need of try-catch clause

class GFG {
    public static void main (String[] args) {

        // array of size 4.
        int[] arr = new int[4];

        // this statement causes an exception
        int i = arr[4];

        // the following statement will never execute
        System.out.println("Hi, I want to execute");
    }
}
```

Output :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at GFG.main(GFG.java:9)
```

Java Exception Handling Example

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Java Multi-catch block

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.
- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example

```
public class MultipleCatchBlock1 {

    public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

Output

```
Arithmetic Exception occurs
rest of the code
```

JAVA

Unit 6(b)

Nested try block

- The try block within a try block is known as nested try block in java.

```java
class NestedTry {

    // main method
    public static void main(String args[])
    {
        // Main try block
        try {

            // initializing array
            int a[] = { 1, 2, 3, 4, 5 };

            // trying to print element at index 5
            System.out.println(a[5]);

            // try-block2 inside another try block
            try {

                // performing division by zero
                int x = a[2] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
        }
    }
    // end of main method
}
```

Output:

```
ArrayIndexOutOfBoundsException
Element at such index does not exists
```

Java throw keyword

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

Java throw keyword

```
throw exception;
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error);
```

Java throw keyword

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
  }
}
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

JAVA throw keyword example

```
1  class myexception extends Exception
2  {
3      myexception(String s)
4      {
5          super(s);
6      }
7  }
8  class excep1
9  {
10     static void validage(int age) throws myexception
11     {
12         if(age<18)
13         {
14             throw new myexception("Not Valid to Give Vote");
15         }
16         else
17         {
18             System.out.println("Welcome to Vote");
19         }
20     }
21     public static void main(String args[])
22     {
```

Java finally block

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.
- Why use java finally: Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.
- NOTE: For each try block there can be zero or more catch blocks, but only one finally block.

Case 1

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Test it Now

```
Output:5
       finally block is always executed
       rest of the code...
```

Case 2

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Test it Now

```
Output:finally block is always executed
       Exception in thread main java.lang.ArithmeticException:/ by zero
```

built-in exceptions

- https://www.geeksforgeeks.org/built-exceptions-java-examples/

Java Custom Exception

```
class TestCustomException1{

  static void validate(int age)throws InvalidAgeException{
    if(age<18)
     throw new InvalidAgeException("not valid");
    else
     System.out.println("welcome to vote");
  }

  public static void main(String args[]){
    try{
    validate(13);
    }catch(Exception m){System.out.println("Exception occured: "+m);}

    System.out.println("rest of the code...");
  }
}
```

Test it Now

```
Output:Exception occured: InvalidAgeException:not valid
       rest of the code...
```

Chained Exception

- Assignment qs

JAVA

Unit 7

Multithreading

- Multithreading in java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Also, Threads are independent. If there occurs exception in one thread, it doesn't affect other threads.

Multithreading

- The Java language is, at its core, a threaded language. Every Java application runs at least two threads: the **main thread** started by the function:

  public static void main(String args[]);

- and a **garbage collection thread** which deallocates unused pointer memory as needed.

Advantages of Multithreading

- It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- You **can perform many operations together, so it saves time**.
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Limitation of Multithreading

- In the case of multithreading we cant predict the exact order of output because it will vary from system to system or JVM to JVM.

Creating threads

There are two ways to create a thread:

1. By extending Thread class: Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

2. By implementing Runnable interface: The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

Public void run(): is used to perform action for a thread.

Starting a thread

- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:
- A new thread starts.
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

```
Output:thread is running...
```

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

