Unit-I

Syllabus

- Review of Object oriented concepts
- History of Java
- Java buzzwords
- JVM architecture
- Data types
- Variables
- Scope and life time of variables
- arrays
- operators
- control statements
- type conversion and casting,
- simple java program,
- constructors,
- methods,
- Static block,
- Static Data,
- Static Method String and String Buffer Classes,
- Using Java API Document

Object oriented concepts

OOPs

- Object-Oriented Programming is a paradigm that provides many concepts, such as **Abstraction**, **Encapsulation**, **Inheritance**, **Polymorphism**, etc.

- 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.

- Popular object-oriented languages are Java, C#, PHP, Python, C++, Kotlin etc..

Abstraction

- Providing the essential features without its inner details is called abstraction (or) hiding internal implementation is called Abstraction.
- We can enhance the internal implementation without effecting outside world.
- Abstraction provides security.
- A class contains lot of data and the user does not need the entire data.
- The advantage of abstraction is that every user will get his own view of the data according to his requirements and will not get confused with unnecessary data.

Encapsulation

- Wrapping up of data (variables) and methods into single unit is called Encapsulation.
- Class is an example for encapsulation.
- Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.

Inheritance

- Acquiring the properties from one class to another class is called inheritance (or) producing new class from already existing class is called inheritance.
- Reusability of code is main advantage of inheritance.
- In Java inheritance is achieved by using extends keyword.
- The properties with access specifier private cannot be inherited.

Polymorphism

- The word polymorphism came from two Greek words ‘poly’ means ‘many’ and ‘morphos’ means ‘forms’.
- Thus, polymorphism represents the ability to assume several different forms.
- The ability to define more than one function with the same name is called Polymorphism

OOPs Terms

- In object-oriented programming, a class is a programming language construct that is used as a blueprint to create objects.
- This blueprint includes attributes and methods that the created objects all share.
- Usually, a class represents a person, place, or thing - it is an abstraction of a concept within a computer program.
- Fundamentally, it encapsulates the state and behavior of that which it conceptually represents.
- It encapsulates state through data placeholders called member variables; it encapsulates behavior through reusable code called methods

History of Java

- Initially Java was developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released first official version in 1995.

- He is also known as **Father of Java**

- The history of Java starts with the **Green Team.**

- **James Gosling** , **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991.

- Initially it was designed for small, embedded systems in electronic appliances like set-top boxes, TV’s

- Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.

- After that, it was called **Oak** and was developed as a part of the Green project.

Why Java named "Oak"?

- Oak is a name of Tree (It’s a symbol of strength)

- And it is national tree of many countries like the U.S.A., France, Germany, Romania, etc.

Oak Tree

- Because of some trademark issues Oak was renamed as **"Java“** in 1995
- Java is **an island** of Indonesia where the **first coffee** was produced (called java coffee). It is a kind of espresso bean.

- Note : -JAVA is not an acronym

  -It is not extension of C++

Symbol of JAVA

Java Buzzwords

- Simple
- Platform independent
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

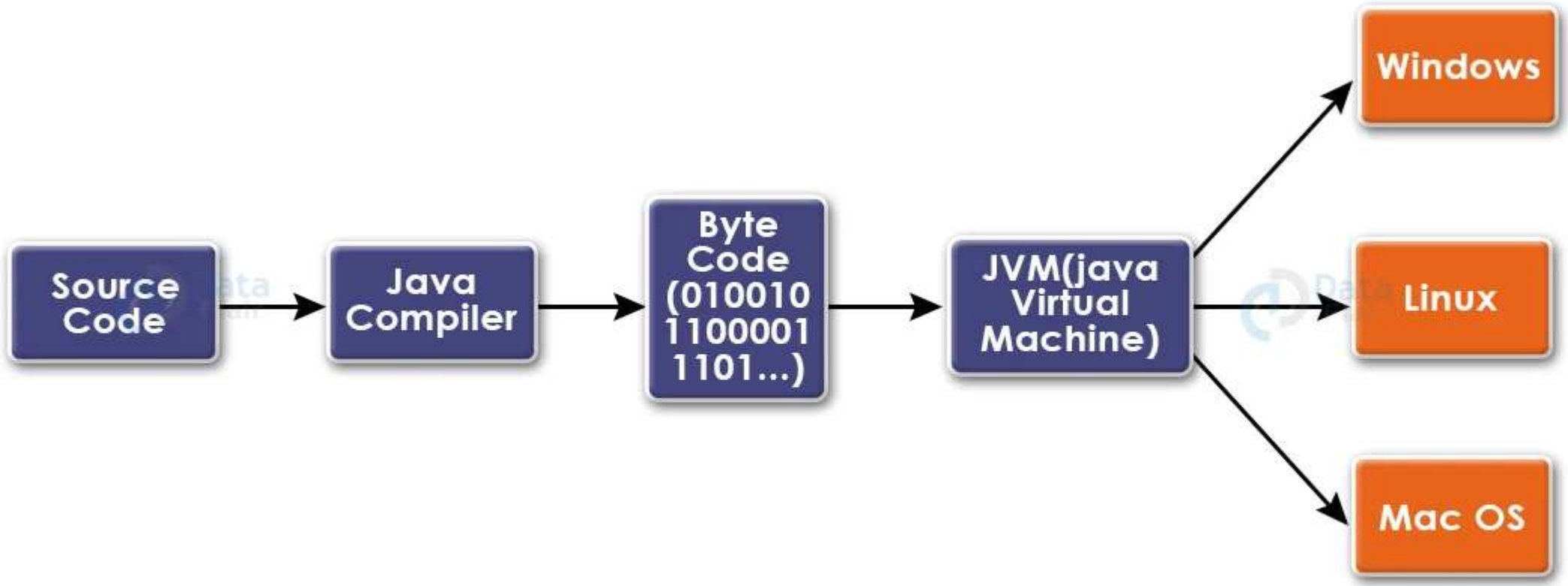
- Java is a small and simple language.
- Java does not use pointers, pre-processor header files, goto statement and many other.
- It also eliminates operator overloading and multiple inheritance.
- Java inherits the C/C++ syntax and many of the object oriented features of C++.

Platform Independent

- Compile the Java program on one OS (operating system) that compiled file can execute in any OS(operating system) is called Platform Independent Nature.

- The java is platform independent language.

- The java applications allows its applications compilation one operating system that compiled (.class) files can be executed in any operating system



Java is platform-independent

Secure

- Security becomes an important issue for a language that is used for programming on Internet.
- Every time when you download a “normal program”, here is a risk of viral infection.
- When we use a java compatible web browser, we can safely download Java applets without fear of viral infection.

- Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

Portable

- Java programs can be easily moved from one computer system to another, anywhere and anytime.

- This is the reason why Java has become a popular language for programming on Internet.

Object-Oriented

- Java is a true object oriented language.
- Almost everything in java is an object.
- All program code and data reside within objects and classes Java comes with an extensive set of classes, arranged in packages, that we can use in our programs by inheritance.
- The object model in java is simple and easy to extend.

ROBUST

- Any technology if it is good at two main areas it is said to be ROBUST
  - Exception Handling
  - Memory Allocation

- JAVA is having very good predefined Exception Handling mechanism whenever we are getting exception we are having meaning full information.

- JAVA is having very good memory management system that is Dynamic Memory (at runtime the memory is allocated) Allocation which allocates and deallocates memory for objects at runtime

Multithreaded

- Multithreaded means handling multiple tasks simultaneously.
- This means that we need not wait for the application to finish one task before beginning another.
- To accomplish this, java supports multithreaded programming which allows to write programs that do many things simultaneously.

Architecture-Neutral

- A central issue for the Java designers was that of code longevity and portability.

- One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine.

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.

Ex:

- In C programming, **int** data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture.

- But in java, int occupies 4 bytes of memory for both 32 and 64 bit architectures.

- Java Virtual Machine solves this problem. The goal is "**write once; run anywhere, any time, forever**."

Interpreted and High Performance

- Java performance is impressive for an interpreted language, mainly due to the use of byte code.
- This code can be interpreted on any system that provides a JVM.
- Java was designed to perform well on very low power CPUs.

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.

- The original version of Java (Oak) included features for intra address-space messaging.

- This allowed objects on two different computers to execute procedures remotely.
- Java revived these interfaces in a package called Remote Method Invocation (RMI).
- This feature brings an unparalleled level of abstraction to client/ server programming.

Dynamic

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.

- This makes it possible to dynamically link code in a safe and expedient manner.

JVM Architecture

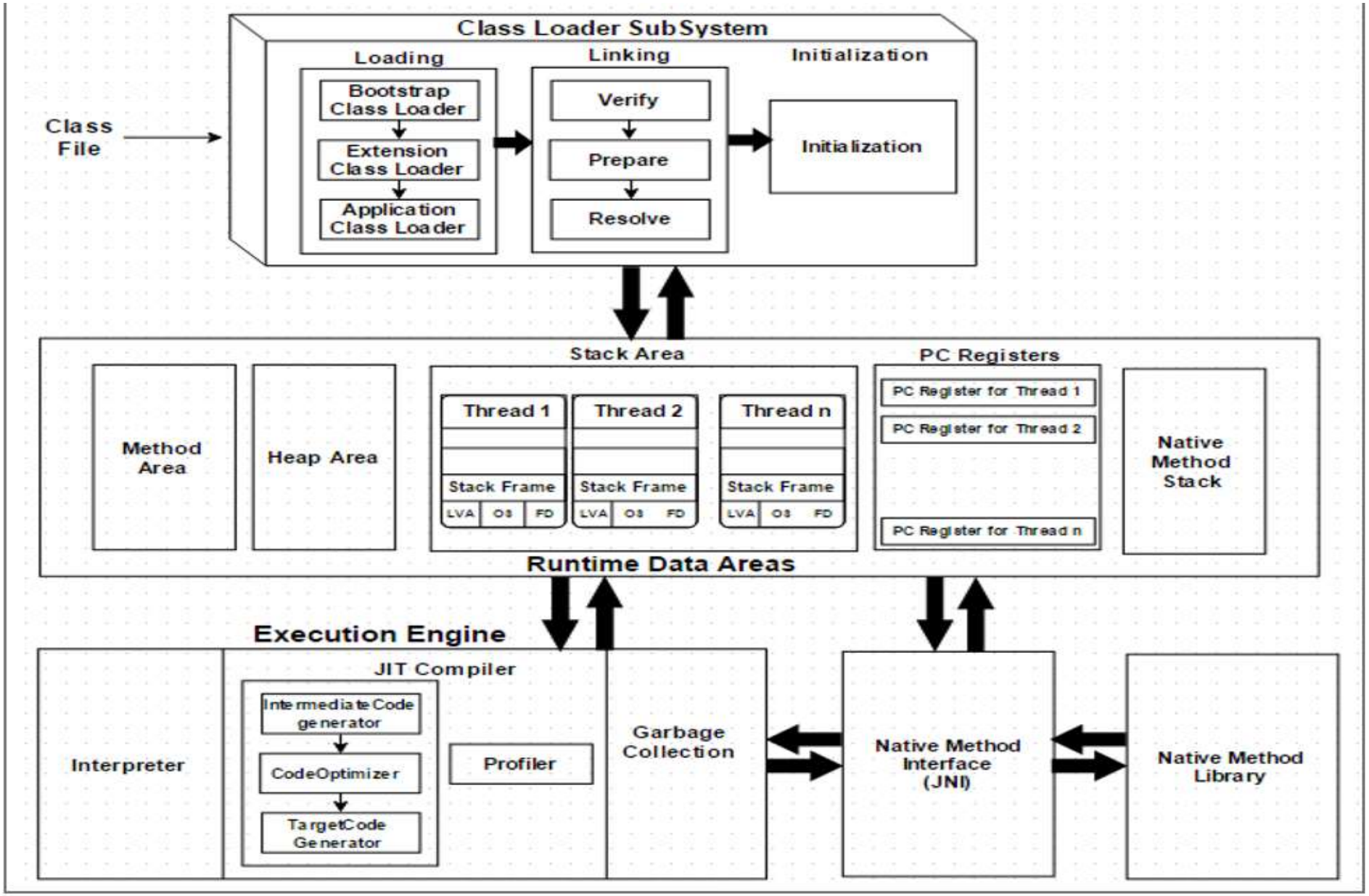
What Is the JVM?

- A **Virtual Machine** is a software implementation of a physical machine.

- Java was developed with the concept of **WORA (*Write Once Run Anywhere*)**, which runs on a **VM**.

- The **compiler** compiles the Java file into a Java **.class** file, then that .class file is input into the JVM, which loads and executes the class file.

JVM Architecture



As shown in the above architecture diagram, the JVM is divided into three main subsystems:

- ❖Class Loader Subsystem
- ❖Runtime Data Area
- ❖Execution Engine

1. ClassLoader Subsystem

- Java's dynamic class loading functionality is handled by the ClassLoader subsystem.

- It loads, links. and initializes the class file when it refers to a class for the first time **at runtime**, not compile time.

a) Loading

There are three class loaders in JVM i.e, BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader

- **BootStrap ClassLoader** : Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar**. Highest priority will be given to this loader.
- **Extension ClassLoader** :Responsible for loading classes which are inside the ext folder (jre\lib).
- **Application ClassLoader** :Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

b) Linking

- **Verify** : Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.

- **Prepare** : For all static variables memory will be allocated and assigned with default values.

- **Resolve** : All symbolic memory references are replaced with the original references from Method Area.

c) Initialization

- This is the final phase of ClassLoading.
- Here, all static variables will be assigned with the original values.
- And the static block will be executed.

2. Runtime Data Area

- The Runtime Data Area is divided into five major components
- **Method Area** : All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
- **Heap Area** : All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM.

Note:

Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

Stack Area:

- For every thread, a separate runtime stack will be created.
- For every method call, one entry will be made in the stack memory which is called Stack Frame.
- All local variables will be created in the stack memory.
- The stack area is thread-safe since it is not a shared resource.
- The Stack Frame is divided into three sub entities:
  - **Local Variable Array**
  - **Operand stack**
  - **Frame data**

- **PC Registers** :Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

- **Native Method stacks** : Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

3. Execution Engine

- The bytecode, which is assigned to the **Runtime Data Area,** will be executed by the Execution Engine.
- The Execution Engine reads the bytecode and executes it piece by piece.
- **a) Interpreter** :The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

- **b) JIT Compiler** :The JIT Compiler neutralizes the disadvantage of the interpreter.
- The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code.
- This native code will be used directly for repeated method calls, which improves the performance of the system.

Data types

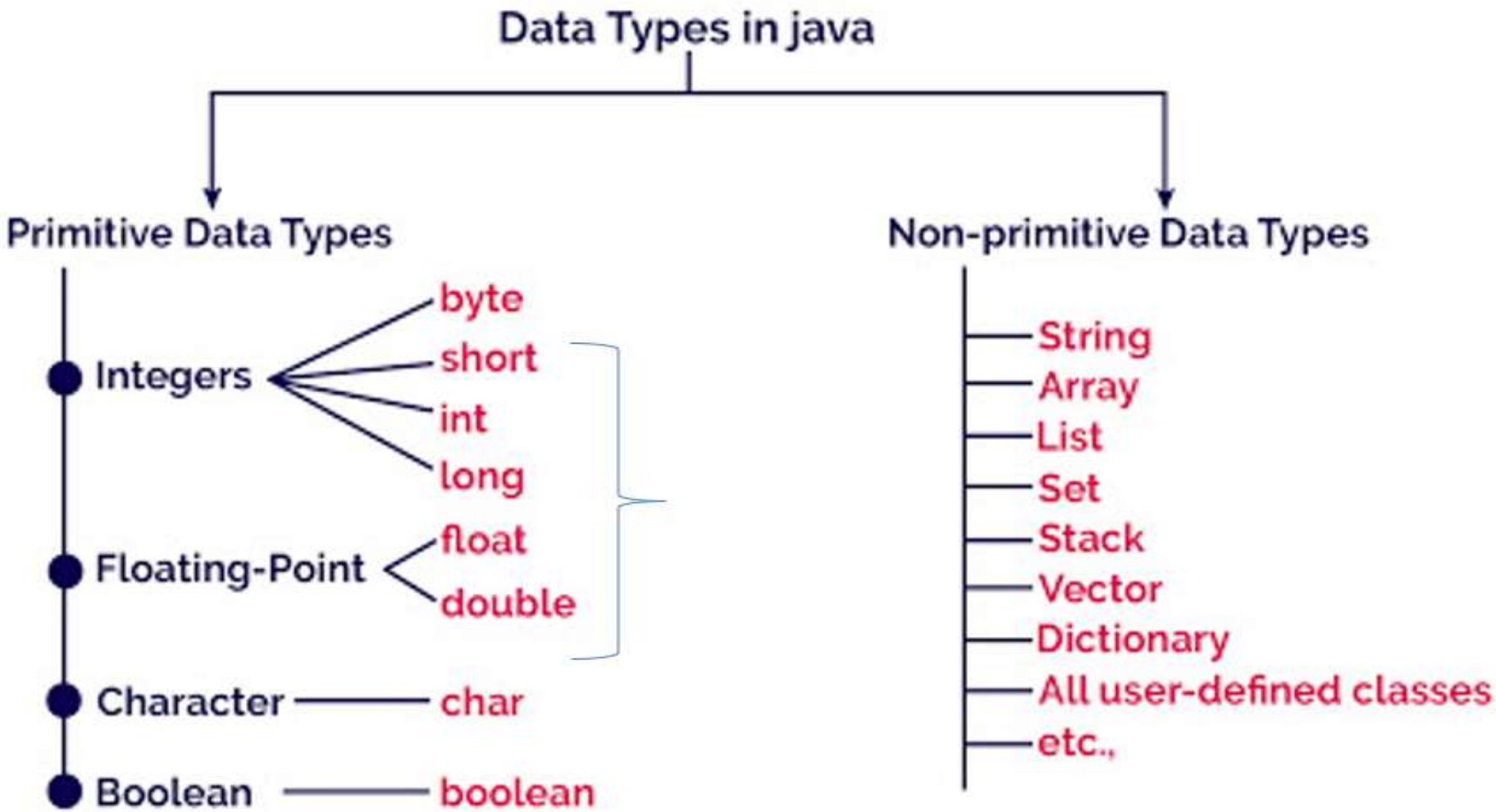
What is data type ?

- Data types are used to represent the type of the variable and type of the expression.

- Java is **Strictly typed / Strongly typed / Statically typed**

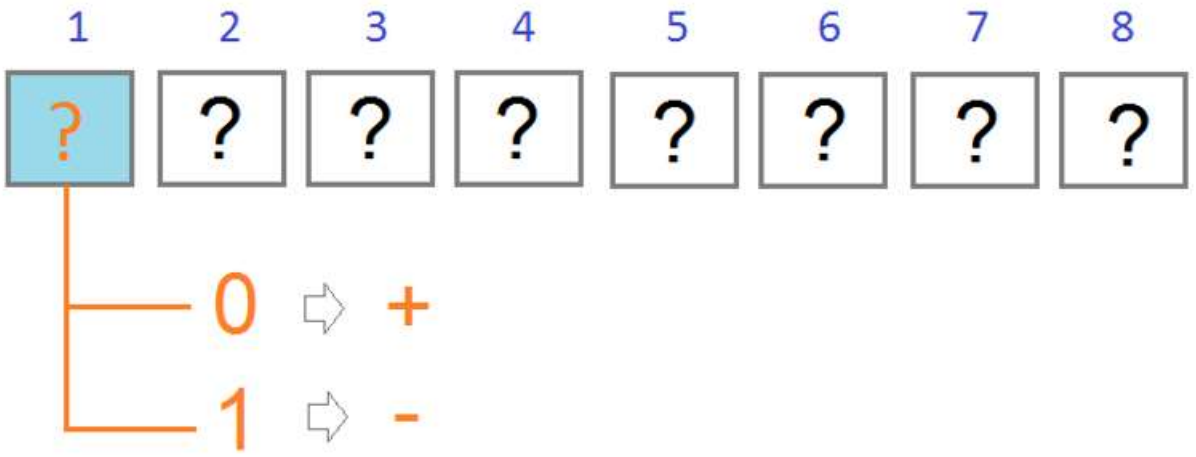
Java is a Strongly Typed

- Every variable has a type
- Every expression is a type
- All assignments are checked for type compatibility at compile time


Data Types in java
Primitive Data Types
Integers
byte
short
int
long
Floating-Point
float
double
Character
char
Boolean
boolean
Non-primitive Data Types
String
Array
List
Set
Stack
Vector
Dictionary
All user-defined classes
etc.,

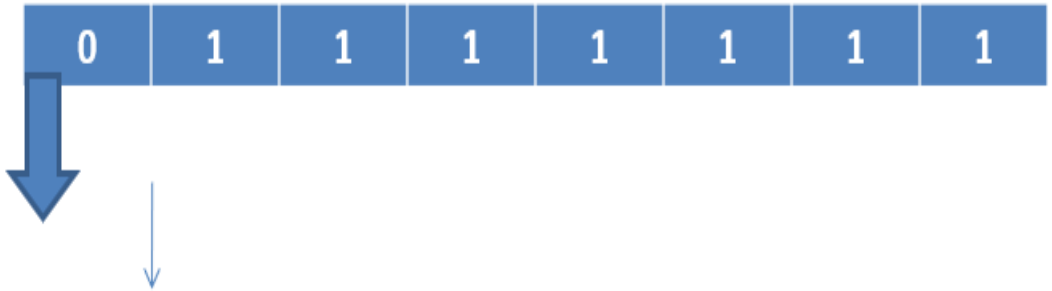
Byte

- A byte is composed of 8 consecutive **bits** in the memory of computer. Each **bit** is a binary number of 0 or 1.