```
Output:thread is running...
```

Commonly used methods of Thread class

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

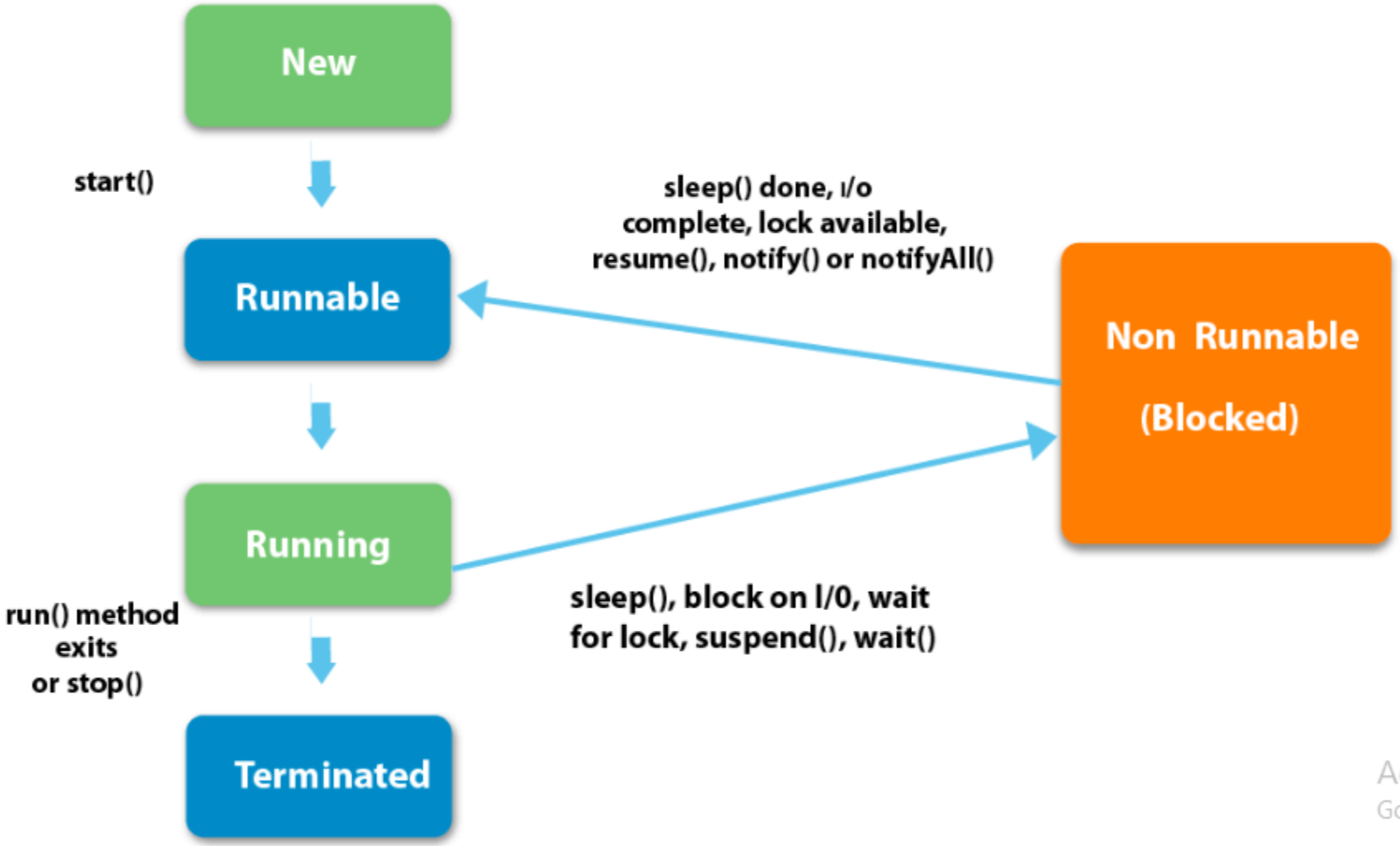
Multithreading Example

- https://www.youtube.com/watch?v=Xj1uYKa8rIw: Best explanation

Life cycle of a Thread

- There are 5 states of thread
- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated

Life cycle of a Thread



Life cycle of a Thread

1. New: The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable: The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running: The thread is in running state if the thread scheduler has selected it.

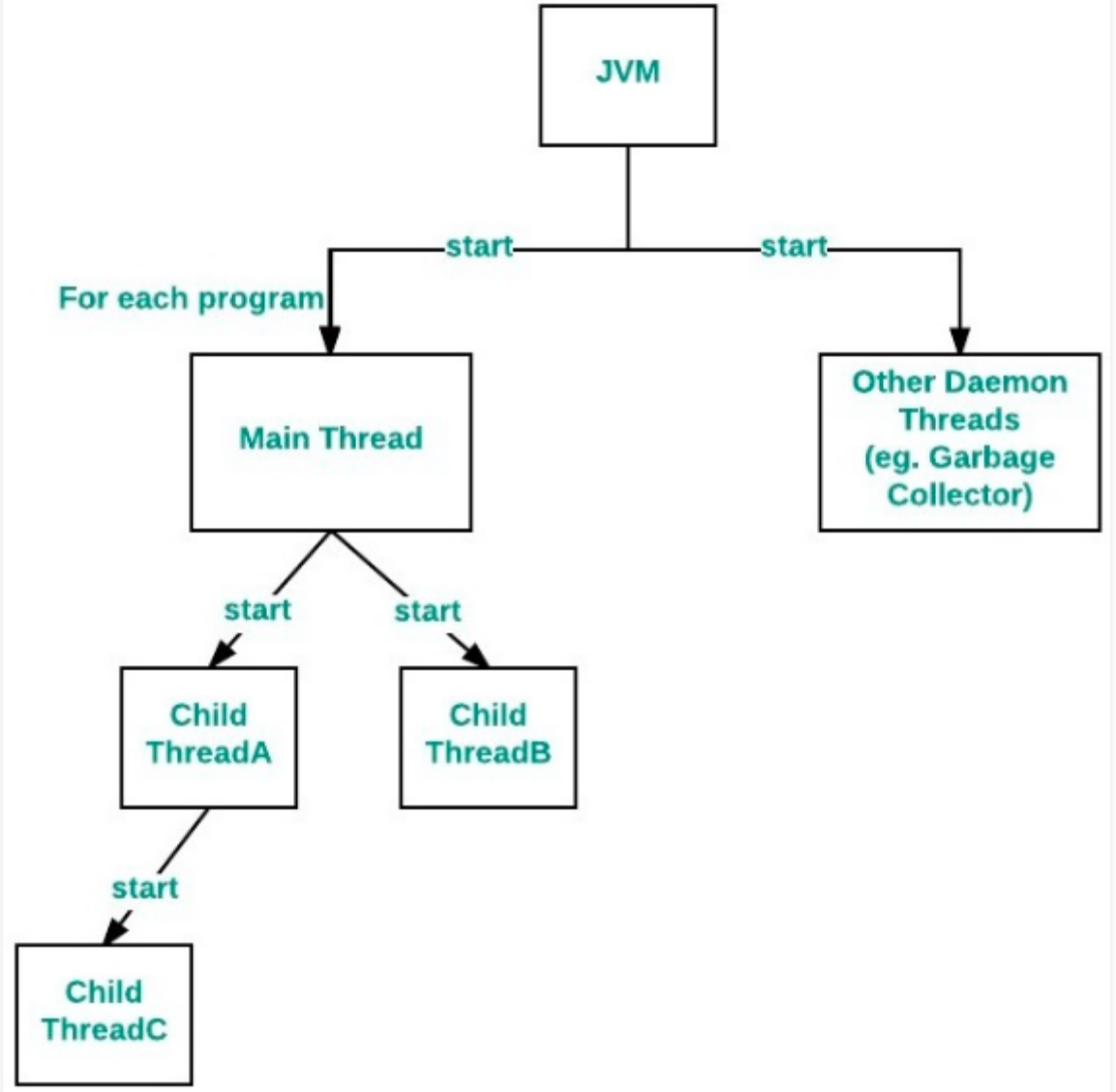
4) Non-Runnable (Blocked): This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated: A thread is in terminated or dead

Main thread

- When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program, because it is the one that is executed when our program begins.
- **Properties :**
- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions

Main thread


JVM
start
start
For each program
Main Thread
Other Daemon Threads (eg. Garbage Collector)
start
start
Child ThreadA
Child ThreadB
start
Child ThreadC

Main thread

```
public class Test extends Thread
{
    public static void main(String[] args)
    {
        // getting reference to Main thread
        Thread t = Thread.currentThread();

        // getting name of Main thread
        System.out.println("Current thread: " + t.getName());

        // changing the name of Main thread
        t.setName("Geeks");
        System.out.println("After name change: " + t.getName());

        // getting priority of Main thread
        System.out.println("Main thread priority: "+ t.getPriority());
}
```

Output:

```
Current thread: main
After name change: Geeks
Main thread priority: 5
```

JAVA

Unit 7(b)

isAlive()

- The **isAlive()** method of thread class tests if the thread is alive. A thread is considered alive when the start() method of thread class has been called.
- Syntax: **public final boolean** isAlive()

isAlive()

```
public class JavaIsAliveExp extends Thread
{
   public void run()
   {
      try
      {
         Thread.sleep(300);
         System.out.println("is run() method isAlive "+Thread.currentThread().isAlive());
      }
      catch (InterruptedException ie) {
      }
   }
   public static void main(String[] args)
   {
      JavaIsAliveExp t1 = new JavaIsAliveExp();
      System.out.println("before starting thread isAlive: "+t1.isAlive());
      t1.start();
      System.out.println("after starting thread isAlive: "+t1.isAlive());
   }
}
```

Output

```
before starting thread isAlive: false
after starting thread isAlive: true
is run() method isAlive true
```

join()

- The **join()** method of thread class waits for a thread to die.
- It is used when you want one thread to wait for completion of another.
- It does not **return** any value.

Join()

```
public class JoinExample1 extends Thread
{
   public void run()
   {
      for(int i=1; i<=4; i++)
      {       Thread.sleep(500);
          }
 public static void main(String args[])
   {
        JoinExample1 t1 = new JoinExample1();
      JoinExample1 t2 = new JoinExample1();
      JoinExample1 t3 = new JoinExample1();
      // thread t1 starts
      t1.start();
      // starts second thread when first thread t1 is died.
      try
      {             t1.join();
      }    catch(Exception e){System.out.println(e);}
      // start t2 and t3 thread
      t2.start();
      t3.start();    } }
```

Output:

```
1
2
3
4
1
1
2
2
3
3
4
4
```

Thread priority

- https://www.geeksforgeeks.org/java-thread-priority-multithreading/

Synchronization in Java

- Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.
- Java provides a way of creating threads and synchronizing their task by using **synchronized blocks**.
- Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Synchronization in Java

- Synchronized keyword is only applicable on methods, can't apply for classes and variables.
- Synchronized keyword creates a block known as critical section.

Synchronization in Java

- https://www.javatpoint.com/synchronization-in-java

Interthread Communication

- https://www.javatpoint.com/inter-thread-communication-example

Suspending, Resuming and Stopping threads

public void suspend()

This method puts a thread in the suspended state and can be resumed using resume() method.

---

public void stop()

This method stops a thread completely.

---

public void resume()

This method resumes a thread, which was suspended using suspend() method.

Important note

- The **suspend()** method of **Thread** class was deprecated by Java 2 several years ago. This was done because the **suspend()** can sometimes cause serious system failures. Assume that a thread has obtained locks on the critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.
- The **resume()** method is also deprecated. It does not cause problems, but cannot be used without the **suspend()** method as its counterpart.
- The **stop()** method of **Thread** class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. The trouble is that, the **stop()** causes any lock the calling thread

JAVA

UNIT 9

Java I/O

- **Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform **file handling in Java** by Java I/O API.

Stream

- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
- **1) System.out:** standard output stream
- **2) System.in:** standard input stream
- **3) System.err:** standard error stream

Read input from console

- **Using Buffered Reader Class**
- **Using Scanner Class**
- **Using Console Class**

Using Scanner Class

- The main purpose of the Scanner class is to read input from the user in the command line.
- Advantages:
- Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
- https://www.geeksforgeeks.org/scanner-class-in-java/

Writing console output

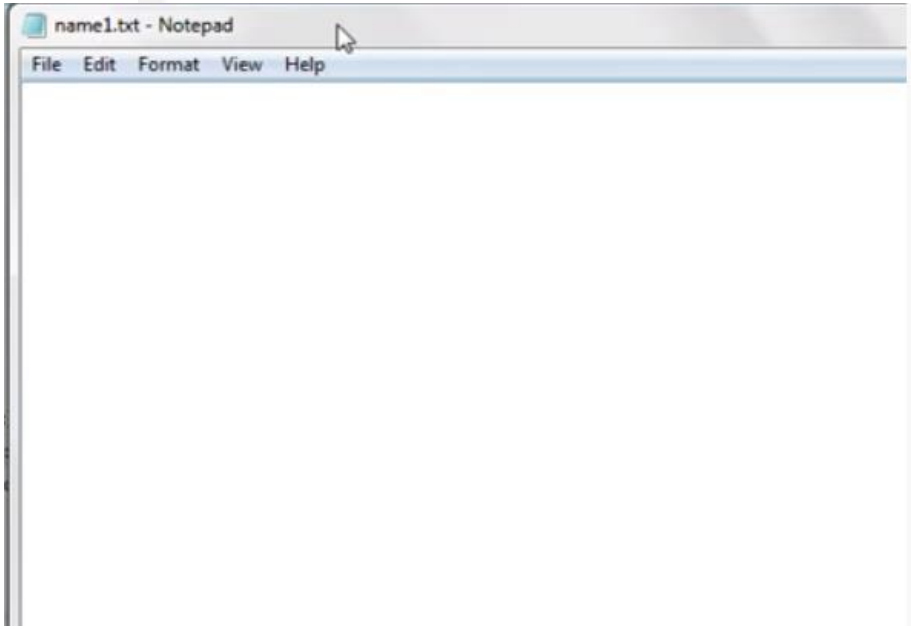
- For console output, we can use ***System.out* — an instance of the *PrintStream* class,** which is a type of *OutputStream*.
- In our example, we'll use console output to provide a prompt for user input and display a final message to a user.
- Let's use the *println()* method to print a *String* and terminate the line:

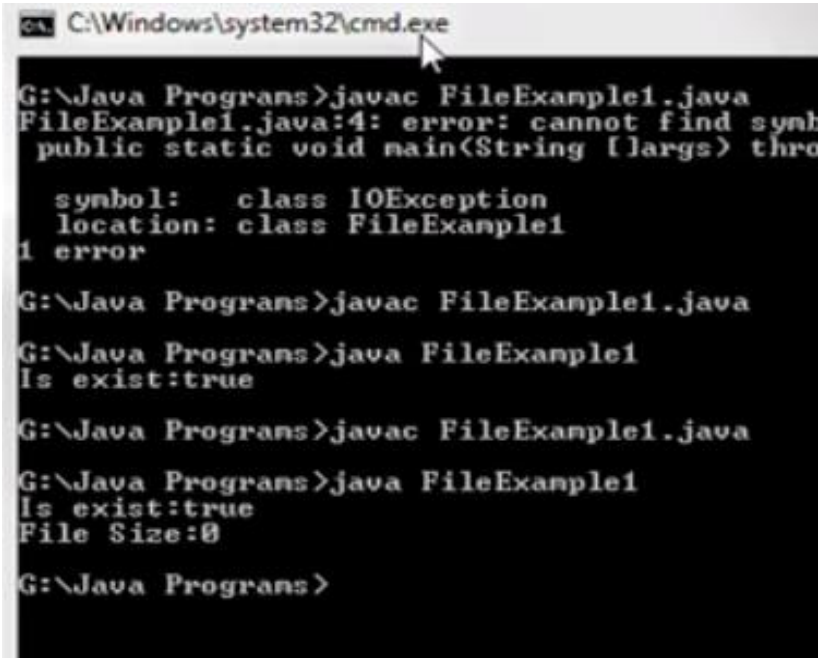
```
System.out.println("Please enter your name ");
```

- Alternately, we can use the *print()* method, which works similarly to *println()*, but without terminating the line.

How to Create a File

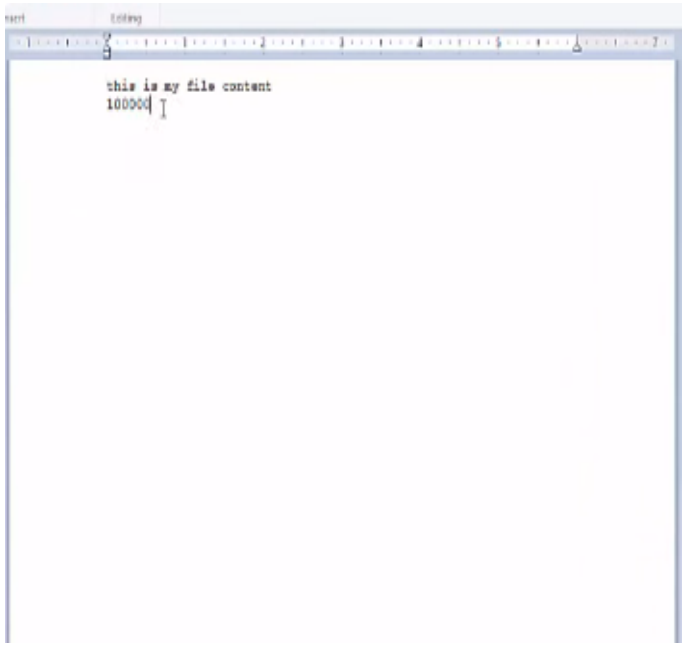
```
import java.io.File;
import java.io.IOException;
public class FileExample1
{
 public static void main(String []args) throws IOException
 {
  File f1=new File("g:/Java Programs/name1.txt");
  f1.createNewFile();
  System.out.println("Is exist:"+f1.exists());
  System.out.println("File Size:"+f1.length());
 }
}
```





Write in file using PrintWriter class