Range

- Max value=127

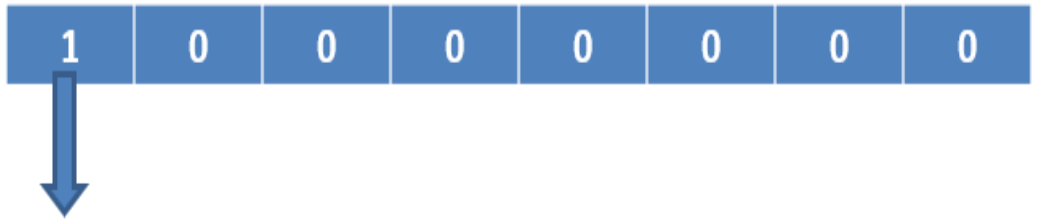


MSB=0----> +ve value

Value-→ 1+2+4+8+16+32+64=127

Range:

- Min value=-128



MSB=1----> -ve value

Value→ All negative values will be calculated in 2's complement form

0000000--->1111111

+1

10000000 ===>-128

Application area

- Used to store data into Files
- Network

short data type

- The short data type is a 16-bit signed two's complement integer.
- Rarely used data type.
- **Size:** 2 bytes
- **Range:-**32,768 to 32767.
- **Default value:** 0

int Data type

- int data type is the preferred data type when we create variables with a numeric value.
- It will take 32 bits or 4bytes of memory.
- **Range:-**2147483648 to 2147483647
- **Default value:** 0

long Data type

- Used when int is not large enough to hold the value, it has wider range than int data type.
- **Size** :8 bytes or 64 bits
- **Range:**-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- **Default value:** 0

float Data type

- A float data type is a **single precision** number format that occupies 4 bytes or 32 bits of memory.
- A float data type in java stores a decimal value with 6-7 total digits of precision.
- Ex: float f=10.25212121212f
- But it will take only **10.252121.**
- **Range:** 3.4e-038 to 3.4e+038
- **Default value: 0.0**

double Data type

- The double data type is a **double-precision** 64-bit IEEE 754 floating points.
- In Java any floating value by default it’s a **double** type
- 12-13 digits precisions it will take.
- EX: double d=10.252121212121343432222;
- Output: **10.252121212121343**

char Data type

- The char data type is a single 16-bit Unicode character.
- Ex: char c=‘A’;
- Unsigned data type.
- **Range:** 0 to 65535
- **Default value:** \u0000

Why 16 bits for char type?

- In C/C++ uses only ASCII characters and to represent all ASCII characters 8-bits is enough.

- Java uses Unicode system, to represent Unicode system 8 bit is not enough.

- Unicode=ASCII + Other language symbols.

Boolean Data type

- The Boolean data type has only two possible values: true and false.
- Use this data type for simple flags that track true/false conditions.
- This data type represents one bit of information, but its “size” isn’t something that’s precisely defined.

- Default value is **false**
- **EX:**
- **boolean b=true;**
- **boolean b=0; ➔Not allowed**

Identifiers

- An **identifier** is a name used to identify entities like a variable, methods, classes and interfaces etc.

(Or)

- Any name in the java program like variable name, class name, method name, interface name is called identifier.

Ex:

```
class Test          →    Test    identifier
{
void add()          →    add     identifier
{
int a=10;           →    a       identifier
int b=20;           →    b       identifier
}
}
```

Rules to declare identifiers

- Java identifiers should not start with numbers, it may start with alphabet symbol and underscore symbol and dollar symbol.
- An identifier should not contains symbols like + , - , . , @ , # . Only allowed special symbols are _ and $
- Duplicate identifiers are not allowed.

Ex:

```
class Test
{
   void add()
  {
        int a=10;
        int a=20;   →   An identifier should not be duplicated.
  }
}
```

- In the java applications it is possible to declare all the predefined class names and predefined interfaces names as a identifier. But it is not recommended to use.

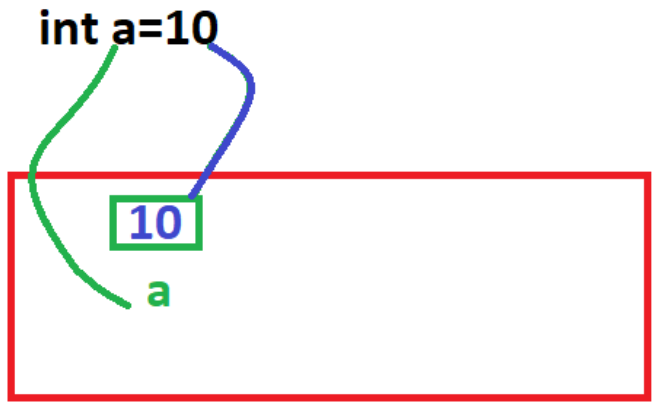
- Keywords are not allowed to use as an identifier.

- There is no length limit to define an identifier but it recommended to use short and meaning full names.

Variables

What is a variable??

- A variable is a name of the memory location. It is the small unit of storage in a program.



- The value stored in a variable can be varied during program execution.

- **Syntax:**

  datatype variable_name= value;

Ex: int a= 10;

char c=‘A’;

Note:

- **Java is strongly typed**. So before using any variable in it is mandatory to declare with specific data type

Naming convention of a variable

- As per documentation all user variables are small case.

- All constant variables should be in uppercase letter

  Ex: final double **PI**=3.14159265358979323 8;

  final int **DATABASE_VERSION**=1;

- **Note:**

  Above rules are optional. But it is highly recommended to follow Java coding standards

Types of Variables

- There are three types of variables in Java:
  - Local Variables
  - Instance Variables
  - Static Variables

Local variables

- The variables which are declare inside a method or inside a block or inside a constructor is called local variables.

Ex: class Student

```
{
        public void studentInfo()
        {
    // local variables
    String name=“Ramu”;
    int age = 26;
    System.out.println("Student name : " + name);
    }
}
```

- These variables are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function.

- Hence The scope of local variables are inside a **method** or inside a **constructor** or inside a **block**.

- JVM wont provide any initial values for local variables. So initialization of Local Variable is Mandatory. If not it will generate compile time error.

- Access modifiers (public , private , protected , default) are not allowed for local variables.

- Local variables will be stored in **stack** area.

Example:

```
public class Test
{
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
    public void m1()
    {
        //Declaring variable inside method..
        int i=10;
        System.out.println(i);
    }
    public  Test()
    {
        //Declaring variable inside constructor..
        int j=20;
        System.out.println(j);
    }
    {
        //Declaring variable inside block..
        int k=30;
        System.out.println(k);
    }
}
```

Instance Variables

- Instance variables are declared inside class and outside of methods or constructor or block.

Ex: class A

```
{
        int a;        //instance variable
        public static void main(String[] args)
        {
        }
}
```

- Instance variables are created when an object of the class is created and destroyed when the object is destroyed.
- We are able to access instance variables only inside the class any number of methods.

- Initialization of Instance Variable is **not mandatory** JVM automatically allocates default values.
- Unlike static variable, instance variables have their own separate copy i.e, if any changes done in instance variable that will not reflect on other objects.
- Instance variables are not allowed inside static area directly. But using object it will allow.

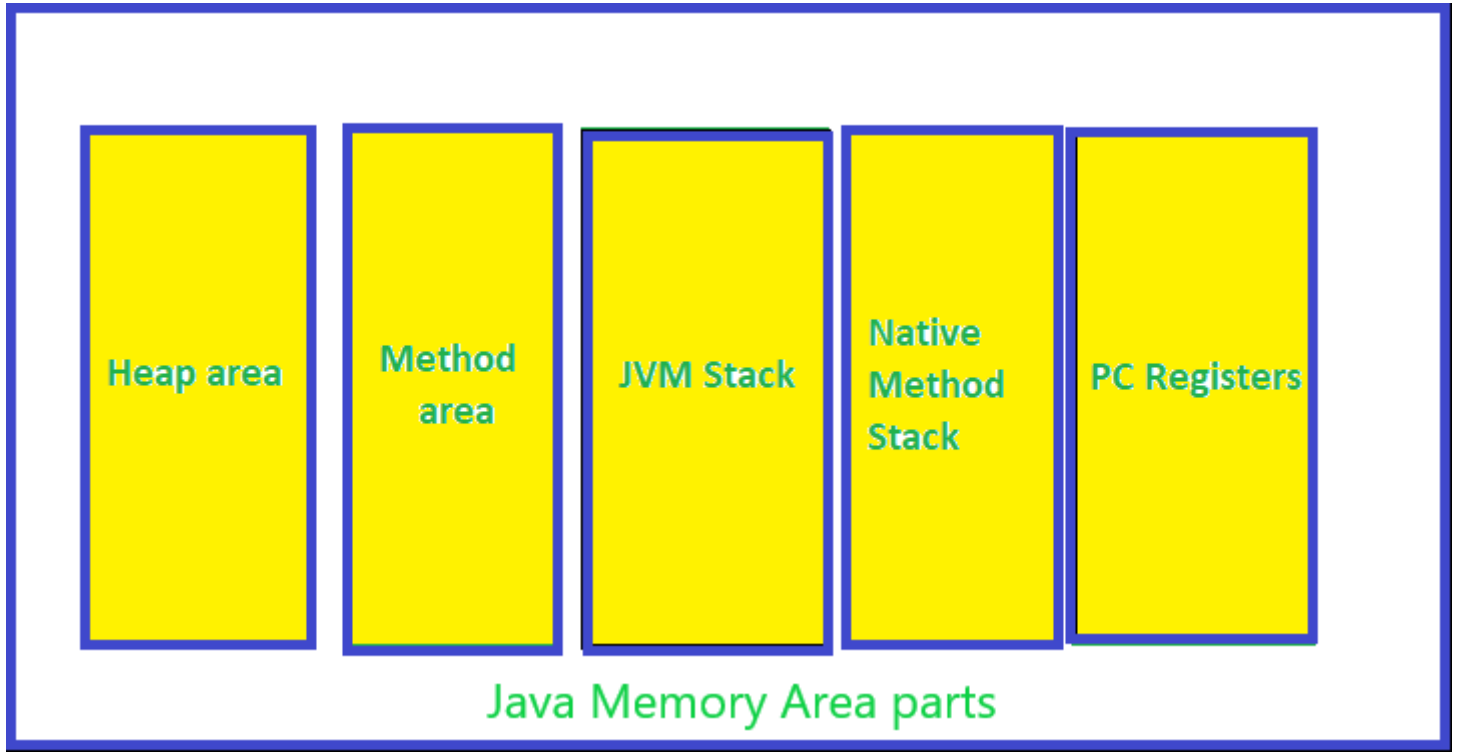
Static variables

- If any variable declared inside class and out side methods with "**static**" key word is called static variable.

Ex:
```
public class Test
{
    static int a=10;
    public static void main(String[] args)
    {
    }
}
```

- **static** variables are also called as **class variable** because they are associated with the class and common for all the instances of the class.

- **static** variables are created at class loading time and destroyed at class unloading.
- All static variables will be stored within **method** area.



- Static variables are can be accessed from any area(instance or static) directly.

- Static variables can be accessed by using objects and using class name.

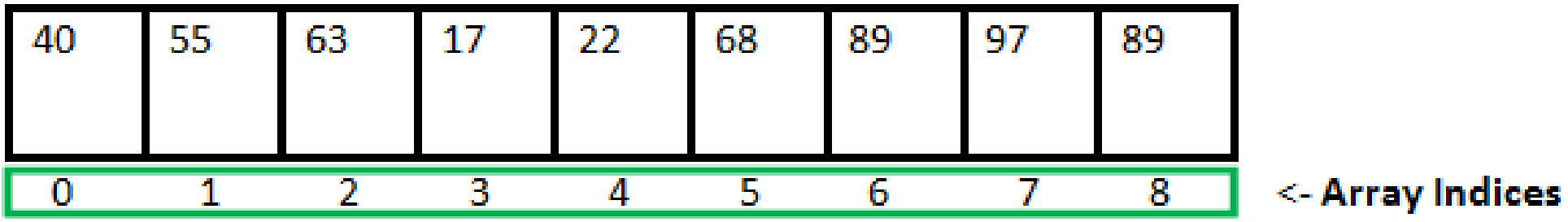
- Initialization of static variables is not mandatory

Arrays

What is an Array?

- An array is a container object that holds a fixed number of values of a single type.
- In Java, all arrays are dynamically allocated. (discussed below)
- Arrays are stored in contiguous memory [consecutive memory locations].
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using sizeof.

- The variables in the array are ordered, and each has an index beginning with 0.
- No –ve index concept in Java.
- Java array can also be used as a static field, a local variable, or a method parameter.
- The size of an array must be specified by int ,short , byte, and char value and not long.
- Maximum size of an Array is 2147483647.
- The direct superclass of an array type is Object.
- The size of the array cannot be altered(once initialized).



Array Length = 9

First Index = 0

Last Index = 8

How to create an Array:

Syntax:

Type[] arrayName;

- Data type can be primitive types like int, char, float,..(or) class type.
- Array name can be any valid identifier.

Ex: int a[];

String s[];

Student student[];

How to initialize an Array:

Approach 1:-

➢ **Syntax:**

Type variable_name[]={element1, elem2, element3, element4,..etc};

Ex: int a[]={10,20,30,40};

Approach 2:-

- In this approach we can create an object to the Array.
- Initialization is Mandatory.
- Allowed data types to specify size of an Array are byte, short, char, int only.
- **Syntax:**

```
Type Variable_name[]=new Type[Initialization];

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

    a[0]=10;
    a[1]=20;
    a[2]=30;
    a[3]=40;
```

Traversing array using for loop

Ex:

```
public class Array
{
    public static void main(String[] args) {
        int a[]={10,20,30,40};
        for (int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }
    }
}
Output:
10
20
30
40
```

Traversing array using for each loop

Ex:

```
public class Array
{
    public static void main(String[] args) {
        int a[]={10,20,30,40};
        for (int i:a)
        {
            System.out.println(i);
        }
    }
}
```

Output:

```
10
20
30
40
```

Operators

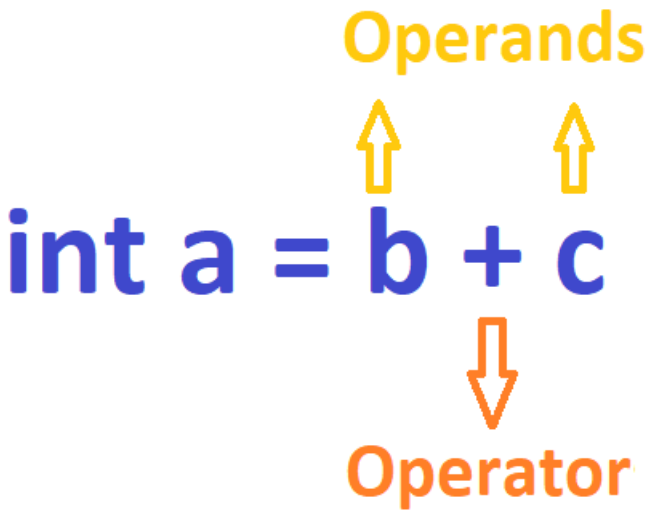
- **Operator** in java is a symbol that is used to perform operations.

  Example: +, -, *, / etc.

- There are many types of operators in java which are given below:
  - Unary Operators
  - Arithmetic Operators
  - Shift Operators
  - Bitwise Operators
  - Logical Operators
  - Relational Operators
  - Assignment Operators.
  - Ternary Operators

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Operators and Operands



Unary operators

- The unary operators require only one operand;
- They perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

Ex: int a=10;

```
a++;
++a;
```

Ex: int a=2;

```
~a;
```

Pre Increment and Post Increment

Pre Increment:

- When placed before the variable name, the operand's value is incremented instantly.

Ex: int a=10;

++a; //11

Post Increment:

- The value of the operand is incremented but the previous value is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

Ex: int a=10;

a++; // 10

Ex:

```
public class Ex2
{
    public static void main(String[] args)
    {
        int x=10;
        int y=20;
        int z=++x+y++;
        int a=++x+x++;
        int b=++y+x++;
        System.out.println(z);
        System.out.println(a);
        System.out.println(b);
}
}
```

Output:

```
31
24
35
```

Arithmetic Operators

- The Java programming language provides operators that perform addition, subtraction, multiplication, and division.
- The only symbol that might look new to you is "%", which divides one operand by another and returns the remainder as its result.

Operator	Use	Description
+	op1 + op2	Adds op1 and op2; also used to concatenate strings
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types.

- ❖Right shift(>>)
- ❖Left shift(<<)

Right shift(>>):

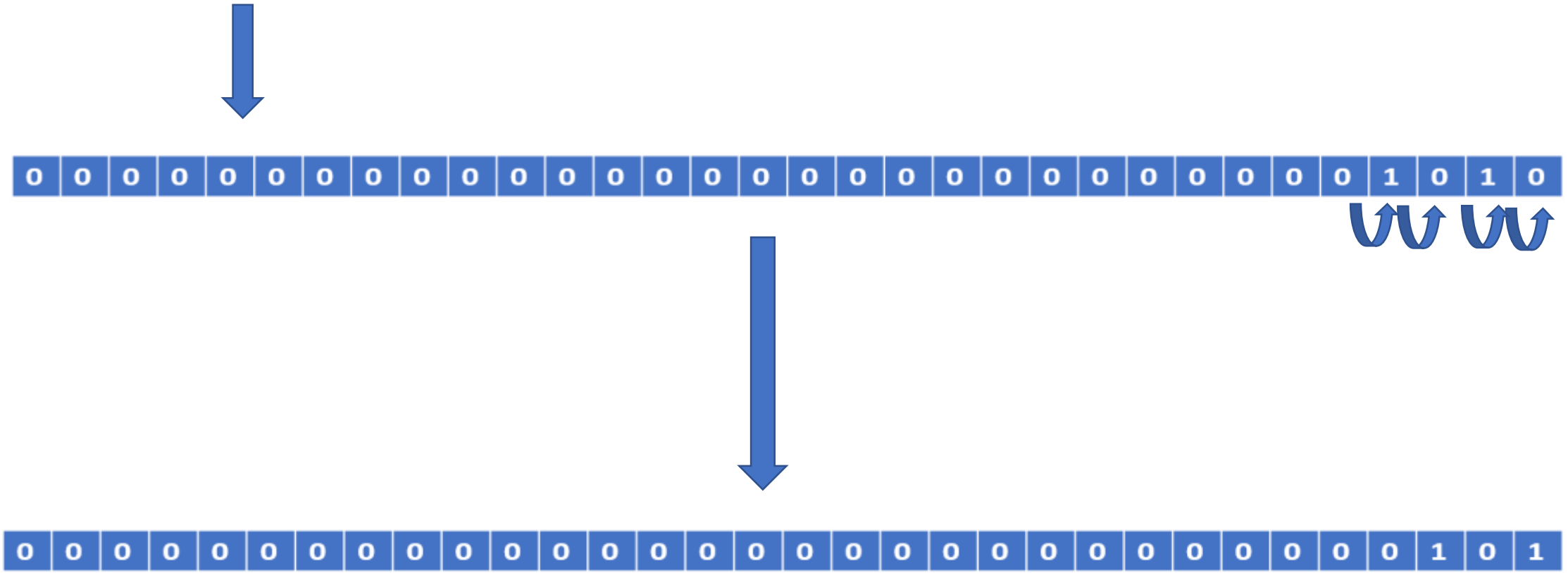
- Shifts the bits of the number to the right and fills 0 on voids left as a result.
- The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.

Ex: int a=10;

System.out.println(10>>1);

Output: **5**

➢int a=10;


0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1

Left shift(<<):

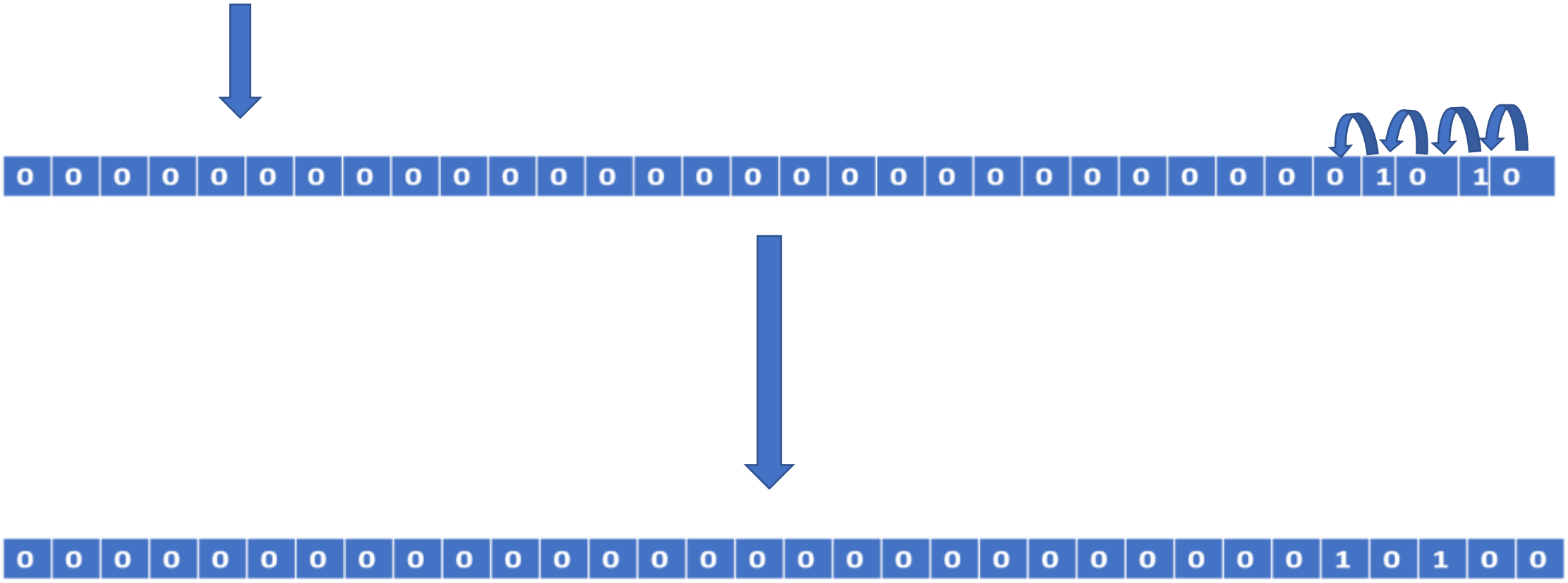
Shifts the bits of the number to the left and fills 0 on voids left as a result.