```java
import java.io.PrintWriter;

public class Demo {

    public static void main(String[] args) {

        try {
        File file = new File("fileName.txt");

        if(!file.exists()) {
                file.createNewFile();
        }

        PrintWriter pw = new PrintWriter(file);
        pw.println("this is my file content");
        pw.println(100000);
        pw.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```



PrintWriter class

- https://www.javatpoint.com/java-printwriter-class
- Note: flush() writes the content of the buffer to the destination and makes the buffer empty for further data to store but it does not closes the stream permanently. That means you can still write some more data to the stream.
- But close() closes the stream permanently. If you want to write some data further, then you have to reopen the stream again and append the data with the existing ones.

Read java file

- *FileReader* is useful to read data in the form of characters from a ‘text’ file.

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
class ReadFile
{       public static void main(String[] args) throws IOException
   {              int  ch;
          // check if File exists or not
             try
        {          FileReader fr   = new FileReader("text");
        }
        catch (FileNotFoundException fe)
        {
           System.out.println("File not found");
        }
        // read from FileReader till the end of file
        while ((ch=fr.read())!=-1)
           System.out.print((char)ch);
        // close the file
        fr.close();        }    }
```

Unit 8

Part b

Applet

- An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET tag and hosted on a web server.
- Applets are used to make the web site more dynamic and entertaining.

Applet

- All applets are sub-classes (either directly or indirectly) of *java.applet.Applet* class.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at main() method.
- Output of an applet window is not performed by *System.out.println()*. Rather it is handled with various AWT methods, such as *drawString()*.

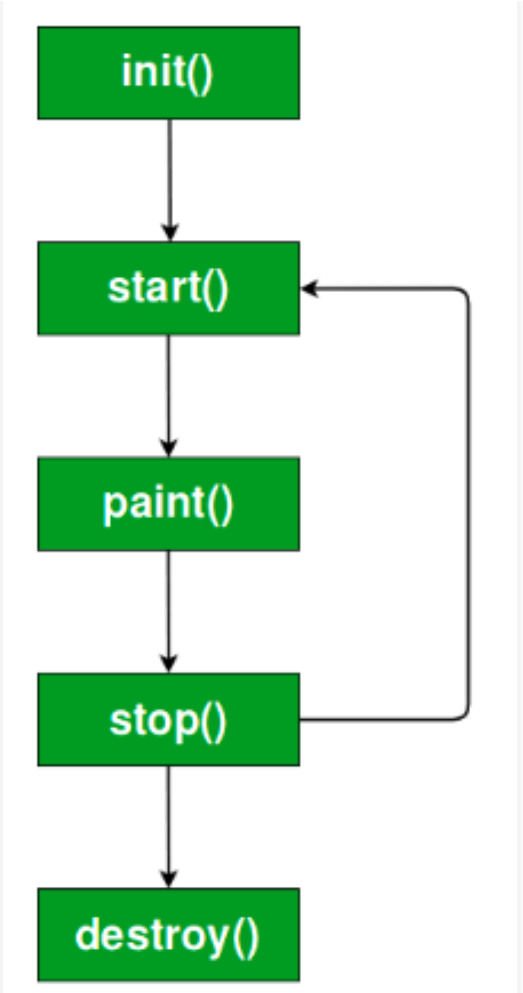
Applet

- **Advantage of Applet**
- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.
- **Drawback of Applet**
- Plugin is required at client browser to execute applet.

Application program v/s Applet

Application	Applet
Applications are stand-alone programs that can be run independently without having to use a web browser.	Applets are small Java programs that are designed to be included in a HTML web document. They require a Java-enabled browser for execution.
Java applications have full access to local file system and network.	Applets have no disk and network access.
It requires a main method() for its execution.	It does not require a main method() for its execution.
Applications can run programs from the local system.	Applets cannot run programs from the local machine.
An application program is used to perform some task directly for the user.	An applet program is used to perform small tasks or part of it.
It can access all kinds of resources available on the system.	It can only access the browser specific services.

Applet life cycle



Applet life cycle

- **init( ) :** The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.
- **start( ) :** The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Note that **init( )** is called once i.e. when the first time an applet is loaded whereas **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

Applet life cycle

- **paint( ) :** The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.**paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.
- The **paint( )** method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Applet life cycle

- **stop** – This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** – This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.

Creating an executable applet

- First a java file is created with applet code.
- Then a HTML file is created in which we want to embed our applet.

Creating applet

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
   public void paint (Graphics g) {
      g.drawString ("Hello World", 25, 50);
   }
}
```

```
<html>
   <title>The Hello, World Applet</title>
   <hr>
   <applet code = "HelloWorldApplet.class" width = "320" height = "120">
      If your browser was Java-enabled, a "Hello, World"
      message would appear here.
   </applet>
   <hr>
</html>
```

Creating applet

- The above java program begins with two import statements. The first import statement imports the Applet class from applet package. Every AWT-based(Abstract Window Toolkit) applet that you create must be a subclass (either directly or indirectly) of Applet class. The second statement import the Graphics class from AWT package.
- The next line in the program declares the class HelloWorld. This class must be declared as public because it will be accessed by code that is outside the program. Inside HelloWorld, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet.
- Inside **paint( )** is a call to *drawString( )*, which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location. It has the following general form:void drawString(String message, int x, int y) Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to *drawString( )* in the applet causes the message “Hello World” to be displayed beginning at location 20,20.