Ex: int a=10;

System.out.println(a<<1);

Output:20

int a=10;


0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0

Bitwise Operators:

Operator	Use	Operation
&	op1 & op2	Bitwise AND if both operands are numbers; conditional AND if both operands are boolean
|	op1 | op2	Bitwise OR if both operands are numbers; conditional OR if both operands are boolean
^	op1 ^ op2	Bitwise exclusive OR (XOR)
~	~op2	Bitwise complement

- When its operands are numbers, the **&** operation performs the bitwise **AND** function on each parallel pair of bits in each operand.
- The **AND** function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following table.

Bit in op1	Corresponding Bit in op2	Result=op1&op2
0	0	0
0	1	0
1	0	0
1	1	1

- **EX:**

```
1101     //13
  &
1100     //12
----------------
1100     //12
```

- *Inclusive or* means that if either of the two bits is 1, the result is 1. The following table shows the results of an *inclusive or* operation.

Bit in op1	Corresponding Bit in op2	Result=op1|op2
0	0	0
0	1	1
1	0	1
1	1	1

- *Exclusive or* means that if the two operand bits are different the result is 1; otherwise the result is 0.
- The following table shows the results of an *exclusive or* operation.

Bit in op1	Corresponding Bit in op2	Result=op1^op2
0	0	0
0	1	1
1	0	1
1	1	0

Logical Operators

- Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 **&&** expression2	true only if both expression1 and expression2 are true
|| (Logical OR)	expression1 || expression2	true if either expression1 or expression2 is true
! (Logical NOT)	**!**expression	true if expression is false and vice versa

Ex:

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

        // && operator
        System.out.println((5 > 3) && (8 > 5));   // true
        System.out.println((5 > 3) && (8 < 5));   // false

        // || operator
        System.out.println((5 < 3) || (8 > 5));   // true
        System.out.println((5 > 3) || (8 < 5));   // true
        System.out.println((5 < 3) || (8 < 5));   // false

        // ! operator
        System.out.println(!(5 == 3));   // true
        System.out.println(!(5 > 3));   // false
    }
}
```

Relational Operators

- Relational operators are used to check the relationship between two operands.

Operator	Description	Example
==	Is Equal To	3 == 5 returns **false**
!=	Not Equal To	3 != 5 returns **true**
>	Greater Than	3 > 5 returns **false**
<	Less Than	3 < 5 returns **true**
>=	Greater Than or Equal To	3 >= 5 returns **false**
<=	Less Than or Equal To	3 <= 5 returns **true**

Ex:

```
public class RelationalOpr {

        public static void main(String[] args) {
            // create variables
            int a = 7, b = 11;
            System.out.println("a is " + a + " and b is " + b);

            // == operator
            System.out.println(a == b);  // false

            // != operator
            System.out.println(a != b);  // true

            // > operator
            System.out.println(a > b);  // false

            // < operator
            System.out.println(a < b);  // true

            // >= operator
            System.out.println(a >= b);  // false

            // <= operator
            System.out.println(a <= b);  // true
        }
    }
```

Assignment Operators

- Assignment operators are used in Java to assign values to variables.
  Ex:

```
int age;
age = 5;
```

- Here, = is the assignment operator.
- It assigns the value on its right to the variable on its left. That is, 5 is assigned to the variable age.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Ternary Operator

- Java ternary operator is the only conditional operator that takes three operands.
- It’s a one-liner replacement for the if-then-else statement and is used a lot in Java programming.
- We can use the ternary operator in place of if-else conditions or even switch conditions using nested ternary operators.

Syntax:

variable = Condition ? Expression2: Expression3


If Condition is True
Execute Expression 1
Condition ? Expr 1 : Expr 2
If Condition is False
Execute Expression 2

Ex:

```
import java.util.Scanner;

public class Max
{
    public static void main(String[] args) {
        System.out.println("Enter num1,num2");
        Scanner scr=new Scanner(System.in);
        int num1=scr.nextInt();
        int num2=scr.nextInt();
        int max=num1>num2?num1:num2;
        System.out.println("Max number:"+max);
    }
}
```

Output:

```
Enter num1,num2
200
300
Max number:300
```

Control statements

Unit-II

Topics

- INHERITANCE AND POLYMORPHISM: Basic concepts, Types of inheritance, Member access rules, Usage of this and Super key word, Method Overloading, Method overriding, Abstract classes, Dynamic method dispatch, Usage of final keyword.
- PACKAGES AND INTERFACES: Defining package, Access protection, importing packages, Defining and Implementing interfaces, and Extending interfaces

Types of inheritance

What is Inheritance ?:

- The process of acquiring properties and behaviors from one class to another class is called Inheritance.

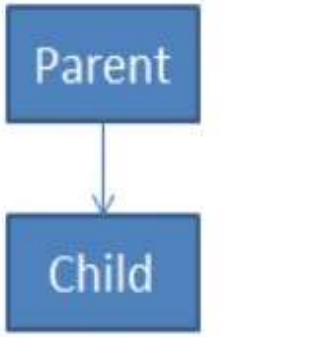
Properties	:	variables
Behaviours	:	methods

- The main purpose of the inheritance is code extensibility whenever we are extending automatically the code is **reused**.

- By using **extends** keyword we are achieving inheritance concept.

- Inheritance is also known as **is-a** relationship means two classes are belongs to the same hierarchy.

- In inheritance one class giving the properties and behavior and another class is taking the properties and behavior.
- In the inheritance the person who is giving the properties is called **parent ,**The person who is taking the properties is called **child**.



Types of Inheritance:

- ❖Single Inheritance
- ❖Multilevel Inheritance
- ❖Hierarchical Inheritance
- ❖Hybrid Inheritance
- ❖Multiple Inheritance

Single inheritance

- In single inheritance, subclasses inherit the features of one superclass.

- class A serves as a base class for the derived class B.

Multilevel Inheritance

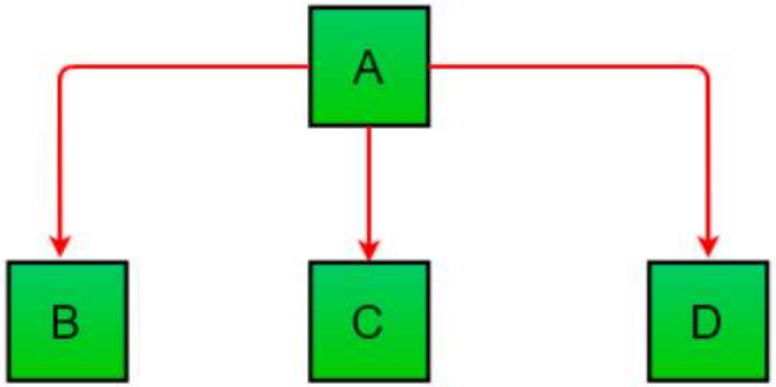
- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

Ex:

Hierarchical Inheritance

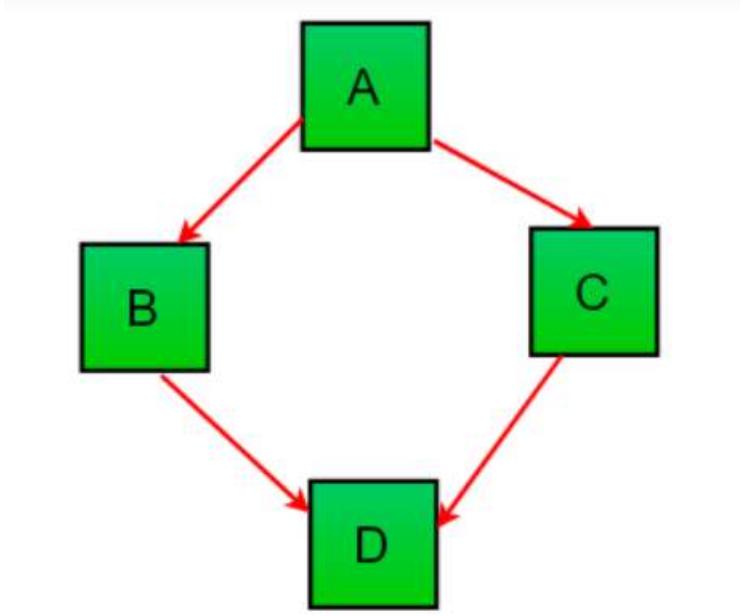
- In Hierarchical Inheritance, one class serves as a superclass for more than one sub class.

Ex:



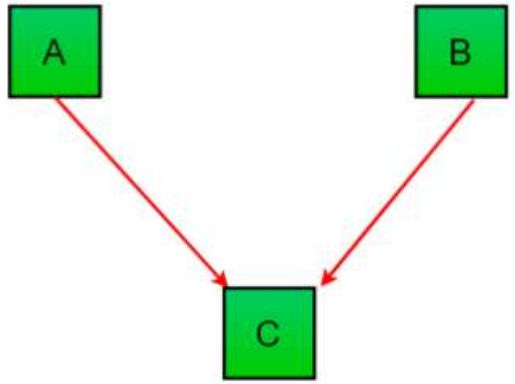
Hybrid Inheritance

- It is a mix of two or more of the above types of inheritance.
- Since java doesn’t support multiple inheritance with classes, the hybrid inheritance is also not possible with classes.



Multiple Inheritance

- In Multiple inheritance ,one class can have more than one parent class and inherit features from all parent classes.



- Java does **not** support **multiple inheritance** with classes.

Object class

- Every class in the java programming is a child class of **Object**.
- The base class for all java classes is **Object** class.
- The default package in the java programming is **java.lang** package.

Ex:

```
class Test
{
}
```

=====>The above class declaration is equal to below one

```
class Test extends Object
{
}
```

Member access rules

Member access rules

public:

- ❖Members with the public access modifier are accessible from anywhere, both within the class hierarchy and from outside the class.
- ❖ Public members of a superclass are inherited and can be accessed in the subclass.

protected:

- ❖Members with the protected access modifier are accessible within the same package and by subclasses, even if they are in a different package.
- ❖Protected members of a superclass are inherited by the subclass.

default:

- If no access modifier is specified (default access), members are accessible within the same package but not outside of it.
- Default members of a superclass are inherited by the subclass if they are in the same package.

private:

- Members with the private access modifier are only accessible within the class where they are declared.
- Private members of a superclass are not inherited by the subclass

this and super key word

this Keyword

this

- In Java, the "this" keyword is a reference to the current object(instance) of the class in which it is used.
- It can be used to refer to instance variables and instance methods of the current object within that object's scope.
- Here are a few common use cases for the "this" keyword in Java:
  - Accessing instance variables
  - Calling another constructor in the same class
  - Passing this for current object as an argument

Accessing instance variables using this

- We can use "this" to distinguish between instance variables and method parameters or local variables when they have the same name.

Ex:

```
class Student
{
    String name;
    String mobile;
    Student(String name,String mobile)
    {
        this.name=name;
        this.mobile=mobile;
    }
}
```

Calling another constructor in the same class using this

- When a class has multiple constructors, you can use "this" to call another constructor from the same class. This is useful for constructor overloading and code reuse.

```
class Employee
{
    private int value;

    public Employee() {
        this(0); // Calls the parameterized constructor with an
initial value of 0
    }

    public Employee(int value) {
        this.value = value;
    }
}
```

Passing this for current object as an argument

➢"this" can be used to pass the current object as an argument to a method or another constructor.

Ex:

```
public class Test
{
    int i=10;
    public static void main(String[] args) {
        Test t=new Test();
        t.m1();
    }
    public void m1()
    {
        System.out.println("m1- method is calling..");
       m2(this);
    }
    public void m2(Test t)
    {
        int i=100;
        System.out.println("Local var     i="+i);
        System.out.println("Instance var i="+t.i);

    }
}
```

super Keyword

super is a keyword in Java which is used to

- Call the Super class **variable**
- Call the super class **methods**
- Call the super class **constructor**
- The most common use of the super keyword is to eliminate the confusion and ambiguity between super classes and subclasses that have methods , Variables with the same name.

Calling Super class variable:

- If the child class and parent class has same data members (Variables). In that case there is a possibility of ambiguity for the JVM.

- To avoid above ambiguity we should use super keyword inside child class

Example:

```
class Parent
{
    int X=100;
    int Y=200;
}
class Child extends Parent
{
    int X=300;
    int Y=400;
    public void add()
    {
        System.out.println("Parent class value ="+(super.X+super.Y));
        System.out.println("Child class value ="+(X+Y));
    }
}
class SuperDemo
{
    public static void main(String[] args) {
        Child c=new Child();
        c.add();
    }
}
```

Calling super class methods

- If the child class and parent class has same data method names. In that case there is a possibility of ambiguity for the JVM.

- To avoid above ambiguity we should use super keyword inside child class

Calling super class methods:

```
class Parent
{
    int X=100;
    int Y=200;
    public int  add()
    {
        return X+Y;
    }
}
class Child extends Parent
{
    int X=300;
    int Y=400;
    public int  add()
    {
        return X+Y;
    }
    public void printResult()
    {
        System.out.println("Parent class vlaue:"+super.add());//Super method to call a method
        System.out.println("Child class vlaue:"+add());
    }
}
class Test
{
    public static void main(String[] args) {
        Child c=new Child();
        c.printResult();
    }
}
```

Calling the super class constructor

- **super()** method is used to call Super class(Parent class) constructor.
- **super()** method call must be inside constructor only. No other method is allowed to call.
- **super()** must be **first statement** of the constructor.
- Both this(), super() methods must be call inside the Constructor. But at a time only one is allowed to call either this() or super().

super() method is used to call Super class(Parent class) constructor.

```
public class Parent
{
    public Parent()
    {
        System.out.println("Parent class constructor:");
    }
}
class Child extends Parent
{
    public Child()
    {
        super();//calling parent class constructor
    }
    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

super() must be first statement of the constructor.

```
class Child extends Parent
{
    public Child()
    {
        System.out.println("Hello..");
        super();
    }
    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

super() method call must be inside constructor only. No other method is allowed to call.

```
class Child extends Parent
{
    public Child()
    {

    }
      public void m1()
      {
              super();

      }

    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

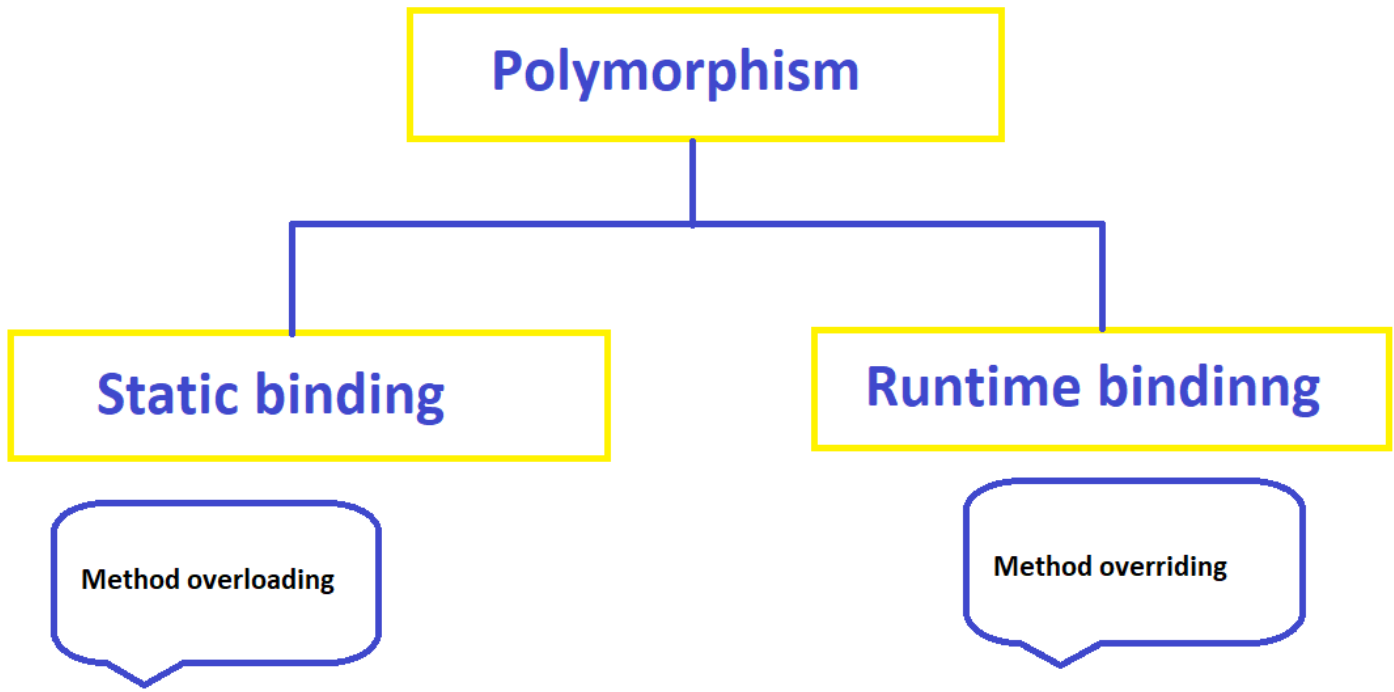
Both this(), super() methods must be call inside the Constructor. But at a time only one is allowed to call either this() or super().

```
class Child extends Parent
{
    public Child()
    {
        this();
        super();
    }
    public static void main(String[] args) {
        Child c=new Child();
    }
}
```

Polymorphism

Polymorphism?

- Poly=Many
- Morph=forms
- **Polymorphism** means "many forms", Performing single task in many ways.


Polymorphism
Static binding
Runtime bindinng
Method overloading
Method overriding

Method Overloading

Method overloading:

- It is a process of rewriting a method with different signatures within the same class is called Method overloading.

- Two methods are said to be overloaded methods if and only if two methods are having same name but different argument list.

- Method overloading can be done in one class.

- We can overload any number of methods

- The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

- Method resolution takes care by the Compiler.

- This process also called Compile time polymorphism (or) Static binding (or) Early binding.

Ex:

```
public class MethodOverloading
{
    public int getSum(int a,int b)
       {
        return a+b;
       }

      public float getSum(float a,float b)
       {
        return a+b;
       }
}
```

Method overriding

Method overriding:

- Rewriting a parent class method in child class is called Method overriding.

- Overriding concept is possible if and only if two classes should be in Parent – Child relationship.

- In Method overriding **JVM** is the responsible for Method resolution based on object type.

- This process also called as Dynamic binding (or) Late binding (or) Dynamic polymorphism.

Rules for overriding

- In method overriding method name and method signature should be same.

- **Return type:** Must be same (optionally covariant type)

- **Modifier :** Method scope should not decrease

- Scope order as follows : **public>protected>default>private**

- Possible cases:
  - public → public
  - protected → protected, public
  - default → default, protected, public
  - private → ✗ **(Private methods are can not be override)**

- **Static methods**: For static Methods overriding concept is not applicable. But Method hiding is possible.
- In Method hiding compiler is the responsible for method resolution.

Ex: Method hiding

```
class Parent
{
     public static void m1()
    {
        System.out.println("Parent class method");
    }

}
class Child extends Parent
{
    public   static void m1()
    {
        System.out.println("Child class method");
    }
}
```

- **final** Methods: final Methods cannot be overridden..

- final → final ,Non final (Overriding is not possible)
- Non final → final (Overriding is possible)

Dynamic method dispatch

Dynamic method dispatch

Usage of final keyword

- **final is a** keyword or **modifier** applicable to
  - variables (for all instance, Static and local variables).
  - methods
  - classes

final Variables:

- If a variable is declared with ***final*** keyword, its value can't be modified, and we can make a variable as a constant.
- We must initialize a final variable, otherwise compiler will generates compile-time error.
- As per Java documentation final variables naming convention should be in all uppercase, use underscore to separate words.

Ex: public static final double PI=3.141592653589793

Initializing a final variable :

There are three ways to initialize a final variable :

- You can initialize a final variable when it is declared. This approach is the most common
- A blank final variable can be initialized inside instance block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
- A blank final static variable can be initialized inside static block.

Ex:

```
public class FinalVariable
{
    public final double PI=3.141592653589793;//Final Variable
                                              initialization at declaration
    public final int I;
    public final int J;
    public static final int DATABASE_VERSION;
    FinalVariable()
    {
        I=10;// Final Variable initialization inside constructor block
    }
    {
        J=20;// Final Variable initialization inside instance block
    }
   static  {
        DATABASE_VERSION=2;// Final Variable initialization
                           inside static block

    }
}
```

final Class

- When a class is declared with ***final*** keyword, it is called a final class.

- If a class is declared as final, then we cannot inherit that class i.e., we cannot create any child class for that final class.

  Ex: You can not create child class to the **String** class . Because String is the **final class.**

Ex:

```
final class Parent
{

}
class Child  extends Parent
{
    //Child class is not possible
    //Compile time error
}
```

- **Every method present inside a final class is always final by default but every variable present inside the final class need not be final.**

Example:

```
final class Demo
{
    int a=10;
    void m1()
{
    System.out.println("m1 method is final");
    System.out.println(a=a+1);
}
    public static void main(String[] args)
    {
        Demo d=new Demo();
        d.m1();
    }
}
```

final methods

- If a method is declared with *final* keyword, it is called a final method. A final method **cannot be overridden**.

- If you want to restrict implementation of a method then you can declare it as final.

- For example in Object class we can override some methods like **equals(), toString()** but you cant override method like **wait(), notify(), notifyAll()** because these methods are declared as final.

Eample:

```
class Parent
{
    public  void m1()
    {
        System.out.println("Parent class method ... m1()");
    }

    public final void m2()
    {
        System.out.println("It is method -2");
    }
  }

class Child  extends Parent
{
    public void m1()
    {
        System.out.println("Hello this is child class method..m1()");
    }

    public void m2()
    {
       //Compile thime error
    }

}
```

Note:

- The main **advantage** of **final** modifier is ,We can achieve **security** as no one can be allowed to change our implementation.

- But the main **disadvantage** of final keyword is we are missing key benefits of Oops like inheritance and polymorphism.

- Hence if you have specific requirement you can use but it never recommended to use final modifier.

Abstract classes

Abstraction

What is Abstraction?

- In Object-oriented programming, abstraction is a process of hiding the internal implementation details from the user, only the essential functionality will be provided to the user.

- Abstraction can be achieved with either **abstract classes** or **interfaces**

Normal methods vs Abstract Methods

Normal methods:

- Normal method contains declaration as well as method definition

Ex:

```
public void method()
{
   ---------
   --------body;
   ---------
}
```

Abstract methods:

- The method which is having declaration but not definition such type of methods are called abstract methods.
- Every abstract method should end with “**;**”.
- The child classes are responsible to provide implementation for parent class abstract methods.

Ex: abstract void method(); //abstract method

Note: The methods marked abstract end in a semicolon rather than curly braces.

Normal classes vs Abstract classes

Normal classes:

- Normal class is a java class contains only normal methods.

Ex:
```
class Test
    {
        void m1()
        {
          --
        }
        void m2()
        {
          --
        }
    }
```

Abstract class:

- If a class contains at least one abstract method then it is a abstract class.
- To specify the particular class is abstract and particular method is abstract method to the compiler use **abstract** modifier.
- It is not possible to create an object to the Abstract class. Because it contains the unimplemented methods.
- For any class if we don’t want instantiation then we have to declare that class as abstract i.e., for abstract classes instantiation (creation of object) is not possible.

Ex:

```
abstract class Test
{
    abstract public void m1();
    public void m2()
    {
    }

}
```

➢Abstract method (all) implementation should be done in Child class.

Ex:

```
abstract class Parent
{
    abstract public void m1();

}

class Demo extends Parent
{

    public void m1()
    {
        System.out.println("Method m1() implementation");

    }
}
```

- Even though class does not contain any abstract method still we can declare the class as abstract.
- Abstract class contain zero or more number of abstract methods.
- For abstract classes it is not possible to create an object

Ex:

```
abstract class Test
{
     void m1()
     {
          System.out.println("m1-method");
     }
     void m2()
     {
          System.out.println("m2-method");
     }
     public static void main(String[] args)
     {
          Test t=new Test()                    //Compile time Error
          t.m1();
     }
};
```

Interfaces

- Interface is also one type of class and It contains only abstract methods.

- All interface methods are implicitly public and abstract. In other words, you do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.

- For the every interface compiler will generates .class files

- Each and every interface by default abstract hence it is **not possible to create an object.**

- Interfaces not alternative for abstract class it is extension for abstract classes.

- Interface contains only abstract methods means unimplemented methods.
- Interface also called 100% Abstract class.
- Interfaces giving the information about the functionalities or Services And it will hide the information about internal implementation.
- To provide implementation for abstract methods we have to use **implements** Keyword
- For the interfaces also inheritance concept is applicable.

- By using **interface** keyword we can declare interfaces in Java

Syntax:

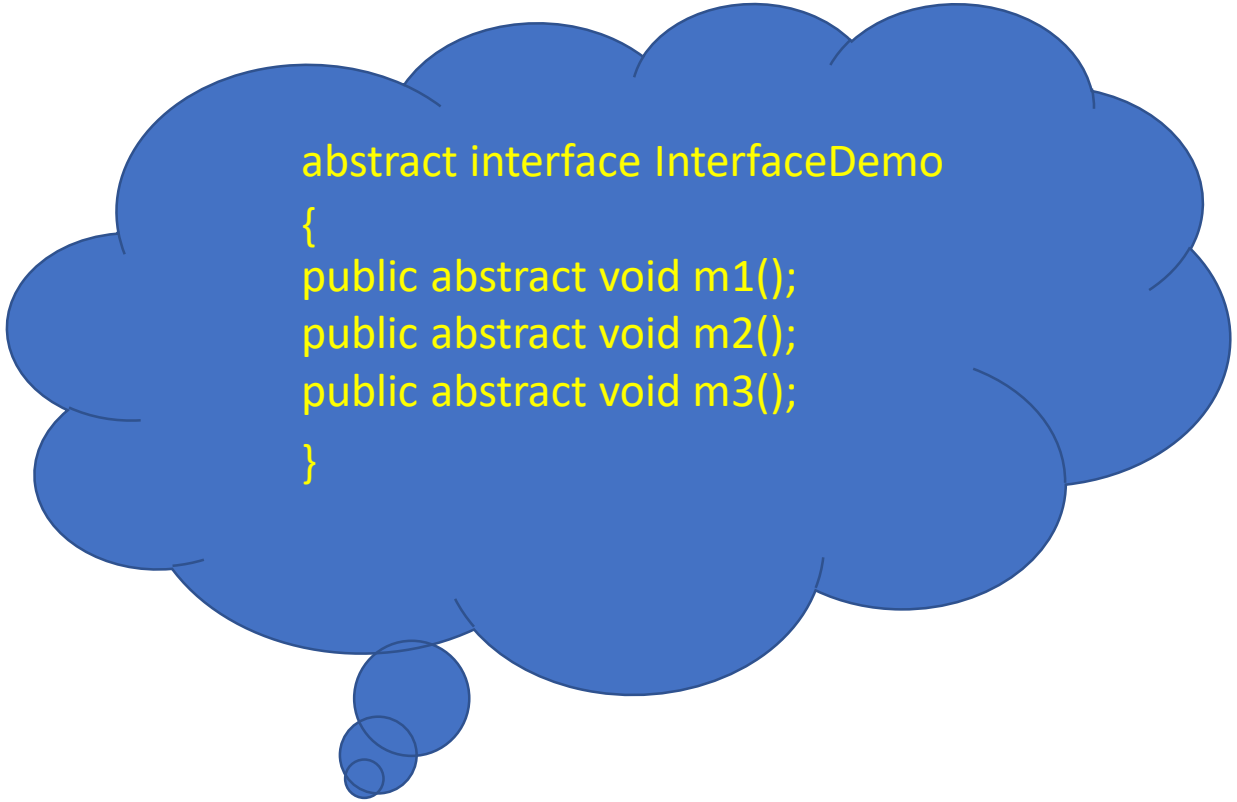
```
interface Interface_Name
  {
        --
  }
```

Ex:

```
interface Demo
{
        void m1();
}
```

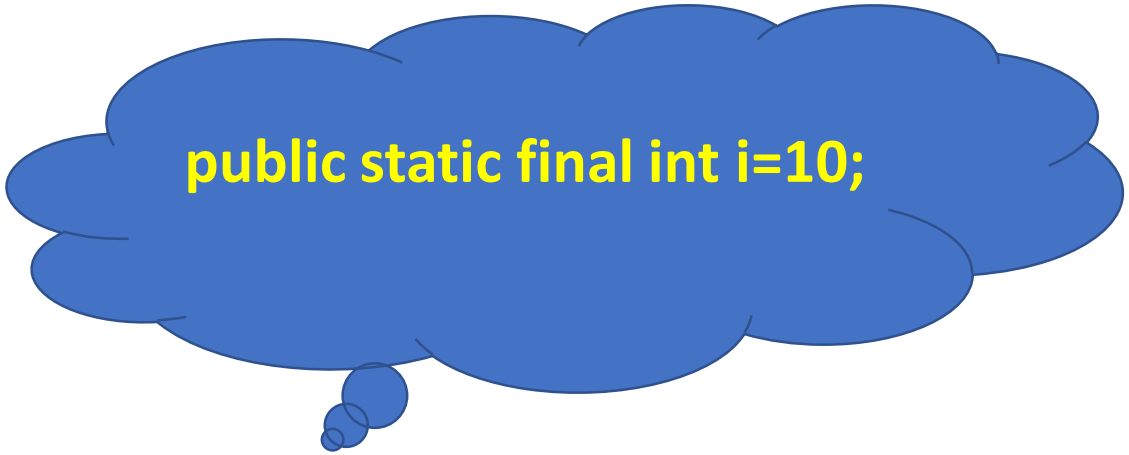
Ex:

```
interface InterfaceDemo
{
void m1();
void m2();
void m3();
}
```



➢Every variable in Interface by default public static final

```
interface InterfaceDemo
{
  int i=10;
}
```

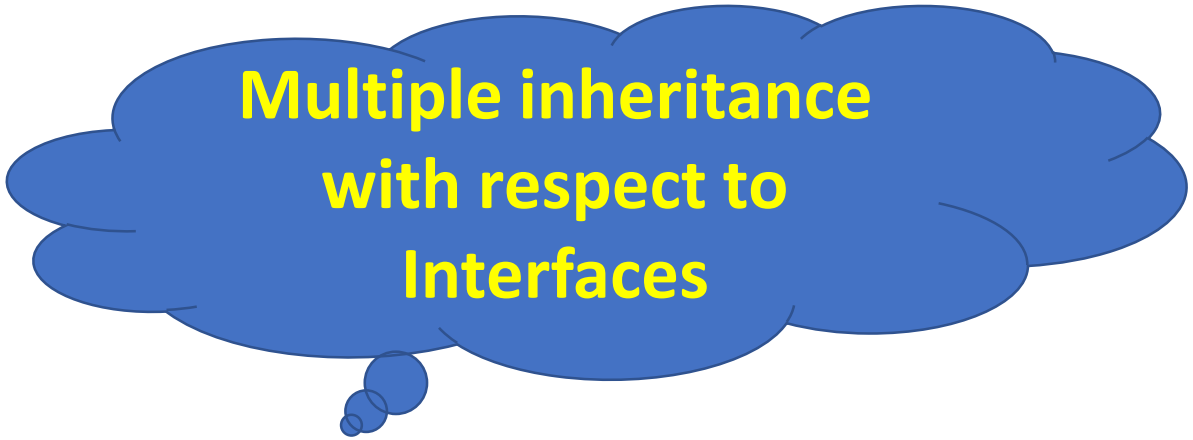



```
interface InterfaceDemo1
{
    public void method1();
    public void method2();
}
interface InterfaceDemo2 extends InterfaceDemo1
{
    public void method3();
    public void method4();
}
class Demo implements InterfaceDemo2
{
    @Override
    public void method1() {
        System.out.println("InterfaceDemo1-  method1() implementation");

    }
    @Override
    public void method2() {
        System.out.println("InterfaceDemo2-  method2() implementation");

    }
    @Override
    public void method3() {
        System.out.println("InterfaceDemo3-  method3() implementation");
    }
    @Override
    public void method4() {
        System.out.println("InterfaceDemo4-  method4() implementation");

    }
}
```




```
interface InterfaceDemo1
{
    public void method1();
}
interface InterfaceDemo2
{
    public void method2();
    public void method3();
}
class Demo implements InterfaceDemo1,InterfaceDemo2
{
    @Override
    public void method1() {
        System.out.println("InterfaceDemo1-  method1() implementation");
    }
    @Override
    public void method2() {
        System.out.println("InterfaceDemo2-  method2() implementation");
    }
    @Override
    public void method3() {
        System.out.println("InterfaceDemo3-  method3() implementation");
    }
}
```

Packages

Access control

Access modifiers:

- The access modifiers in Java talks about the accessibility or scope of a Variables ,Methods, Constructor, or Class.
- Depends on our requirements we can change the access level.
- In Java we have 4 types of access modifiers:
  - private
  - Default(No access modifier)
  - protected
  - public

private:

- Private modifier is applicable to methods , constructor and data members**(Variables).**
- private members are accessible only **within the class** in which they are declared. Out side of the class we cannot access.
- We cannot declare class with **private** modifier. But it is possible to declare for inner class.
- **private** modifier having less scope to comparative other access modifiers i.e, more restrictive.

default modifier(No modifier):

- If we are not specifying any access modifier for a class , method or data member ,It is said to be having the **default** access modifier by default.
- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

- **default** modifier having more scope than **private** and less scope than protected and public modifiers.

protected:

- The methods or data members declared as protected are **accessible within same package or sub classes in different package.**

- **protected** modifier having more scope than **default, private** modifiers and less scope than **public** modifier.

public:

- The public access modifier has the **more scope** among all other access modifiers.
- Classes, methods or data members which are declared as public are **accessible from every where** in the program.
- There is no restriction on the scope of a public data members.

Scope order:

public > protected > default > private

	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 non-subclass	No	No	No	Yes

Packages

Information regarding packages:

- The package contains group of related classes , interfaces, sub packages.
- The package is an **encapsulation mechanism** it is binding the related classes and interfaces.
- We can declare a package with the help of **package** keyword.
- Package is nothing but physical directory (folder) structure and it is providing clear-cut separation between the project modules.
- Whenever we are dividing the project into the packages(modules) the shareability of the project will be increased.

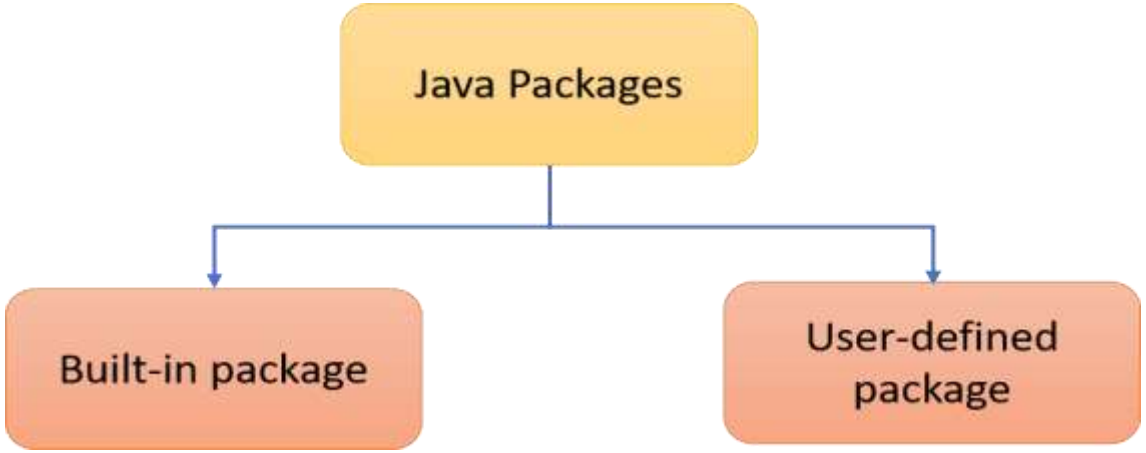
Syntax:

package package_name;

Ex: package com.mlrit;

Packages are divided into two types

- Predefined packages
- User defined packages



Predefined packages:

Java predefined packages contains all predefined classes and interfaces.

Ex:

java.lang ,

Java.io,

Java.awt ,

Java.util

Java.net.. etc.

Java.lang:

- The most commonly required classes and interfaces to write a sample program is encapsulated into a separate package is called **java.lang** package.

- It is a default package , No need to import this package.

Ex: String(class)

StringBuffer(class)

Object(class)

Runnable(interface)

Cloneable(nterface)

Java.io :

- The classes which are used to perform the input output operations that are present in the **java.io** packages.

Ex:

FileInputStream(class)
FileOutputStream(class)
FileWriter(class)
FileReader(class)
Serializable(Inteface)

java.net :

The classes which are required for connection establishment in the network that classes are present in the **java.net** package.

Ex:

HttpURLConnection
Socket
URL
ServerSocket
InetAddress
SocketOptions (Interface)

Java.util

- Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes.

Ex:

Calender
Date
Scanner
Arrays
ArrayList
Collection<E> (Interface)
Iterator<E> (Interface)
.. etc

java.sql

- Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java™ programming language.

Ex:

Date

DriverManager

Blob (Interface)

java.awt:

The classes which are used to prepare graphical user interface those classes are present in the **java.awt** package.

Ex: Button (class)

Checkbox(class)

Choice (Class)

List (class)

ActiveEvent (Interface)

User defined packages:

- The packages which are defined by the user are called user defined packages.
- In the single source file it is possible to take the only one package. If we are trying to take two packages at that time compiler raise a compilation error.
- In the source file it is possible to take single package statement.
- While taking package name we have to follow some coding standards.

Rules to follow while writing package:

- The package name is must reflect with your organization name and package name is reverse of the organization domain name.

Domain name: **www.example.com**

Package name: **package com.example**;

- The package must be the first statement of the source file and it is possible to declare **at most one** package within the source file .

- The import statement must be in between the package and class statement. And it is possible to declare **any number of import statements** within the source file.

- The class declaration must be after package and import statement and it is possible to declare any number of class within the source file.

- It is possible to declare **at most one public class.**
- It is possible to declare any number of non-public classes.
- The package and import statements are applicable for all the classes present in the source file.
- It is possible to declare comments at beginning and ending of any line of declaration it is possible to declare any number of comments within the source file.

Advantages of Packages:

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package removes **naming collision** or **naming conflicts**.
- Packages provide **reusability** of code .
- Java package provides **access protection**.

Path and Class Path:

- Path and Classpath both are operating system level environment variables.

- Path is used define where the system can find the **executables (.exe) or bin** files.

- Classpath is used to specify the location **.class** files.

Static import

- Static import allows you to access the static member of a class directly without using the fully qualified name.

Example:

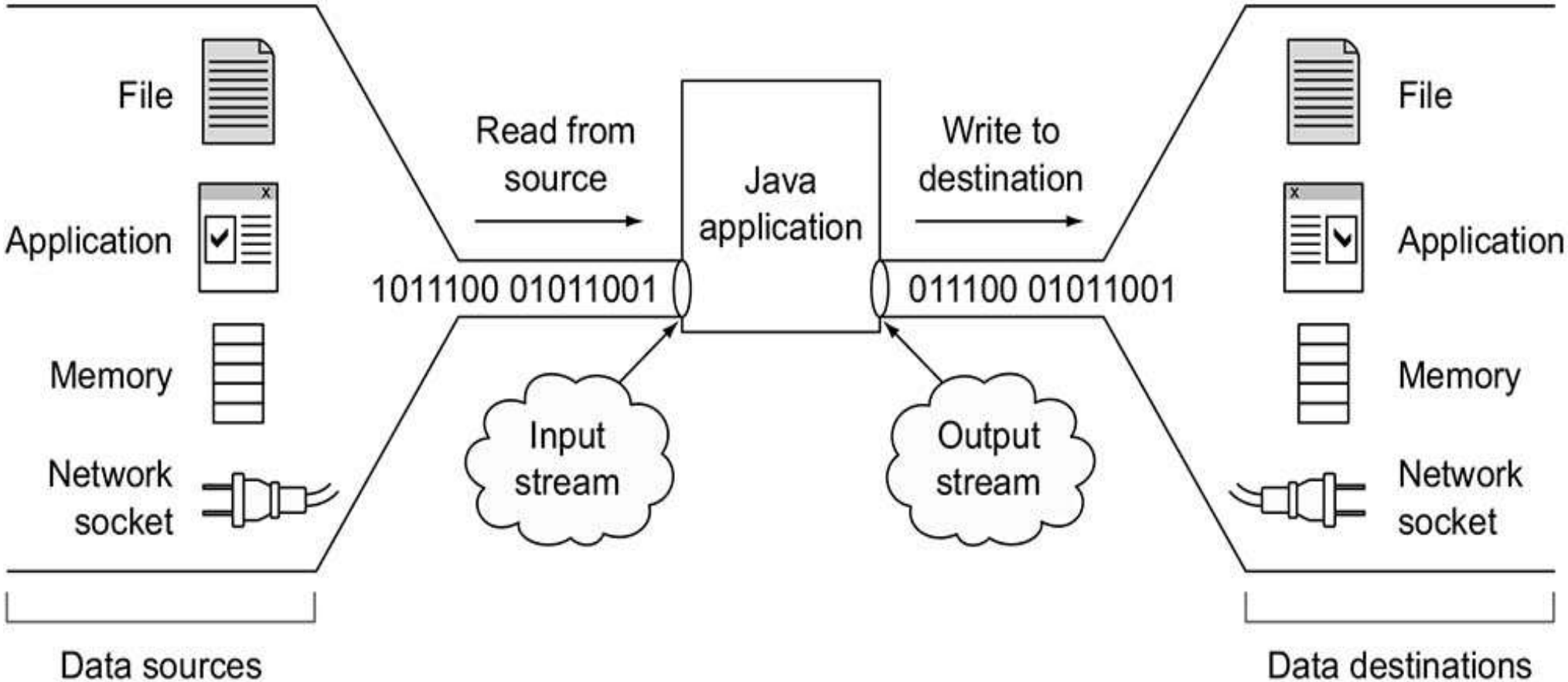
```
import static java.lang.Math.*;
class Test
{
    public static void main(String[] args)
    {
        System.out.println(sqrt(16));
        System.out.println(min(20,30));
    }
}
```

Unit-III

IO Streams

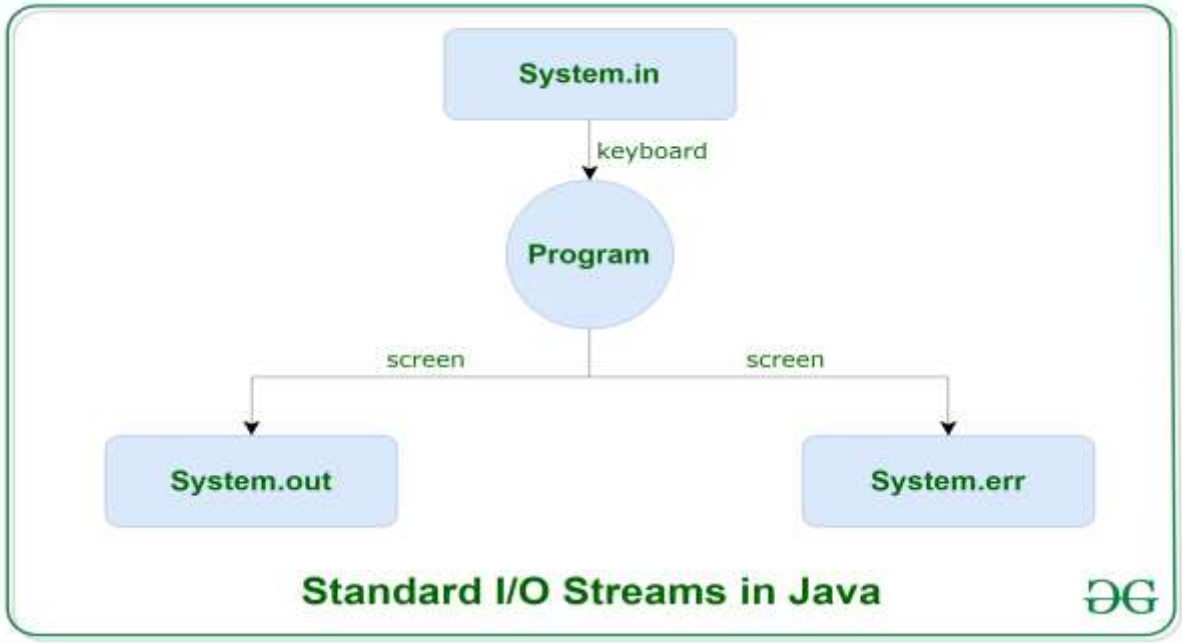
Stream?