Output

Using appletviewer : This is the easiest way to run an applet. To execute HelloWorld with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is saved with RunHelloWorld.html,then the following command line will run HelloWorld :

```
appletviewer RunHelloWorld.html
```

What is an Event?

- Change in the state of an object is known as event i.e. event describes the change in state of source.
- Events are generated as result of user interaction with the graphical user interface components.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

What is Event Handling?

- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- This mechanism have the code which is known as event handler that is executed when an event occurs.
- Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

Event handling models

Till now two models have been introduced in Java for receiving and processing events. The event handling mechanisms of these models differ a lot from each other.

- **Java 1.0 Event Model**
- **Delegation Event Model**

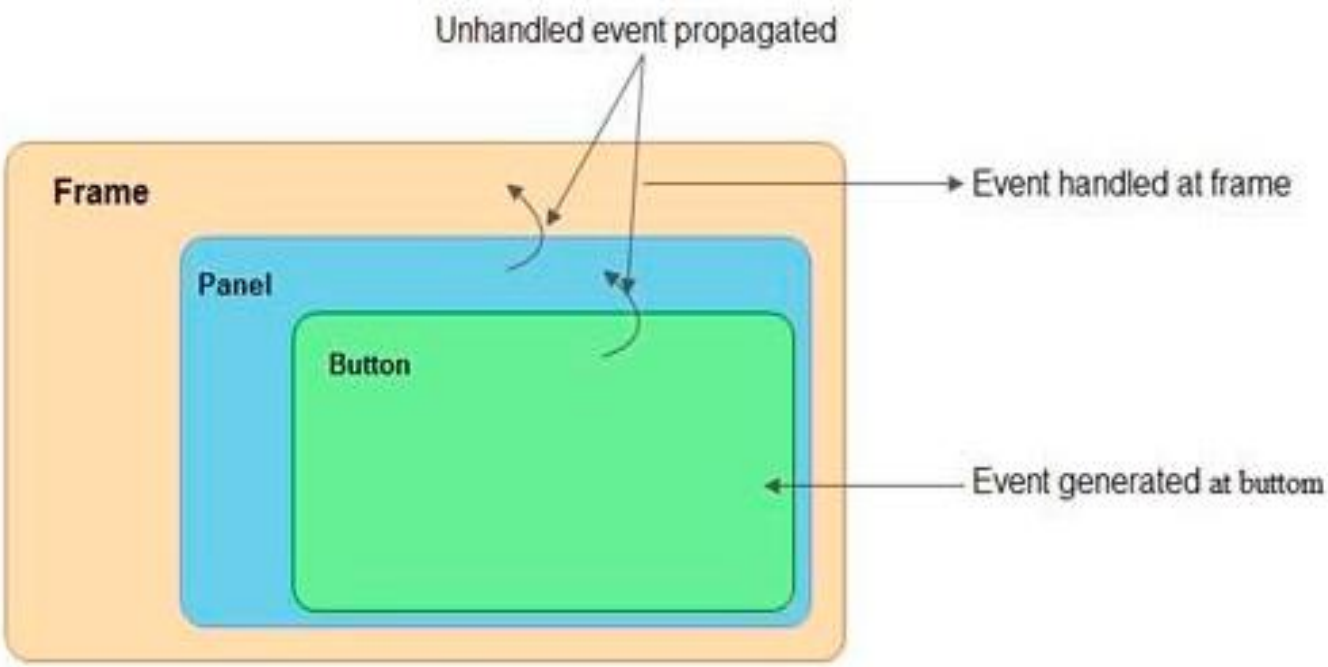
Java 1.0 Event Model

- The Java 1.0 Event model for event processing was based on the concept of containment.
- In this approach, when a user-initiated event is generated it is first sent to the component in which the event has occurred. But in case the event is not handled at this component, it is automatically propagated to the container of that component.
- This process is continued until the event .is processed or it reaches the root of the containment hierarchy

Java 1.0 Event Model

- For example, as shown in the next slide, *Button* is contained in the *Panel* which itself is contained within the *Frame.*
- When the mouse is clicked on *Button,* an event is generated which is first sent to the *Button.* If it is not handled by it then this event is forwarded to the *Panel* and if it cannot handle the event, it is further sent to the *Frame.*
- *Frame* being the root of the given hierarchy processes this event. So the event is forwarded up the containment hierarchy until it is handled by a component.

Java 1.0 Event Model



Java 1.0 Event Model

- The major drawback in this approach is that events are frequently sent to those components that cannot process them, thus wasting a lot of CPU cycles.