- A stream is a sequence of data.
- In Java, Stream is a channel or a path along which data flows between source and destination.
- Java brings various Streams with its **I/O package** that helps the user to perform all the input-output operations.
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.


File
Application
Memory
Network socket
Data sources
Read from source
1011100 01011001
Java application
Input stream
Write to destination
011100 01011001
Output stream
File
Application
Memory
Network socket
Data destinations

Standard(Default) IO Streams in Java

- There are three standard IO Streams in Java
  - ❖System.in
  - ❖System.out
  - ❖System.err



Types of Streams

- Depending on the **type of operations**, streams can be divided into two primary classes.
  - InputStream
  - OutputStream

InputStream

- These streams are used to read data that must be taken as an input from a source array or file or any peripheral device.

- The Java **InputStream** class is the base class (superclass) of all input streams in the Java IO API.

- Each subclass of InputStream typically has a very specific use, but can be used as an InputStream.

Example:

- FileInputStream,
- BufferedInputStream,
- ByteArrayInputStream etc.

OutputStream

- These streams are used to write data as outputs into an array or file or any output peripheral device.

- The Java **OutputStream** class is the base class (superclass) of all output streams in the Java IO API.

- Each subclass of OutputStream typically has a very specific use, but can be used as an OutputStream

Example:

- FileOutputStream,
- BufferedOutputStream,
- ByteArrayOutputStream etc.

Types of Streams(Based on file types)

- Based on file types Streams can be divided into two primary classes

  - ByteStream
  - CharacterStream

ByteStreams:

- Programs use byte streams to perform input and output of (8-bit) bytes.
- All byte stream classes are descended from **InputStream** and **OutputStream**.

CharacterInputStream:

- Java **Character** streams are used to perform input and output for 16-bit Unicode.

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

File Handling

- File handling in Java implies reading data from the file and writing data to a file.
- The File class from the **java.io** package, allows us to work with different formats of files.
- In order to use the File class, you need to create an object of the class and specify the filename or directory name.

Ex:

```
// Import the File class
import java.io.File;
// Specify the filename
File obj = new File(“sample.txt");
```

File and Directory

- A file is a named location that can be used to store related information.
- Ex: main.java is a Java file that contains information about the Java program.
- A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

File handling methods

Method	Type	Description
canRead()	Boolean	It tests whether the file is readable or not
canWrite()	Boolean	It tests whether the file is writable or not
createNewFile()	Boolean	This method creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	It tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

Example: Creating a File

```
import java.io.IOException;
public class FileDemo
{
   public static void main(String[] args)
   {
      File f=new File("sample.txt");
      if(f.exists())
      {
         System.out.println("File already existed..");
      }
      else
      {
         try
         {
            f.createNewFile();
            System.out.println("File created successfully..");
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }
}
```

Example: No of Files and Directories in a Directory

```
import java.io.File;
public class Ex2
{
    public static void main(String[] args)
    {
        File f1=new File(".");
        String[] list=f1.list();
        int count=0;
        for (String name:list)
        {
            count++;
            File f2=new File(name);
            if(f2.isFile())
            {
               System.out.println("File      :"+name);
            }
            else if (f2.isDirectory())
            {
                System.out.println("Directory:"+name);
            }

        }
        System.out.println("No of files and dir :"+count);
    }
}
```

Example: File Information

```
import java.io.File;
import java.util.Scanner;
public class Ex3
{
    public static void main(String[] args) {
        System.out.println("Enter your file name");
        Scanner scr=new Scanner(System.in);
        String fname=scr.next();
        File f=new File(fname);
        if(f.exists())
        {
            System.out.println("Size of the file :"+f.length());
            System.out.println("Absolute path    :"+f.getAbsolutePath());

            if(f.canRead())
                System.out.println(f.getName()+":  Is readable file");
            else
                System.out.println(f.getName()+":  Is not readable file");

            if(f.canWrite())
                System.out.println(f.getName()+":  Is writeable file");
            else
                System.out.println(f.getName()+":  Is not writeable file");

            if(f.canExecute())
                System.out.println(fname+":  Is Executable file");
            else
                System.out.println(fname+":  Is not Executable file");
        }
        else
        {
            System.out.println("No such file existed..");
        }
    }
}
```

Example: Deleting Specified File

```
import java.io.File;
import java.util.Scanner;
public class Ex4
{
    public static void main(String[] args)
    {
        System.out.println("Enter your file name");
        Scanner scr=new Scanner(System.in);
        String fname=scr.next();
        File f=new File(fname);
        if(f.exists())
        {
            if(f.delete())
            {
                System.out.println(f.getName()+" is deleted successfully..");
            }
            else
            {
                System.out.println(f.getName()+" is not deleted..");
            }
        }
        else
        {
            System.out.println("No such file existed..");
        }
    }
}
```

Exception Handling

What is an Exception?

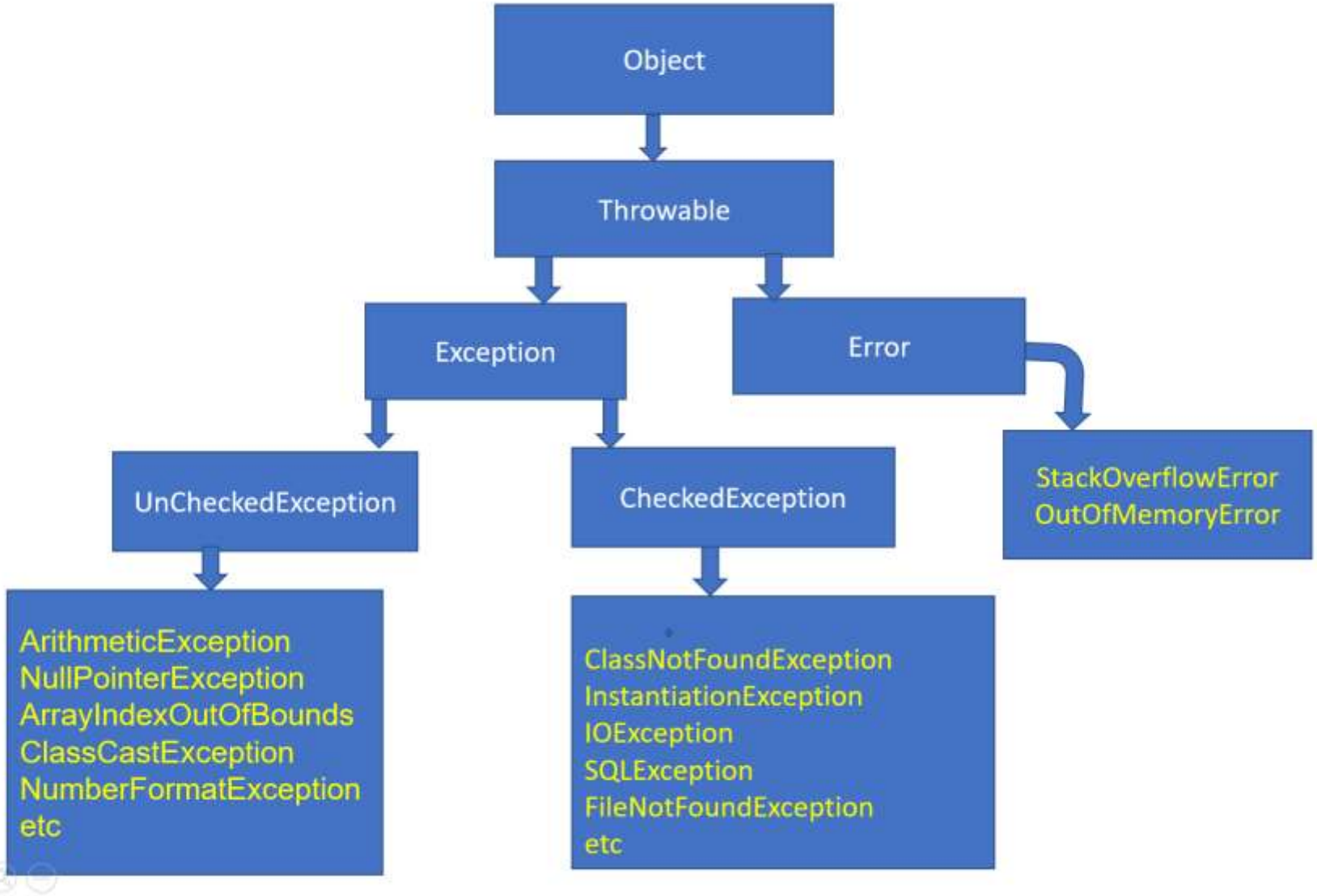
- It is an unexpected unwanted event which disturbs entire execution flow of the program.

Ex:

- SleepingException
- TirePuncharedException
- PowerCutException
- ArithmaticException

- Exception also called as **runtime error**.

Exceptions Hierarchy



- All exceptions and errors are sub classes of class **Throwable.**
- **Throwable** class having two child classes.
  - **Error**
  - **Exceptions**

- Both **Error and Exceptions** are present in **java.lang** package.

Error

- **Error** is an event caused by lack of System resources.
- These are exceptional conditions that are external to the application, and that the application usually cannot anticipate .
- Which cannot get recovered by any handling techniques.
- It surely cause termination of the program abnormally.
- Errors belong to **unchecked** type and mostly occur at runtime.

Ex:

- OutOfMemoryError
- StackOverFlowError

Exception:

- An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- These are recoverable by using programing techniques.

- Exceptions are divided into two types
  - UnCheckedExceptions (RuntimeExceptions)
  - CheckedExceptions (CompileTimeExceptions)

UnChecked Exceptions

- Also called as Runtime Exceptions
- Compiler does not check this type of Exceptions
- This type of programs are not connected to External resources(like files, printers, scanner).
- Exception handling is **optional** i.e, If we handle the exceptions program terminates normally, If not it leads to abnormal termination.

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.(out of range)
InputMismatchException	If we are giving input is not matched for storing input.
ClassCastException	If the conversion is Invalid.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.

Examples for UnChecked Exceptions:

i) ArrayIndexOutOfBoundException:

```
int a[]={10,20,30}
System.out.println(a[3]); // Array index out of bound exception
```

ii) NumberFormatException:

```
String s="ten"
int i=Integer.parseInt(s);
System.out.println(i); // Number Format Exception
```

iii) Arithmetic Exception:

```
System.out.println(100/0);
```

iv)NullPointerException:

```
String s=null;
System.out.println(s); // Null Pointer Exception
```

v) IllegalArgumentException

```
public class Ex1
{
    public static void main(String[] args)
    {
        for (int i=0;i<10;i++)
        {
            System.out.println("Hello");
            try {
                Thread.sleep(-10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
}
```

Checked Exceptions

- The Exceptions which are checked by the compiler at compilation time for the proper execution of the program at runtime is called Checked Exceptions.

- Also called compile time exceptions.

- External resources (like files , printers , scanners) may connected to the programs.

- Exception handling is mandatory for this type of Exceptions . If not handled even **.class** file wont generate.

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

Exception handling?

- Exception handling means it is not repairing an exception we are providing alternative way to continue rest of the program normally.

- The program must be graceful termination.

- There are two types of Exception handling techniques.
  - Default Exception Handling
  - Customized Exception Handling

Default Exception Handling

- When ever an exception raised in the method in which it is raised is responsible for the preparation of exception object by including the following information.
  - Name of Exception.
  - Description.
  - Location of Exception.

- After preparation of Exception Object, The method handovers the object to the JVM.

- JVM will check for Exception handling code in that method

- If the method doesn’t contain any exception handling code then JVM terminates that method abnormally and removes corresponding entry from the stack.

- This process will be continued until **main()** method.

- If the main method also doesn’t contain any exception handling code then JVM terminates main method abnormally.

- Just before terminating the program JVM handovers the responsibilities of exception handling to default exception handler.

- Default exception handler prints the error in the following format.
  - Name of Exception
  - Description stackTrace

Ex:

```
class Test
{
    public static void main(String[] args)
    {
        doStuff();
    }
    public static void doStuff()
    {
        doMoreStuff();
    }
    public static void doMoreStuff()
    {
        System.out.println(10/0);
    }
}
```

O/P:-

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo.doMoreStuff(ExceptionDemo.java:30)
        at ExceptionDemo.doStuff(ExceptionDemo.java:25)
        at ExceptionDemo.main(ExceptionDemo.java:21)
```

Exception Handling:

- Providing alternative way to execute rest of the program normally.
- Exception Handling is normal Execution of the program or graceful termination of the program at runtime.
- Exception class present in **java.lang** package.
- We can handle the exceptions in two ways.
  - By using try-catch blocks
  - By using throws keyword.

In Java we have 5 key words to handle the Exceptions:-

- try
- catch
- finally
- throw
- throws

Exception handling by using try-catch block:

- In Exception Handling **try** block contains **risky code** of the program and **catch** block contains handling code of the program.
- **Catch** block code is a alternative code for Exceptional code. If the exception is raised the alternative code is executed fine then rest of the code is executed normally.

Syntax:

```
try
{
    Risky code;
}
Catch(ExceptionName reference_variable)
{
    Alternative code if Exception raised;
}
```

Without try-catch

```
class ExceptionDemo
{
    public static void main(String[] args) {
        System.out.println("statement-1");
        System.out.println("statement-2");
        System.out.println("statement-3");
        System.out.println(10/0);
        System.out.println("statement-4");
        System.out.println("statement-5");
        }
}
```

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
statement-1
statement-2
statement-3
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at ExceptionDemo.main(ExceptionDemo.java:7)
```

With try-catch

```
class ExceptionDemo
{
    public static void main(String[] args) {
        System.out.println("statement-1");
        System.out.println("statement-2");
        System.out.println("statement-3");
        try {
            System.out.println(10/0);
        }
        catch (ArithmeticException e)
        {
           System.out.println("Divide / Zero exception");
        }
        System.out.println("statement-4");
        System.out.println("statement-5");
        }
}
```

output:

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
statement-1
statement-2
statement-3
Divide / Zero exception
statement-4
statement-5
```

Multiple catch() blocks:

- 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.
- 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.

Syntax:

```
try{
}
catch(Exception1 e)
{
}
catch(Exception2 e)
{
}
.
.
etc
```

finally:

- It is never recommended to write clean up code in try block. Because try block may execute or may not execute.

- And never recommended to use catch block for clean up code , because if there is no exception catch block wont execute.

- The finally keyword is used in association with a try/catch block and guarantees that a section of code will be executed, even if an exception is thrown.

- The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

- Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

- **Important:** The finally block is a key tool for preventing resource leaks

Possible combinations (try-catch()-finally())

- Try-catch() ➔ Allowed
- Try-finally() ➔ Allowed
- Try-catch-finally() ➔ Allowed
- Try-finally()-catch() ➔ Not allowed
- Catch()-finally() ➔ Not allowed

throw:

- The main purpose of the **throw** keyword is to creation of **Exception object explicitly** either for predefined or user defined .

- Throw keyword works like a try block. The difference is try block is automatically find the situation and creates a Exception object implicitly. Whereas throw keyword creates a Exception object explicitly.

- **throw** keyword must call within the method.

- By using throw keyword we can throw **only one Exception object** at a time.

Example:

```
if (withdrawal>balance)
        {
            throw new InsufficientFunds("No funds..");
        }
```

Creating user defined Exceptions

- In Exception Handling user can defined their own Exceptions.

- To create user defined Exceptions we need to create a child class to **RuntimeException** class or **Exception** class.

- Each and every Exception contains two constructors
  - default constructor
  - parameterized constructor

- **Naming convention**: Every user defined exception name must be suffix of **Exception**

Ex:

InsufficientFundsException

Example:

```
public static void checkBalance(double withdrawal)
  {
     double balance=1234567.50;
     if (withdrawal>balance)
     {
        throw new InsufficientFunds("No funds..");
     }
     else
     {
       balance= balance-withdrawal;
       System.out.println("You're A/C balance : "+balance);
     }
  }
}
class InsufficientFunds extends RuntimeException
{
  InsufficientFunds(String str)
  {
    super(str);
  }
}
```

throws:

- If any Checked Exception raised in a program that must be handle by **try-catch** or **throws** keyword.

Ex:

```
class ExceptionDemo
{
    public static void main(String[] args)
    {
        Thread.sleep(100);    // Checked Exception
    }
}
```

```
ExceptionDemo.java:5: error: unreported exception InterruptedException; must be caught or declared to be thrown
        Thread.sleep(100);
                     ^
1 error
```

Handling Exception with try-catch:

Ex:

```
public static void main(String[] args)
{
    try
    {
        Thread.sleep(100);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

Handling Exception with throws keyword:

Ex:

```
public static void main(String[] args) throws InterruptedException
{
    Thread.sleep(100);
}
```

- The main objective of the **throws** keyword is to delegate responsibilities to the caller method about Exception handling.
- **throws** keyword bypass the Exception but it doesn't prevent abnormal termination.

- By using throws keyword we can throw multiple Exceptions at a time.
- We can use throws keyword in Method declaration.
- **throws** keyword is applicable only Throwable objects but not Normal objects.
- throws keyword is required only for **checked exception** and usage of throws keyword for unchecked exception is meaningless.

Differences b/w throw and throws

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Unit-IV

Multithreading

Multitasking

- Executing multiple tasks at a time is called Multi tasking.
- (Or)
- Ability to execute more than one task at the same time is known as multitasking.

- There are two types of Multi tasking
  - Process based multitasking (Multitasking)
  - Thread based multitasking(Multithreading)

Process based multitasking

- In process based multitasking two or more processes and programs can be run concurrently.

- In process based multitasking a process or a program is the smallest unit.

Example:

- We can listen to music and browse internet at the same time. The processes in this example are the music player and browser.

Thread based multitasking

- In thread based multitasking two or more threads can be run concurrently.
- In thread based multitasking a thread is the smallest unit.

Example:

- While you are typing, multiple threads are used to display your document, asynchronously check the spelling and grammar of your document, generate a PDF version of the document.

Thread

- A thread is a flow of execution in a program.
- A thread is a part of the program.
- Thread is a tiny program running continuously. It is sometimes called as light-weight process.
- Thread class is defined in **java.lang** package

Multithreading v/s Multiprocessing

S.No	Multithreading	Multiprocessing
1	Thread is a fundamental unit of multithreading.	Process/Program is a fundamental unit of multiprocessing.
2	Multiple parts of single program gets executed in multithreading environment.	Multiple programs get executed in multiprocessing environment.
3	During multithreading the processor switches between multiple threads in the program.	During multiprocessing the processor switches between multiple programs/processes.
4	Cost effective because CPU can be shared among multiple threads at a time.	Expensive, because when a process uses CPU other process has to wait.
5	Highly efficient	Less efficient.
6	Develops efficient application programs	Develops efficient OS programs.

Defining Instantiating, Starting the Thread

We can define instantiate and starting a thread by using the following 2-ways.

- By extending **Thread** Class.
- By implementing **Runnable** interface.

By extending Thread Class

- We can create a thread by creating a child class to the **Thread** class.
- And we should override a method i.e, **run().**
- We can define a thread job inside **run().**
- By calling Thread class **start()** method we can start execution of a thread

Example:

```
class Mythread extends Thread
{
     @Override
     public void run()
    {
          super.run();
         for (int i=0;i<10;i++)
         {
              System.out.println("Thread is executing");
         }
     }
}
```

Thread Scheduler:

- If multiple threads are there then which thread will get chance first for execution will be decided by “**Thread Scheduler**”.

- Thread Scheduler is the post of JVM.

- The behavior of thread scheduler is vendor dependent and hence we can’t expect exact O/P for the program.

Difference between t.start() & t.run()

- In the case of t.start() a new thread will be created and which is responsible for the execution of run().

- But in the case of t.run() no new thread will be created and run() method will be executed just like a normal method by the main thread.

Importance of Thread Class start() method

- After Creating thread object compulsory we should perform registration with in the Thread scheduler.
- This will take care by start() of Thread class, So that the programmers has to concentrate on only job.
- With out executing Thread class start() method there is no chance of start a new Thread in java.

start()
{
- Register our thread with in the thread scheduler.
- Invoke run() method.

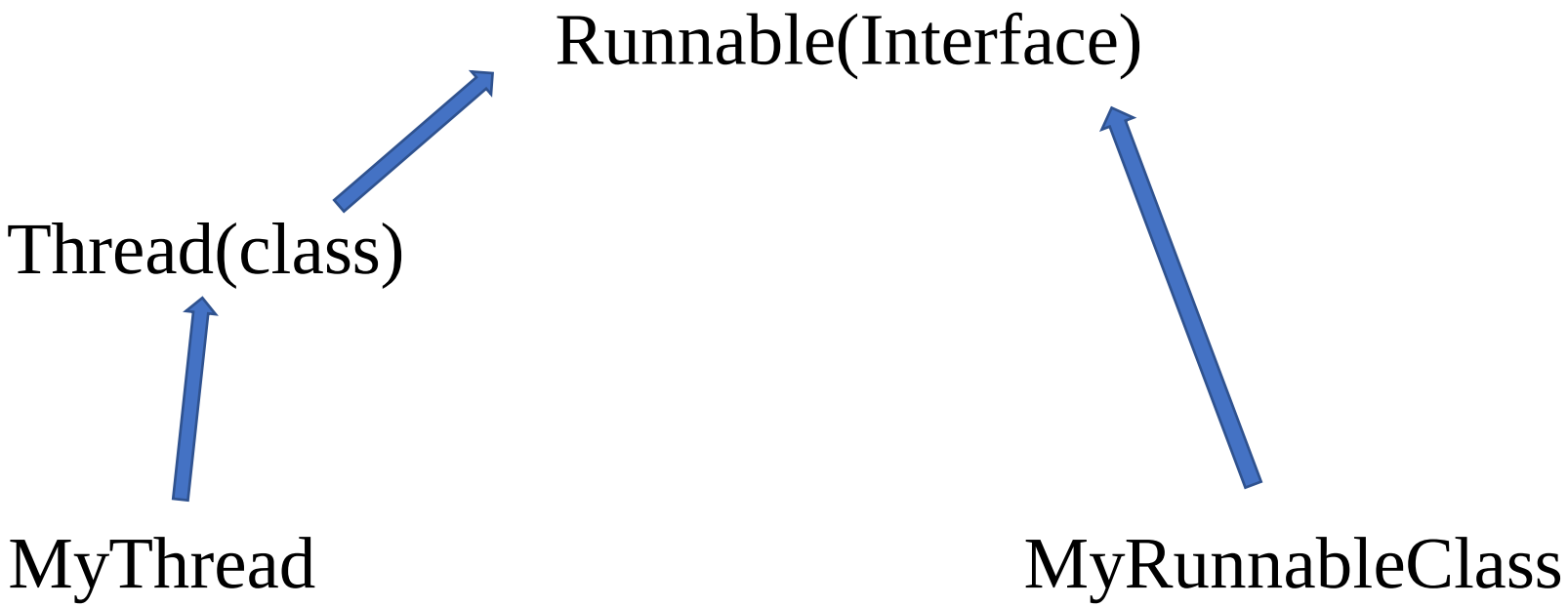
}

By Implementing Runnable Interface

- A Thread can be created by extending Thread class also. But Java allows only one class to extend, it wont allow multiple inheritance.

- So it is always better to create a thread by implementing Runnable interface. Java allows you to implement multiple interfaces at a time.

- By implementing Runnable interface, you need to provide implementation for run() method.


Runnable(Interface)
Thread(class)
MyThread
MyRunnableClass

- To run this implementation class, create a Thread object, pass Runnable implementation class object to its constructor. Call start() method on thread class to start executing run() method.

- Implementing Runnable interface does not create a Thread object, it only defines an entry point for threads in your object. It allows you to pass the object to the Thread(Runnable implementation) constructor.

Example:

```
class MyRunnableThread implements Runnable
{
    @Override
    public void run() {
        System.out.println("Runnable thread job");
    }
}

public class ThreadDemo
{
    public static void main(String[] args)
    {
        MyRunnableThread mythread=new MyRunnableThread();
        Thread t=new Thread(mythread);
        t.start();
    }
}
```

Thread lifecycle states

- A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java.
  - New
  - Runnable
  - Running
  - Non-Runnable (Blocked)
  - Terminated

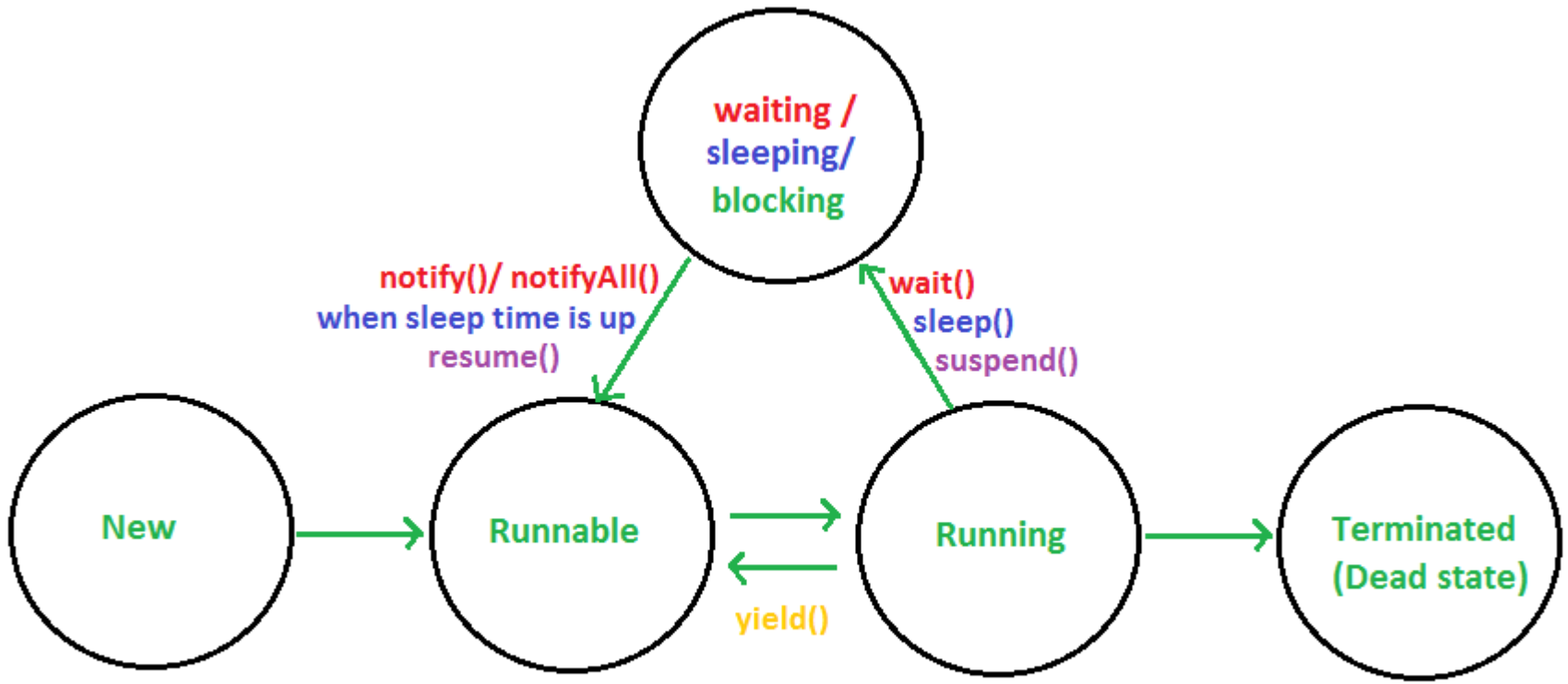


Fig. THREAD STATES

- **New:**

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

- **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.

- **Running:**

The thread is in running state if the thread scheduler has selected it.

- **Non-Runnable (Blocked):**

This is the state when the thread is still alive, but is currently not eligible to run.

- **Terminated:**

A thread is in terminated or dead state when its run() method exits.

Thread priorities

- Every Thread in java has some property.
- It may be default priority provided by the JVM or customized priority provided by the programmer.

- The valid range of thread priorities is 1 – 10. Where 1 is lowest priority and 10 is highest priority.

- The default priority of **main thread is 5**. The priority of child thread is inherited from the parent.

- Thread Scheduler will use priorities while allocating processor the thread which is having highest priority will get chance first and the thread which is having low priority.

- If two threads having the **same priority** then we can’t expect exact execution order it depends upon Thread Scheduler.

- The thread which is having **low priority** has to wait until completion of high priority threads.

Three constant values for the thread priority.

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5
- MAX_PRIORITY = 10

- Thread class defines the following methods to get and set priority of a Thread.
  - public final int getPriority()
  - public final void setPriority(int priority)

- Here ‘priority’ indicates a number which is in the allowed range of 1 – 10.

- Otherwise we will get Runtime exception saying “IllegalArgumentException”.

  - Ex: t.setPriority(11); // IllegalArgumentException

Thread class methods

sleep(long millis)

public static void sleep(long millis) throws InterruptedException

- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- The thread does not lose ownership of any monitors.

Parameters:

- millis - the length of time to sleep in milliseconds

Throws:

- **IllegalArgumentException :** If the value of millis is negative
- **InterruptedException :** If any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Example : sleep()

```
public class SleepDemo
{
public static void main(String[] args)
    {
        Child c=new Child();
        c.start();

    }
}
```

```
class Child extends Thread
{
    @Override
    public void run()

 {
        super.run();
        for (int i=0;i<10;i++)
        {
            System.out.println("Hello MLRIT
:"+i);
            try {
                sleep(1000);
            } catch (InterruptedException e)

{
                e.printStackTrace();
            }
        }
    }
}
```

Output:

Hello MLRIT :0

Hello MLRIT :1

…

Hello MLRIT :9

activeCount():

- This method is used to find out the number of threads in active state.

Synatx:

```
public static int activeCount();
```

- By default active count prints 2.Which means two threads will execute always i.e Main,Monitor

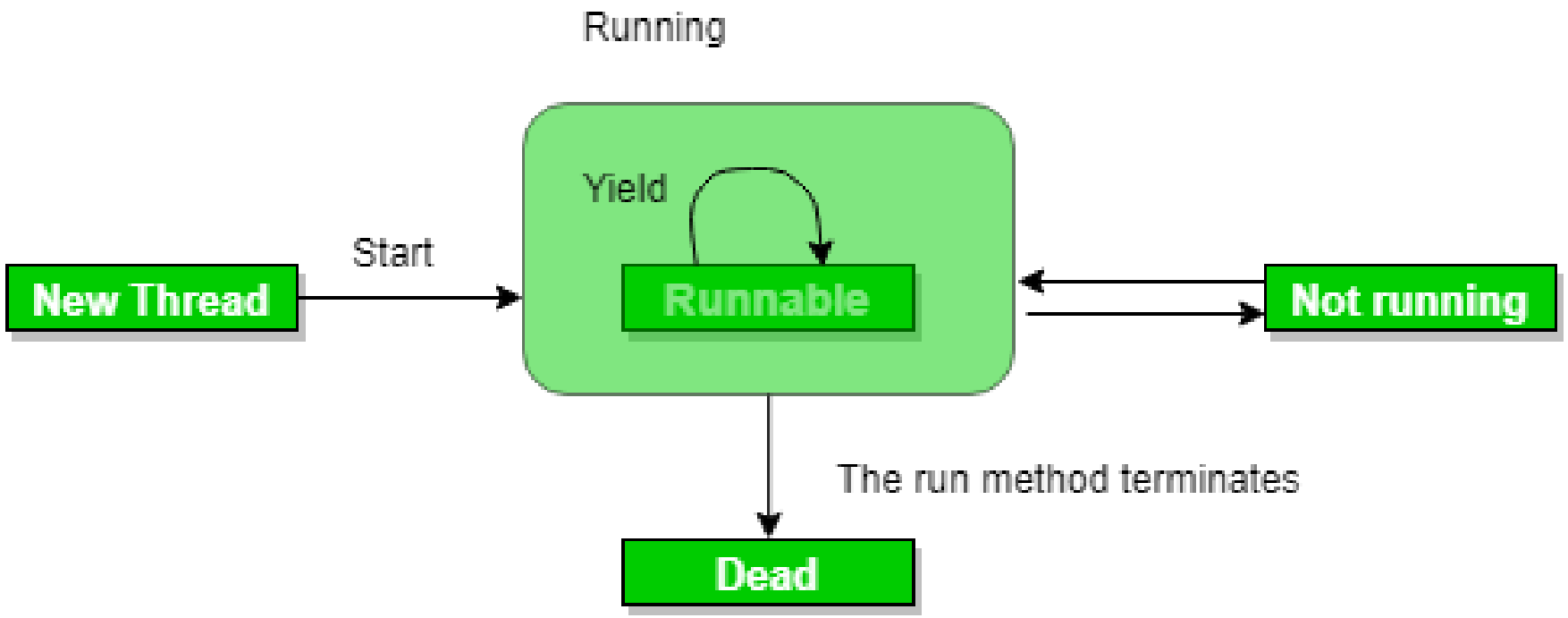
```
import java.util.Set;
public class ActiveThreadDemo
{
   public static void main(String[] args)
   {
      System.out.println(Thread.activeCount());
      Aa a=new Aa();
      a.setName("Ram");
      a.start();
      System.out.println("After starting A thread"+Thread.activeCount());
      B b=new B();
      b.setName("Bheem");
      b.start();
      System.out.println("After starting B thread"+Thread.activeCount());

      Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
      for ( Thread t : threadSet){
         if ( t.getThreadGroup() == Thread.currentThread().getThreadGroup()){
            System.out.println("Thread :"+t+":"+"state:"+t.getState());
         }
      }
   }
}
```

```
class Aa extends Thread
{
public void run()
   {
try {
         Thread.sleep(20000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
class B extends Thread
{
public void run() {
            try {
         Thread.sleep(20000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

yield():

- Suppose there are three threads t1, t2, and t3.
- Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state.
- The completion time for thread t1 is 5 hours and the completion time for t2 is 5 minutes.
- Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job.
- In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent the execution of a thread in between if something important is pending.


Running
Yield
Start
New Thread
Runnable
Not running
The run method terminates
Dead

- Whenever a thread calls java.lang.Thread.yield method gives hint to the thread scheduler that it is ready to pause its execution.

- The thread scheduler is free to ignore this hint.

- If any thread executes the yield method, the thread scheduler checks if there is any thread with the same or high priority as this thread.

- If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – the current thread will keep executing.

- Once a thread has executed the yield method and there are many threads with the same priority is waiting for the processor, then we can’t specify which thread will get the execution chance first.

- The thread which executes the yield method will enter in the Runnable state from Running state.

- Once a thread pauses its execution, we can’t specify when it will get a chance again it depends on the thread scheduler.

- The underlying platform must provide support for preemptive scheduling if we are using the yield method.

join():

If a Thread wants to wait until completing some other thread then we should go for join() method.

- ❖public final void join() **throws** InterruptedExcetion
- ❖public final void join(long ms) **throws** InterruptedException
- ❖public final void join(long ms, int ns) **throws** InterruptedException

```java
public class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t1=new MyThread();
        t1.setName("Sita");
        t1.start();
        t1.join();
        MyThread t2=new MyThread();
        t2.setName("Rama");
        t2.start();
        t2.join();
    }
}
class MyThread extends Thread

{
    @Override
    public void run() {
        for (int i=0;i<10;i++)
        {
            System.out.println(Thread.currentThread().getName()+" is executing..");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Output:

Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..

isAlive() :

used to check whether the thread is live or not.

- ❖public Boolean isAlive();

currentThread():

- ➢This method is used to represent current thread class object.
  - ❖public static thread currentThread();

isAlive() :

```
public class ThreadDemo
{
    public static void main(String[] args) throws InterruptedException {
        MyThread2 t=new MyThread2();
        System.out.println("t1 Thread state:"+t.isAlive());
        t.start();
        System.out.println("t1 Thread state:"+t.isAlive());
        Thread.sleep(20000);
        System.out.println("t1 Thread state:"+t.isAlive());
    }
}
class MyThread2 extends Thread
{
    @Override
    public void run() {
        for (int i=0;i<10;i++)
        {
            System.out.println("Hello..");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

interrupted():

- A thread can interrupt another sleeping or waiting thread.
- For this Thread class defines interrupt() method.
  - public void interrupt();

Note:

- The interrupt() is working good whenever our thread enters into waiting state or sleeping state.
- The interrupted call will be wasted if our thread doesn’t enters into the waiting/sleeping state.

Synchronizing threads

- **Synchronized** modifier is the modifier applicable only for methods and blocks but not for classes and variables.

- If a method or a block declared as synchronized then at a time only one Thread is allowed to operate on the given object.

- The main advantage of synchronized modifier is we can resolve data inconsistency problems.

- But the main disadvantage of synchronized modifier is it increases the waiting time of the Thread and effects performance of the system.

- Hence if there is no specific requirement it is never recommended to use.

- The main purpose of this modifier is to reduce the data inconsistence problems.

Example:

```
public class SyncDemo
{
    public static void main(String[]
args) {
        Whish w=new Whish();
        Mt t1=new Mt("Purushotham",w);
        Mt t2=new Mt("Naresh",w);
        t1.start();
        t2.start();
    }
}
class Whish
{
    public synchronized void
whish(String name) throws
InterruptedException {
        for (int i=0;i<10;i++)
        {
            System.out.println("Good
mrng.."+name);
            Thread.sleep(2000);
        }
    }
}
```

```
class Mt extends Thread
{
    Whish w;
    String name;
    Mt(String name,Whish w)
    {
        this.name=name;
        this.w=w;
    }

    @Override
    public void run() {
        super.run();
        try {
            w.whish(name);
        } catch (InterruptedException e)
{
            e.printStackTrace();
        }
    }
}
```

Inter Thread Communication

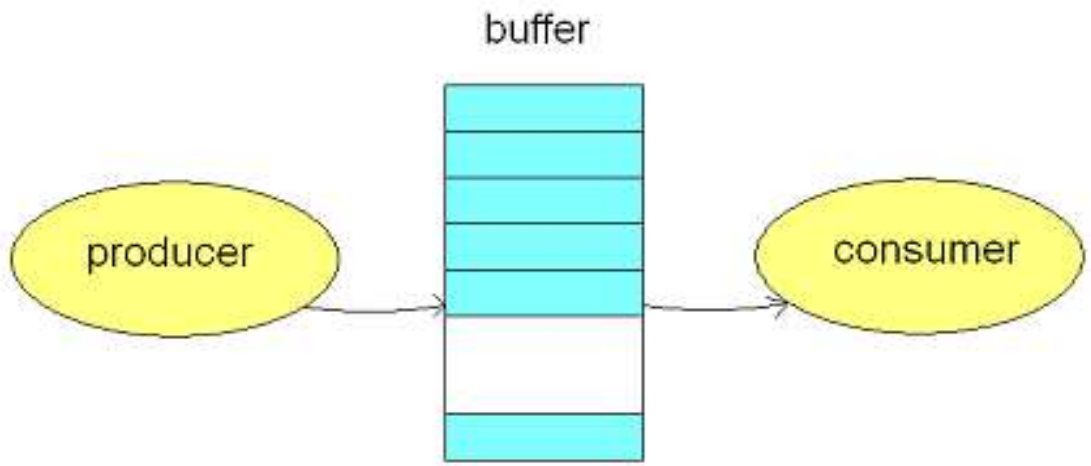
- Two threads can communicate with each other by using
  - wait(),
  - notify(),
  - notifyAll()

- These methods are available in **Object** class but not in Thread class. Because threads are calling these methods on any object.

- We should call these methods only from synchronized area other wise we will get runtime exception saying **IllegalMonitorStateException.**

Producer Consumer Problem

- The producer-consumer problem is a classic example of a multi-process synchronization problem.
- There are two processes, a producer and a consumer, that share a common buffer with a limited size.
- The producer “produces” data and stores it in the buffer, and the consumer “consumes” the data, removing it from the buffer.
- Having two processes that run in parallel, we need to make sure that the producer will not put new data in the buffer when the buffer is full.
- Consumer won’t try to remove data from the buffer if the buffer is empty.


buffer
producer
consumer

Producer Consumer Program:

```
import java.util.LinkedList;
public class ProducerConsumer
{
    public static void main(String[] args) throws InterruptedException {
        PC p=new PC();
        Thread t1=new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    p.produce();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

            }
        });
        Thread t2=new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    p.consumer();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();

}
```

```
class PC
{
    LinkedList<Integer> linkedList=new LinkedList<Integer>();
    int capacity=1;

    public synchronized   void produce() throws InterruptedException
    {
        int value=0;

       while (true)
       {
               while (linkedList.size()==capacity)
               {
                   wait();
               }
               System.out.println("Producer produced item-"+value);
               linkedList.add(value++);
               notify();
               Thread.sleep(3000);
       }
    }
    public synchronized void consumer() throws InterruptedException
    {
        while (true)
        {
                while (linkedList.size()==0) {
                    wait();
                }
                int val=linkedList.removeFirst();
                System.out.println("Consumer consumed item-"+val);
                notify();
                Thread.sleep(3000);
        }

}}
```

Daemon Threads

- Daemon thread in Java is a low-priority thread that performs background operations such as garbage collection, finalizer, Action Listeners, Signal dispatches, etc.

- Daemon thread in Java is also a service provider thread that helps the user thread.

- Its life is at the mercy of user threads i.e, when all user threads expire, JVM immediately terminates this thread.

- Usually daemon thread are running with low priority but based on our requirement we can increase their priority also.

- We can check whether the given thread is daemon or not by using the following thread class thread.
  - **public boolean isDaemon();**

- we can change the daemon nature of a thread by using setDaemon() method of thread class.
  - **public void setDaemon(Boolean b);**

Note:

IllegalThreadStateException:

- If you call the setDaemon() method after the thread has started, it will throw an exception.

Daemon thread example