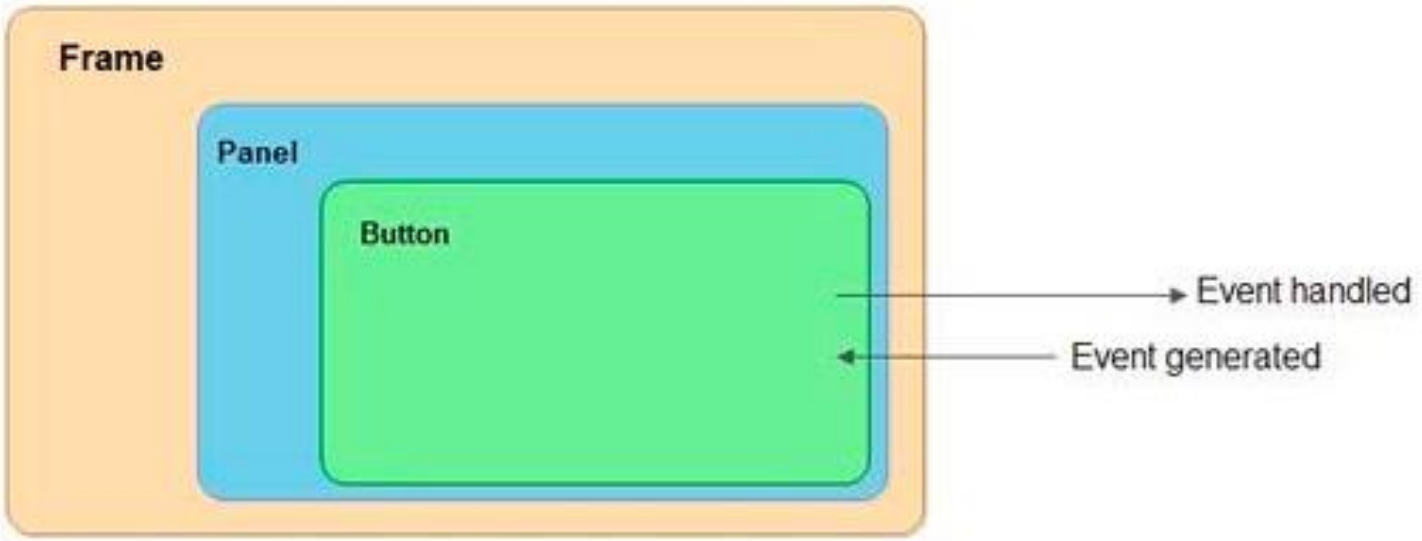
Delegation Event Model

- The advanced versions of Java ruled out the limitations of Java 1.0 event model. This model is referred to as the **Delegation Event Model** which defines a logical approach to handle events.
- It is based on the concept of source and listener. A **source** generates an event and sends it to one or more listeners. On receiving the event, **listener** processes the event and returns it.
- The notable feature of this model is that the source has a registered list of listeners which will receive the events as they occur. Only the listeners that have been registered actually receive the notification when a specific event is generated.

Delegation Event Model

- For example, as shown in Figure, when the mouse is clicked on *Button,* an event is generated.
- If the *Button* has a registered listener to handle the event, this event is sent to *Button,* processed and the output is returned to the user. However, if it has no registered listener the event will not be propagated upwards to *Panel* or *Frame.*

Delegation Event Model



Steps involved in event handling(Delegation event model)

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Sources of events

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Classes and Listener Interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Listener interface examples

- https://www.youtube.com/watch?v=hrh9-oIBr6A

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){

//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);

//register listener
b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

Output

KeyEvent

- On entering the character the Key event is generated. There are three types of key events which are represented by the integer constants. These key events are following
- KEY_PRESSED
- KEY_RELEASED
- KEY_TYPED

JAVA

UNIT 11

String Length() in JAVA

- This function is used to get the length of a Java String. The string length method returns the number of characters written in the String including spaces.
- https://www.geeksforgeeks.org/length-vs-length-java/

String Length() in JAVA

```java
// Java program to illustrate the
// concept of length
// and length()
public class Test {
public static void main(String[] args)
    {
        // Here array is the array name of int type
        int[] array = new int[4];
        System.out.println("The size of the array is " + array.length);

        // Here str is a string object
        String str = "GeeksforGeeks";
        System.out.println("The size of the String is " + str.length());
    }
}
```

Output:

```
The size of the array is 4
The size of the String is 13
```

Special String operations

- https://www.javatpoint.com/java-string

StringBuffer class

- https://www.geeksforgeeks.org/stringbuffer-class-in-java/

[Total No. of Questions: 09] [Total No. of Pages: .....]

Uni. Roll No. …….

Program: Btech IT
Semester: 5th
Name of Subject: Programming in Java
Subject Code: IT-14502
Paper ID: 15404

23-07-21(M)

Time Allowed: 02 Hours **Max. Marks: 60**

NOTE:

1) Each question is of 10 marks.
2) Attempt any six questions out of nine
3) Any missing data may be assumed appropriately

Q1. Illustrate various non-primitive data types used in java along with examples.

Q2. Differentiate widening and narrowing type casting in java with the help of a program.

Q3. Discuss any five array methods in java along with their syntax.

Q4. Write a program in java to print the grades of a student using ladder if else statements.

Q5. Write a java program to display the details of a student using parametrised constructor.

Q6. With the help of an example program demonstrate the use of multiple catch statements in java.

Q7. Explain the concept of multithreads in java along with some suitable example.

Q8. Write a java program to create an executable applet.

Q9. Describe the way to connect database with a java application.

****

[Total no. of Questions: 9]
Uni. Roll No. _____

[Total no. of Pages: 02]

EVENING

05 DEC 2018

Program/Course: B. Tech. (Sem 5)
Name of Subject: Programming in Java
Subject Code: IT-14502
Paper ID: 15404

Time Allowed: 3 Hours

Max. Marks: 60

NOTE:

1) **Section A** is **compulsory**
2) Attempt any **four** questions from **Section-B** and any **two** questions from **Section-C**
3) Any missing data may be assumed appropriately

Section-A

[Marks: 02 each]

Q. 1.

(a) What is the role of "this" keyword?
(b) Differentiate super and final.
(c) Why garbage collection is used in Java?
(d) Draw the life cycle of a thread in Java.
(e) Differentiate recursion and iteration.
(f) Write a statement to establish database connection in Java.
(g) Define adapter class.
(h) Write down the methods for string comparison and string search in Java.
(i) Write down the rules for creating a variable in Java.
(j) What is byte code?

Section-B

[Marks: 05 each]

Q. 2. What is the significance of exception handling in Java? Develop a program to handle following exceptions:
(a) Array index out of bound
(b) Divide by zero

Q. 3. Differentiate between abstract class and interface.

Q. 4. Develop a program to illustrate the concept of multithreading using isAlive() and join() methods in Java.

Q. 5. List all Event Listeners in Java and give methods of any three of them.

Q. 6. Design a program in Java to illustrate method overriding.

Sections-C

[Marks: 10 each]

Q. 7.

(a) Draw and explain the life cycle of Applet in Java. (5)
(b) Design an Applet which sets the color and draws a filled oval. (5)

Q. 8.

Q8.

(a) How would you pass objects as arguments? Demonstrate with a program in Java. (5)

(b) What is a package in Java? How would you provide access protection to packages? Explain. (5)

Q. 9. What is the role of constructors? Design a program to represent constructor overloading in Java.

****

Roll No. [Total No. of Pages: 1]

[Total No. of Questions: 09]

Course: B.Tech.(IT)
Sem.: 5th
Name of Subject: Programming in Java
Subject Code: IT-14502
Paper ID: 15404

EVENING

28 MAY 2018

Time Allowed: 03 Hours **Max. Marks: 60**

NOTE:

1) **Section-A** is **compulsory**
2) Attempt any **four** questions from **Section-B** and any **two** questions from **Section-C**
3) Any missing data may be assumed appropriately

Section – A [Marks: 02 each]

Q1.

a) How Java supports platform independency?
b) Differentiate between applet and application.
c) Differentiate between type conversion and type casting with example(s).
d) How do interfaces support polymorphism?
e) What is the need of static variables?
f) How does String class differ from StringBuffer class?
g) Differentiate between checked and unchecked exceptions.
h) What is the role of adaptor classes? Give example.
i) Why is thread synchronization used in Java?
j) Differentiate between Statement and PreparedStatement interface.

Section – B [Marks: 05 each]

Q2. Write a program to swap two positive numbers (without using third temporary variable) using bitwise operators.

Q3. Discuss with example the concept of packages and nested interfaces.

Q4. Describe the difference between method overriding and overloading with the help of an example.

Q5. Explain thread life cycle with the help of an example.

Q6. Write a program to demonstrate the concept of constructor chaining.

Section – C [Marks: 10 each]

Q7. Discuss in detail two object oriented paradigms.

Q8. Develop an interface to perform any one DML operation by assuming suitable data.

Q9. Explain with the help of program the concept of user defined exception handling to handle multiple exceptions and terminate with a suitable message.

EVENING

03 JUN 2019

[Total No. of Questions: 09] [Total No. of Pages: 2]

Uni. Roll No: .....

Program/Course: B.Tech (5th Semester)

Name of the Subject: Programming in Java

Subject Code: IT-14502

Paper ID: 15404

Time Allowed: 03 Hours **Max. Marks: 60**

NOTE:

1. **Section –A** is **compulsory**
2. Attempt any **four** questions from **Section-B** and any **two** questions from **Section-C**

Section-A [Marks: 02 each]

Q1.

a. Differentiate Java and C++.
b. Describe left shift and right shift operators with examples
c. How a programmer can overcome instance variable hiding using this keyword?
d. How does String class differ from the String Buffer class?
e. Is it possible to achieve true parallelism using multithreading? What are the limitations in it?
f. State four similarities between Interfaces and Classes.
g. Why Java is called Platform independent language?
h. Explain the concept of Typecasting with suitable example.
i. Distinguish between error and exceptions
j. Does Java support multiple inheritance? Justify your answer.

Section-B [Marks: 05 Each]

Q2. Write a program to find whether any given number is an Armstrong number or not.

Q3. Write a program to concatenate a given string to the end of another Stri

Q4. Discuss Java typical environment

Q5. Explain the following terms with respect to exception handling. i) try ii) catch iii) throw iv) finally.

Q6. Define multi threading. How is it implemented in Java?

Section-C

[Marks: 10 each (05 for each subpart if any)]

Q7. Explain the life cycle of an applet with neat diagram. Implement Applets by passing Parameters

Q8. Write a java program that simulates a traffic light. The program lets the user select one of three lights: red, yellow, or green. When a radio button is selected, the light is turned on, and only one light can be on at a time No light is on when the program starts.

Q9. Write a note on following:

a. Overloading vs. Overriding
b. Database Connectivity

****