```
class MyThread extends Thread
{

}
public class Daemon
 {
    public static void main(String[] args)
    {
        MyThread t1=new MyThread();
        System.out.println(t1.isDaemon());
        t1.setDaemon(true);
        System.out.println(t1.isDaemon());
    }
}
```

Output:

fasle

true

JDBC
(Java Database Connectivity)

Storage areas or options

Storage areas or options

- As the part of our Applications, we required to store our data like customers information, Billing Information, Calls Information etc..
- To store this Data, we required Storage Areas. There are 2 types of Storage Areas.
  - Temporary Storage Areas
  - Permanent Storage Areas

Temporary Storage Areas:

These are the Memory Areas where Data will be stored temporarily.

Ex: All JVM Memory Areas (like Heap Area, Method Area, Stack Area etc).
Once JVM shutdown all these Memory Areas will be cleared automatically.

Permanent Storage Areas:

- Also known as Persistent Storage Areas.
- Here we can store Data permanently.
  - Ex: File Systems, Databases, Data warehouses, Big Data Technologies

File Systems:

- File Systems can be stored unstructured data.
- File Systems can be provided by Local operating System.
- File Systems are best suitable to store very less Amount of Information.

Limitations:

- We cannot store huge amount of data.
- There is no Query Language support and hence operations will become very complex.
- There is no security for the data.
- There is no mechanism to prevent duplicate data. Hence there may be a chance of data inconsistency problems.
- To overcome the above problems of File Systems, we should recommended to use Databases.

Databases:

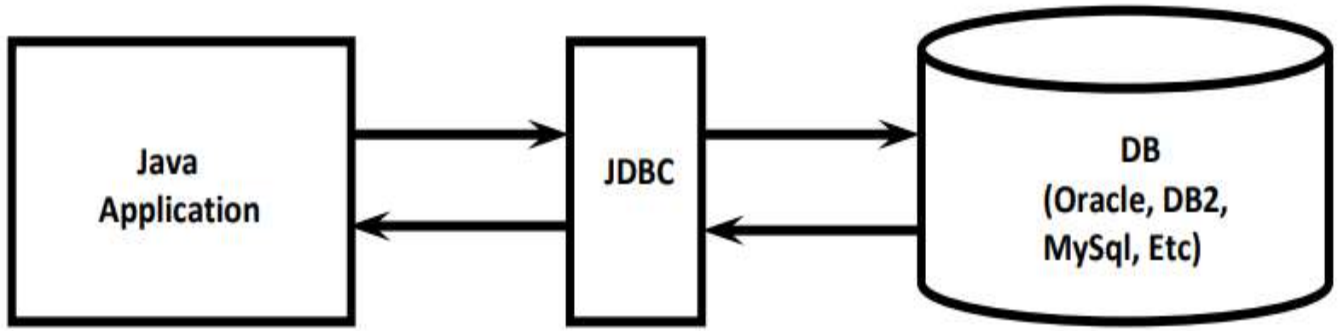
- We can store huge amount of data in the Databases.
- Query language support is available for every Database and hence we can perform Database operations very easily.
- To access data present in the Database, compulsory username and pwd must be required. Hence data is secured.
- Inside Database data will be stored in the form of Tables. While developing database table schemas, Database admin follow various normalization techniques and can implement various constraints like unique key constrains, primary key constraints etc which prevent data duplication.
- Hence there is no chance of Data Inconsistency Problems.

Limitations of Databases:

- Database cannot hold very huge amount of information like terabytes of Data.
- Database can provide support only for structured data (Tabular Data OR Relational Data) and cannot provide support for semi structured data (like XML Files) and Unstructured Data (like Video Files, Audio Files, Images etc)
- To overcome these problems we should go for more advanced storage areas like Big Data Technologies, Data warehouses etc..

JDBC

- JDBC is a Technology, which can be used to communicate with Database from Java Application.



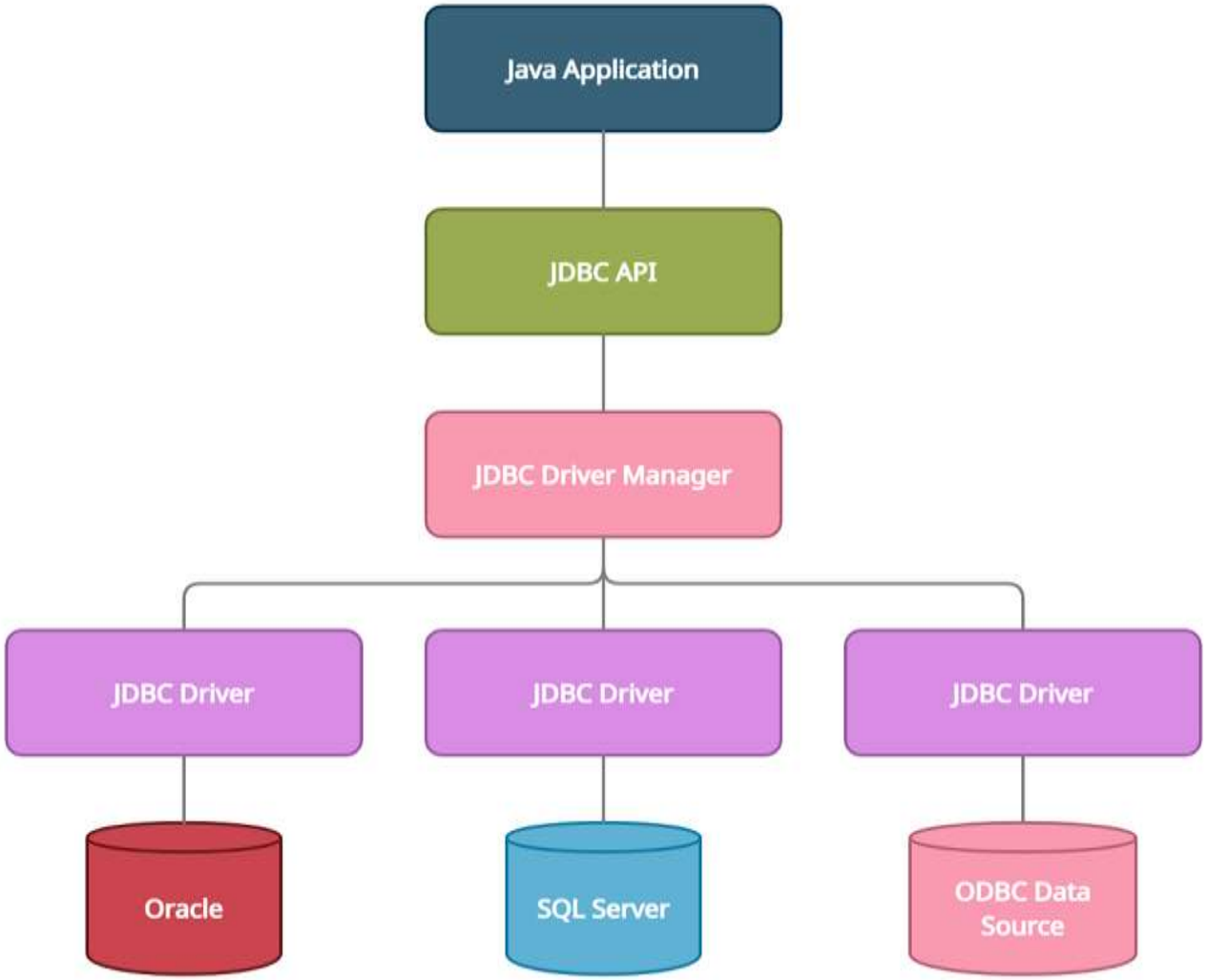
- JDBC is the Part of Java Standard Edition.
- JDBC is a Specification defined by Java Vendor (Sun Micro Systems) and implemented by Database Vendors.
- Database Vendor provided Implementation is called "Driver Software".

JDBC Features:

- JDBC API is standard API. We can communicate with any Database without rewriting our Application i.e. it is Database independent API.
- JDBC Drivers are developed in Java and hence JDBC Concept is applicable for any Platform. i.e. JDBC is platform independent technology.
- By using JDBC API, we can perform basic CRUD operations very easily.
- We can also perform complex operations (like Inner Joins, Outer Joins, calling Stored Procedures etc) very easily by using JDBC API.
- JDBC API supported by large number of vendors and they developed multiple Products based on JDBC API.

JDBC Architecture

JDBC Architecture



JDBC Architecture

- **Application:** It is a java applet or a servlet that communicates with a data source.
- **The JDBC API:** The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:
- **DriverManager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.
- **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

DriverManager:

- It is the Key Component in JDBC Architecture.
- DriverManager is a Java Class present in **java.sql p**ackage.
- It is responsible to manage all Database drivers available in our system.
- DriverManager is responsible to register and unregister Database Drivers.
  - DriverManager.registerDriver(Driver);
  - DriverManager.unregisterDriver(Driver);
- DriverManager is responsible to establish connection to the Database with the help of Driver Software.
  - Connection con = DriverManager.getConnection (jdbcurl, username, pwd);

JDBC API

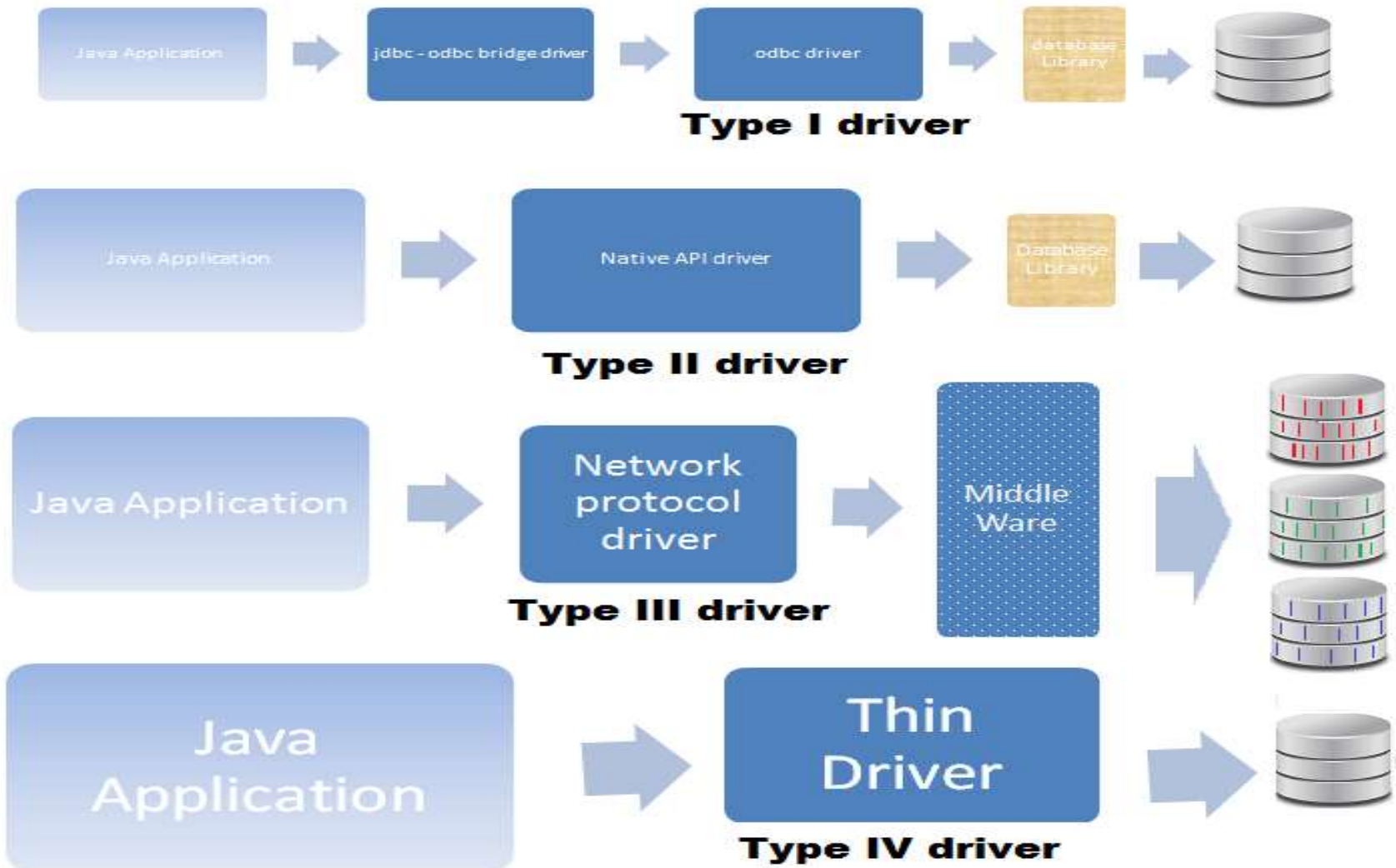
- JDBC API provides several Classes and Interfaces.
- Programmer can use these Classes and Interfaces to communicate with the Database.
- Driver Software Vendor can use JDBC API while developing Driver Software.
- JDBC API defines 2 Packages
  - java.sql Package:
  - javax.sql Package:

java.sql Package:

javax.sql Package:

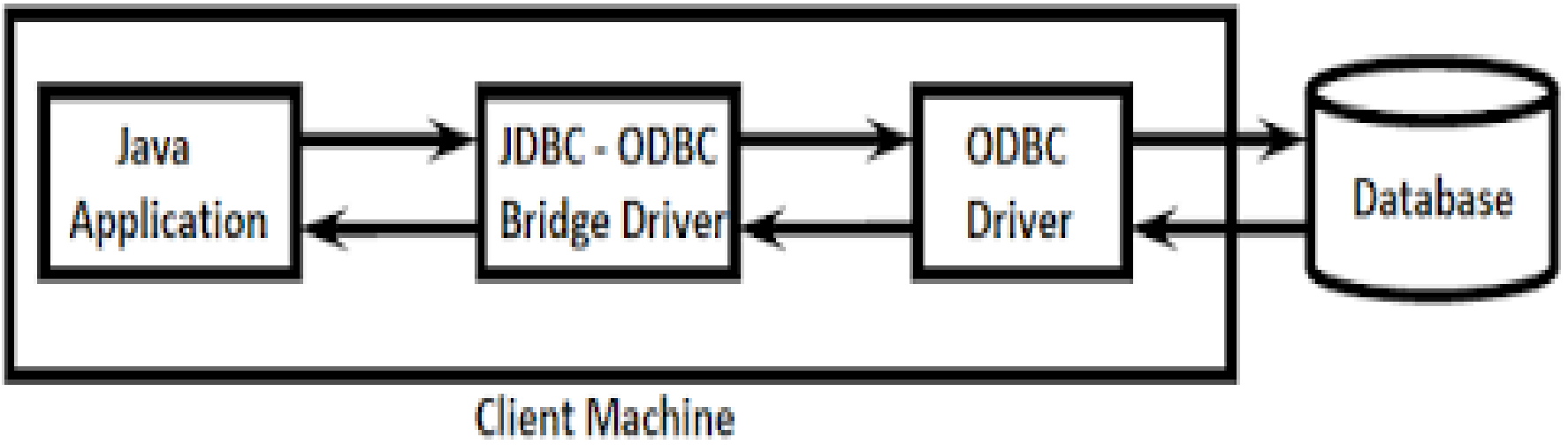
Types of drivers

- While communicating with Database, we have to convert Java calls into Database specific calls and Database specific calls into Java calls. For this driver software is required.
- There are many drivers are available. But based on functionality all drivers are divided into 4 Types.
  - Type-1 Driver (JDBC-ODBC Bridge Driver OR Bridge Driver)
  - Type-2 Driver (Native API-Partly Java Driver OR Native Driver)
  - Type-3 Driver (All Java Net Protocol Driver OR Network Protocol Driver OR Middleware Driver)
  - Type-4 Driver (All Java Native Protocol Driver OR Pure Java Driver OR Thin Driver)



Types of JDBC drivers

Type-1 Driver



Type-1 Driver:

- Type-1 driver provided by Sun Micro Systems as the part of JDK. But this Support is available until 1.7 version only.
- Internally this driver will take support of ODBC Driver to communicate with Database.
- Type-1 Driver converts JDBC Calls (Java Calls) into ODBC Calls and ODBC Driver converts ODBC calls into Database specific Calls.
- Hence Type-1 Driver acts as Bridge between JDBC and ODBC.

Advantages :

- It is very easy to use and maintain.
- We are not required to install any separate Software because it is available as the Part of JDK.
- Type-1 Driver won't communicates directly with the Database. Hence it is Database Independent Driver. Because of this migrating from one Database to another Database will become very easy.

Limitations:

- It is the slowest Driver among all JDBC Drivers (Snail Driver), because first it will convert JDBC Calls into ODBC Calls and ODBC Driver converts ODBC Calls into Database specific Calls.

- This Driver internally depends on ODBC Driver, which will work only on Windows Machines. Hence Type-1 Driver is Platform Dependent Driver.

- No Support from JDK 1.8 Version onwards.

Type-2 Driver

- It is also known as Native API -partly Java Driver OR Native Driver.
- Type-2 Driver is exactly same as Type-1 Driver except that ODBC Driver is replaced with vendor specific Native Libraries.
- Type-2 Driver internally uses vendor specific native libraries to communicate with Database.
- Native libraries means the set of Functions written in Non-Java (Mostly C OR C++).
- We have to install Vendor provided Native Libraries on the Client Machine.
- Type-2 Driver converts JDBC Calls into Vendor specific Native Library Calls, which can be understandable directly by Database Engine.

Advantages:

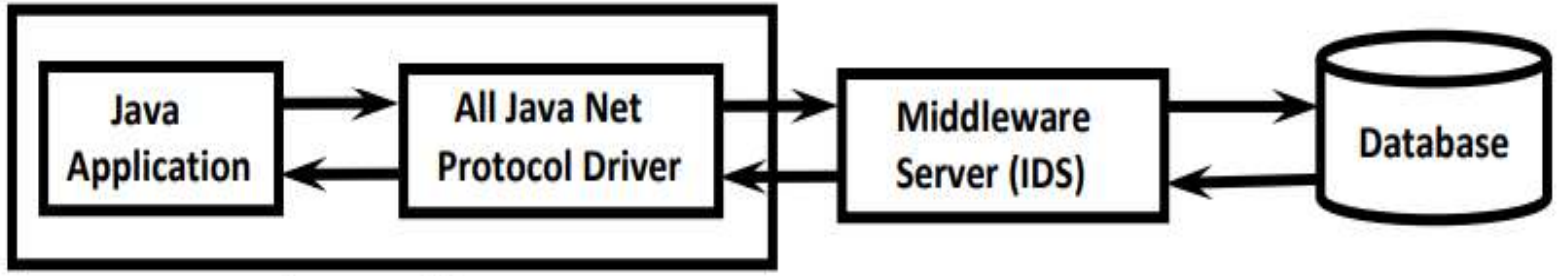
- When compared with Type-1 Driver Performance is High, because it required only one Level Conversion from JDBC to Native Library Calls.
- No need of arranging ODBC Drivers.
- When compared with Type-1 Driver, Portability is more because Type-1 Driver is applicable only for Windows Machines.

Limitations:

- Internally this Driver using Database specific Native Libraries and hence it is Database Dependent Driver. Because of this migrating from one Database to another Database will become Difficult.
- This Driver is Platform Dependent Driver.
- On the Client Machine compulsory we should install Database specific Native Libraries.
- There is no Guarantee for every Database Vendor will provide This Driver.
- (Oracle is providing Type-2 Driver but MySql won’t providing this Driver)

Type-3 Driver:

- Also known as All Java Net Protocol Driver OR Network Protocol Driver OR Middleware Driver.
- Type-3 Driver converts JDBC Calls into Middleware Server specific Calls. Middleware Server can convert Middleware Server specific Calls into Database specific Calls.
- Internally Middleware Server may use Type-1, 2 OR 4 Drivers to communicates with Database.



Advantages:

- This Driver won't communicate with Database directly and hence it is Database Independent Driver.
- This Driver is Platform Independent Driver.
- No need of ODBC Driver OR Vendor specific Native Libraries

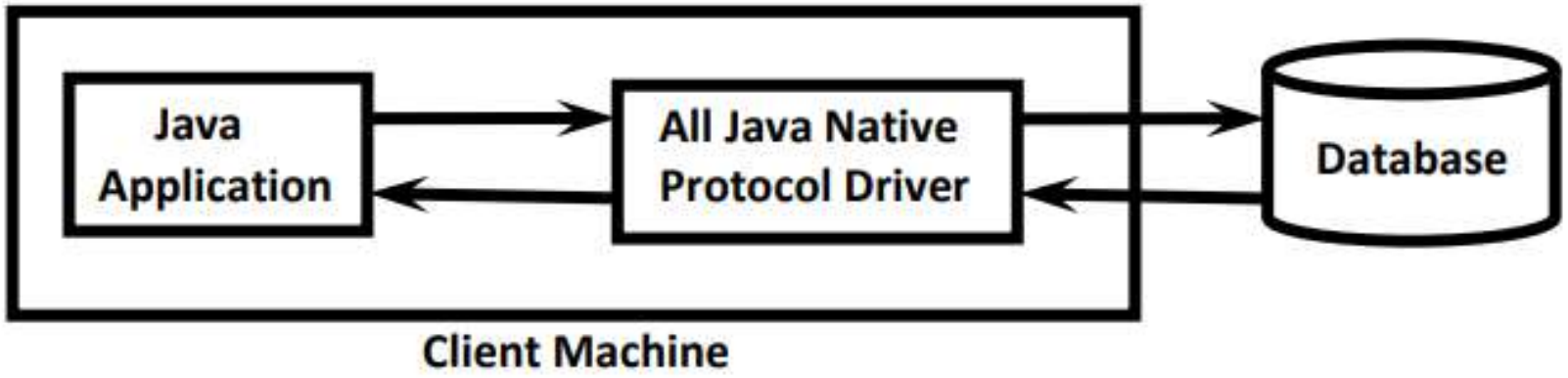
Limitations:

- Because of having Middleware Server in the Middle, there may be a chance of Performance Problems.
- We need to purchase Middleware Server and hence the cost of this Driver is more when compared with remaining Drivers.
  - Ex: IDS Driver (Internet Database Access Server)

Note: The only Driver which is both Platform Independent and Database Independent is Type-3 Driver. Hence it is recommended to use.

Type-4 Driver:

- Also known as Pure Java Driver OR Thin Driver.



- This Driver is developed to communicate with the Database directly without taking Support of ODBC Driver OR Vendor Specific Native Libraries OR Middleware Server.
- This Driver uses Database specific Native Protocols to communicate with the Database.
- This Driver converts JDBC Calls directly into Database specific Calls.
- This Driver developed only in Java and hence it is also known as Pure Java Driver.
- Because of this, Type-4 Driver is Platform Independent Driver. This Driver won't require any Native Libraries at Client side and hence it is light weighted. Because of this it is treated as Thin Driver.

Advantages

- It won't require any Native Libraries, ODBC Driver OR Middleware Server
- It is Platform Independent Driver
- It uses Database Vendor specific Native Protocol and hence Security is more.

Limitations:

The only Limitation of this Driver is, it is Database Dependent Driver because it is communicating with the Database directly.

Ex: Thin Driver for Oracle **Connector/J** Driver for **MySQL**

Collections

Limitations of array:

- Array is indexed collection o fixed number of homogeneous data elements
- Arrays can hold homogeneous data only
- Once we created an array no chance of increasing o decreasing size of array

Ex:

```
Student[ ] s=new Student[100];
   S[0]=new Student();
   S[1]=new Student();
   S[2]=new Customer();
```

compilation error

- To overcome the above limitations of array the sun peoples are introduced collections concept

- To overcome the limitations in Array we should go for collections concept.

- Collections are growable in nature that is based on our requirement we can increase (or) decrease the size hence memory point of view collections concept is recommended to use.

- Collections can hold both homogeneous and heterogeneous objects.

- Every collection class is implemented based on some standard data structure

- Hence for every requirement ready-made method support is available being a programmer we can use these methods directly without writing the functionality on our own

Collections:

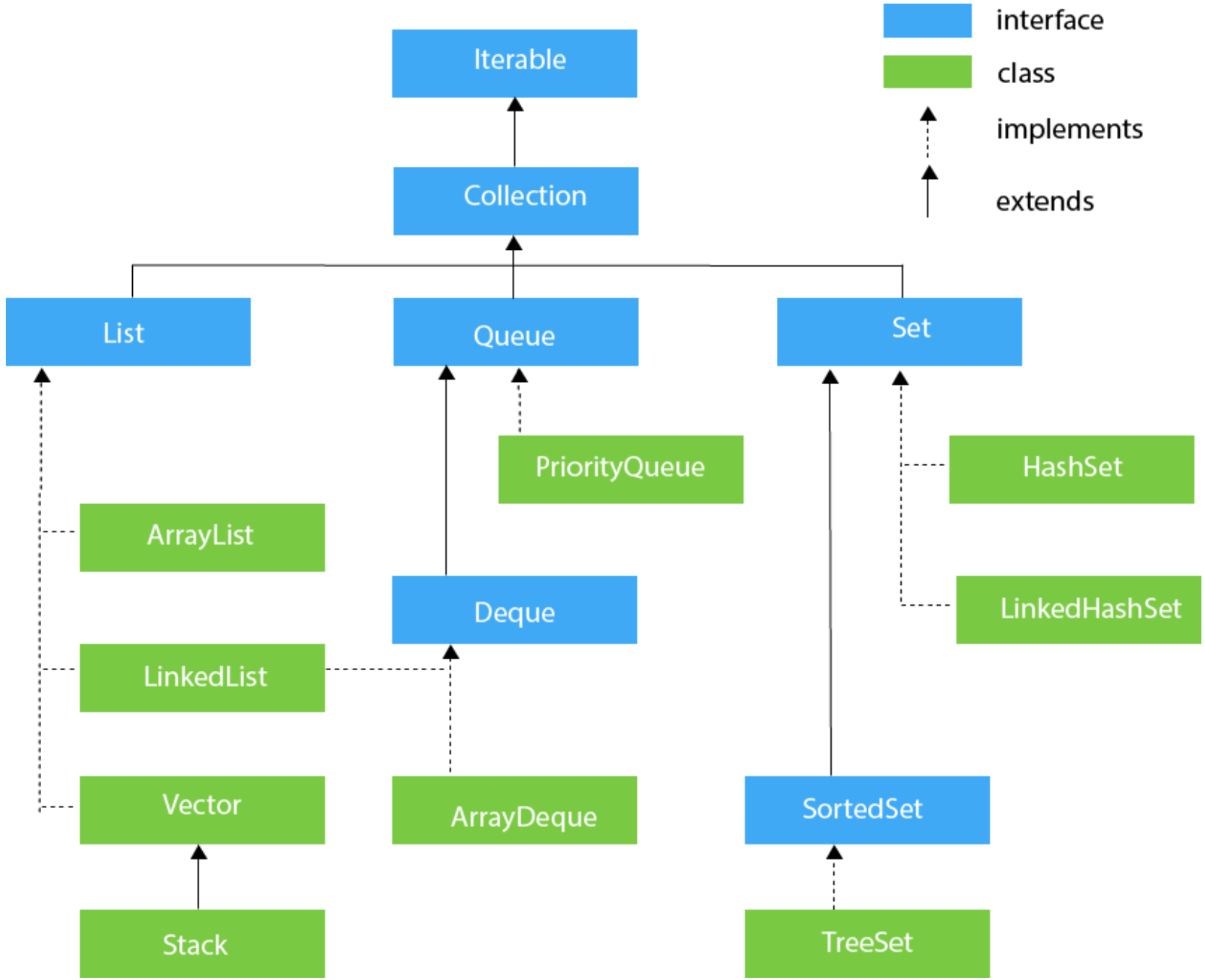
- collection can hold both homogeneous data and heterogeneous data
- collections are growable in nature
- Memory wise collections are good. Recommended to use.
- Performance wise collections are not recommended to use .

Arrays	Collections
1) Arrays are fixed in size.	1) Collections are growable in nature.
2) Memory point of view arrays are not recommended to use.	2) Memory point of view collections are highly recommended to use.
3) Performance point of view arrays are recommended to use.	3) Performance point of view collections are not recommended to use.
4) Arrays can hold only homogeneous data type elements.	4) Collections can hold both homogeneous and heterogeneous elements.
5) There is no underlying data structure for arrays and hence there is no readymade method support.	5) Every collection class is implemented based on some standard data structure and hence readymade method support is available.
6) Arrays can hold both primitives and object types.	6) Collections can hold only objects but not primitives.

Introduction to Collections Framework

- “The Collections Framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a collection.” -java.sun.com
- The standard data structure in Java can be implemented in Java using some library classes and methods. These classes are present in **java.util** package.
- The collection framework is comprised of collection classes and collection interfaces.
- Collection is a group of objects which are designed to perform certain task. These tasks are associated with data structures.

- The collection classes are the group of classes used to implement the collection interfaces. Various collection classes are…
  - LinkedList
  - ArrayList
  - AbstractSet
  - EnumSet
  - HashSet
  - PriorityQueue
  - TreeSet
  - Vector
  - HashTable ..etc


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Collection - interface

- If we want to represent a group of "individual objects" as a single entity then we should go for collection.

- In general we can consider collection as root interface of entire collection framework.

- Collection interface defines the most common methods which can be applicable for any collection object.

- There is no concrete class which implements Collection interface directly.

Collection Interface Methods

Method	Description
add(Object)	This method is used to add an object to the collection.
addAll(Collection c)	This method adds all the elements in the given collection to this collection.
clear()	This method removes all of the elements from this collection.
contains(Object o)	This method returns true if the collection contains the specified element.
containsAll(Collection c)	This method returns true if the collection contains all of the elements in the given collection.
equals(Object o)	This method compares the specified object with this collection for equality.
hashCode()	This method is used to return the hash code value for this collection.
isEmpty()	This method returns true if this collection contains no elements.
iterator()	This method returns an iterator over the elements in this collection.
max()	This method is used to return the maximum value present in the collection.
parallelStream()	This method returns a parallel Stream with this collection as its source.
remove(Object o)	This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.
removeAll(Collection c)	This method is used to remove all the objects mentioned in the given collection from the collection.
removeIf(Predicate filter)	This method is used to remove all the elements of this collection that satisfy the given predicate.
retainAll(Collection c)	This method is used to retain only the elements in this collection that are contained in the specified collection.
size()	This method is used to return the number of elements in the collection.
spliterator()	This method is used to create a Spliterator over the elements in this collection.
stream()	This method is used to return a sequential Stream with this collection as its source.
toArray()	This method is used to return an array containing all of the elements in this collection.

List -interface

- It is the child interface of Collection.
- If we want to represent a group of individual objects as a single entity where "duplicates are allow and insertion order must be preserved" then we should go for List interface.
- We can differentiate duplicate objects and we can maintain insertion order by means of index hence "index play very important role in List“
- All list interface methods build based on index.

Methods in List interface

- boolean add(int index,Object o);
- boolean addAll(int index,Collectio c);
- Object get(int index);
- Object remove(int index);
- Object set(int index,Object new);//to replace
- Int indexOf(Object o);
- Returns index of first occurrence of "o".
- Int lastIndexOf(Object o);
- ListIterator listIterator();

ArrayList

ArrayList

- Introduced in 1.2 version.
- ArrayList supports dynamic array that can be grow as needed.it can dynamically increase and decrease the size.
- Duplicate objects are allowed.
- Null insertion is possible.
- Heterogeneous objects are allowed.
- The under laying data structure is growable array.
- Insertion order is preserved.

Constructors:

1) ArrayList a=new ArrayList();

Creates an empty ArrayList object with default initial capacity "10" if ArrayList reaches its max capacity then a new ArrayList object will be created with

New capacity=(current capacity*3/2)+1

- 2)ArrayList a=new ArrayList(int initialcapacity);

Creates an empty ArrayList object with the specified initial capacity

Vector

Vector

- The underlying data structure is resizable array (or) growable array.
- Duplicate objects are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- Null insertion is possible.
- Implements Serializable, Cloneable and RandomAccess interfaces.
- Every method present in Vector is synchronized and hence Vector is Thread safe.

Vector specific methods:

To add objects:

- add(Object o);-----Collection
- add(int index,Object o);-----List
- addElement(Object o);-----Vector

To remove elements:

- ❖remove(Object o);--------Collection
- ❖remove(int index);--------------List
- ❖removeElement(Object o);----Vector
- ❖removeElementAt(int index);-----Vector
- ❖removeAllElements();-----Vector
- ❖clear();-------Collection

To get objects:

- Object get(int index);----------List
- Object elementAt(int index);-----Vector
- Object firstElement();----------Vector
- Object lastElement();----------Vector

Constructors:

1) Vector v=new Vector();

- Creates an empty Vector object with default initial capacity 10.
- Once Vector reaches its maximum capacity then a new Vector object will be created with double capacity.
- That is "newcapacity=currentcapacity*2"

- 2) Vector v=new Vector(int initialcapacity);
- 3) Vector v=new Vector(int initialcapacity, int incrementalcapacity);
- 4) Vector v=new Vector(Collection c);

Stack

Stack

- It is the child class of Vector.
- Whenever last in first out(LIFO) order required then we should go for Stack.

Constructor:

- It contains only one constructor.

```
Stack s= new Stack();
```

Methods:

- **Object push(Object o)**

  To insert an object into the stack.

- **Object pop()**

  To remove and return top of the stack.

- **Object peek()**

  To return top of the stack without removal.

- **boolean empty()**

  Returns true if Stack is empty.

```java
import java.util.Stack;

public class StackDemo
{
    public static void main(String[] args) {
        Stack s=new Stack();
        System.out.println("Elements in stack :"+s);
        s.push("A");
        s.push("B");
        s.push("C");
        s.push("D");
        s.push("E");
        System.out.println("Elements in stack :"+s);
        s.pop();
        System.out.println("Elements in stack :"+s);
        System.out.println("Top element in stack :"+s.peek());
        System.out.println("Search A element in stack :"+s.search("A"));
        System.out.println("Search F element in stack :"+s.search("F"));
        System.out.println("Search A element in stack :"+s);

    }
}
```

Output:

```
Elements in stack :[]
Elements in stack :[A, B, C, D, E]
Elements in stack :[A, B, C, D]
Top element in stack :D
Search A element in stack :4
Search F element in stack :-1
Search A element in stack :[A, B, C, D]
```

```
import java.util.Scanner;
import java.util.Stack;
public class Ex5
{
    public static void main(String[] args) {
        Stack<Integer> st=new Stack<Integer>();
        int choice=0;
        int position;
        Scanner scr=new Scanner(System.in);
        while(true)    {
            System.out.println("Stack Operations");
            System.out.println("1.Push an element");
            System.out.println("2.Display stack");
            System.out.println("3.Pop an element");
            System.out.println("4.Search an element");
            System.out.println("Enter your choice");
            choice=scr.nextInt();
            switch(choice)  {
                case 1:
                    System.out.println("Enter an element");
                    Integer i=scr.nextInt();
                    st.push(i);
                   break;
                case 2:
                    System.out.println("Elements in stack :"+st);
                    break;
                case 3:
                    System.out.println("Top element popped..");
                    Integer obj = st.pop();
                    System.out.println("Popped element= "+obj);
                    break;
                case 4:
                    System.out.println("Which an element ? ");
                    Integer ele=scr.nextInt();
                    position = st.search(ele);
                    if(position==-1)
                        System.out.println("Element not found");
                    else
                        System.out.println("Position of the element is:= "+position);
                    break;
                default:
                    return;    }   }    }
}
```

Output:

```
Stack Operations
1.Push an element
2.Display stack
3.Pop an element
4.Search an element
Enter your choice
1
Enter an element
10
Stack Operations
1.Push an element
2.Display stack
3.Pop an element
4.Search an element
Enter your choice
1
Enter an element
20
Stack Operations
1.Push an element
2.Display stack
3.Pop an element
4.Search an element
Enter your choice
2
Elements in stack :[10, 20]
Stack Operations
1.Push an element
2.Display stack
3.Pop an element
4.Search an element
Enter your choice
```

LinkedList:

- The underlying data structure is double LinkedList.
- If our frequent operation is insertion (or) deletion in the middle then LinkedList is the best choice.
- If our frequent operation is retrieval operation then LinkedList is worst choice.
- Duplicate objects are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- Null insertion is possible.

Methods in LinkedList

- void addFirst(Object o);
- void addLast(Object o);
- Object getFirst();
- Object getLast();
- Object removeFirst();
- Object removeLast();

Constructors:

- LinkedList l=new LinkedList();

  Creates an empty LinkedList object.

- LinkedList l=new LinkedList(Collection c);

  To create an equivalent LinkedList object for the given collection.

Cursors

The 3 cursors of java:

If we want to get objects one by one from the collection then we should go for cursor.

There are 3 types of cursors available in java. They are:

1. Enumeration
2. Iterator
3. ListIterator

Enumeration:

- We can use Enumeration to get objects one by one from the legacy collection objects.
- We can create Enumeration object by using elements() method.
  - public Enumeration elements();
  - Enumeration e=v.elements();

Enumeration interface defines the following two methods

- public boolean hasMoreElements();
- public Object nextElement();

Example:

```
import java.util.*;
class  EnumerationDemo
{
    public static void main(String[] args)
    {
        Vector v=new Vector();
        for(int i=0;i<=10;i++)
        {
            v.addElement(i);
        }
        System.out.println(v);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        Enumeration e=v.elements();
        while(e.hasMoreElements())
        {
            Integer i=(Integer)e.nextElement();
            if(i%2==0)
                System.out.println(i);//0 2 4 6 8 10
        }
    }
}
```

Limitations of Enumeration:

- We can apply Enumeration concept only for legacy classes and it is not a universal cursor.
- By using Enumeration we can get only read access and we can't perform remove operations.
- To overcome these limitations they introduced Iterator concept in 1.2v.

Iterator:

- We can use Iterator to get objects one by one from any collection object.
- We can apply Iterator concept for any collection object and it is a universal cursor.
- While iterating the objects by Iterator we can perform both read and remove operations.
- We can get Iterator object by using iterator() method of Collection interface.

  public Iterator iterator();

  Ex: Iterator itr=c.iterator();

Iterator interface defines the following 3 methods.

- public boolean hasNext();
- public object next();
- public void remove();

Ex:

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorEx
{
    public static void main(String[] args) {
        ArrayList al=new ArrayList();
        for (int i=0;i<100;i++)
        {
            al.add(i);
        }
        System.out.println("ArrayList Elements:"+al);
        Iterator i=al.iterator();

        while (i.hasNext())
        {
            int n=(int)i.next();
            if(n%2!=0)
                i.remove();
        }
        System.out.println("Even ArrayList
Elements:"+al);

    }
}
```

Output:

ArrayList Elements:[0, 1, 2, 3, 4, 5, 6, 7, 8, .. 99]

Even ArrayList Elements:[0, 2, 4, 6, 8, 10, ..98]

Limitations of Iterator:

- Both enumeration and Iterator are single direction cursors only. That is we can always move only forward direction and we can't move to the backward direction.
- While iterating by Iterator we can perform only read and remove operations and we can't perform replacement and addition of new objects.
- To overcome these limitations sun people introduced listIterator concept

Enumeration	Iterator
Introduced in Java 1.0	Introduced in Java 1.2
Legacy Interface	Not Legacy Interface
It is used to iterate only Legacy Collection classes.	We can use it for any Collection class.
It supports only READ operation.	It supports both READ and DELETE operations.
It’s not Universal Cursor.	It is a Universal Cursor.
Lengthy Method names.	Simple and easy-to-use method names.

ListIterator:

- ListIterator is the child interface of Iterator.

- By using listIterator we can move either to the forward direction (or) to the backward direction that is it is a bi-directional cursor.

- While iterating by listIterator we can perform replacement and addition of new objects in addition to read and remove operations

By using listIterator method we can create listIterator object.

public ListIterator listIterator();

ListIterator itr=l.listIterator();

ListIterator interface defines the following 9 methods.

- public boolean hasNext();
- public Object next(); forward
- public int nextIndex();
- public boolean hasPrevious();
- public Object previous(); backward
- public int previousIndex();
- public void remove();
- public void set(Object new);
- public void add(Object new);

Ex:

```
import java.util.LinkedList;
import java.util.ListIterator;
public class MyLinkedList
{
    public static void main(String[] args)
    {
        LinkedList<String> ll=new LinkedList<String>();
        ll.add("A");
        ll.add("B");
        ll.add("C");
        ll.add("D");
        ll.add("E");
        ListIterator li=ll.listIterator();
        System.out.println("LinkedList elements in forward direction..");
        while (li.hasNext())
        {
            System.out.println(li.next());
        }
        System.out.println("LinkedList elements in backward direction..");
        while (li.hasPrevious())
        {
            System.out.println(li.previous());
        }
    }
}
```

Output:

```
LinkedList elements in forward direction..
A
B
C
D
E
LinkedList elements in backward direction..
E
D
C
B
A
```

Comparison of Enumeration Iterator and ListIterator ?

Property	Enumeration	Iterator	ListIterator
1) Is it legacy ?	Yes	no	no
2) It is applicable for ?	Only legacy classes.	Applicable for any collection object.	Applicable for only list objects.
3) Moment?	Single direction cursor(forward)	Single direction cursor(forward)	Bi-directional.
4) How to get it?	By using elements() method.	By using iterator()method.	By using listIterator() method.
5) Accessibility?	Only read.	Both read and remove.	Read/remove/replace/add.
6) Methods	hasMoreElement() nextElement()	hasNext() next() remove()	9 methods.

Generic classes

Generic Class

- JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of generics.
- Using generics it is possible to create a single class that automatically works with different types of data.
- A Generic class simply means that the items or functions in that class can be generalized with the parameter(example T) to specify that we can add any type as a parameter in place of T like Integer, Character, String, Double or any other user-defined type.

Generics

Advantage of Java Generics:

There are mainly 3 advantages of generics. They are as follows

1) Type-safety:

- We can hold only a single type of objects in generics. It doesn't allow to store other objects.

Ex:
```
List list = new ArrayList();
list.add(10);
list.add("10");
```
With Generics, it is required to specify the type of object we need to store.
```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) Type casting is not required: There is no need to typecast the object.

Ex:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking:

- It will check the type at compile time so problem will not occur at runtime.
- The good programming strategy says it is far better to handle the problem at compile time than runtime.

Example:

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Example:

```
import java.util.ArrayList;
import java.util.Collections;
public class SortNames
{
    public static void main(String[] args)
    {
        ArrayList<String> al=new ArrayList<String>();
        al.add("Bhaanu");
        al.add("Chandhu");
        al.add("Divya");
        al.add("Abhi");
        al.add("Eesha");
        System.out.println("Before sorting");
        System.out.println("----------------------");
        for (String name:al)
            System.out.println(name);
        Collections.sort(al);
        System.out.println("Before sorting");
        System.out.println("----------------------");
        for (String name:al)
            System.out.println(name);

    }
}
```

User defined Generic class

Example:

```
public class Ex1
{
    public static void main(String[] args)
    {
        MyClass<String> m1=new MyClass<String>("100");
        System.out.println("Generic class is returning :"+m1.getInfo());
        MyClass<Double> m2=new MyClass<Double>(10.25);
        System.out.println("Generic class is returning :"+m2.getInfo().getClass());
    }
}
class MyClass<T>
{
    T t;
    MyClass(T t)
    {
        this.t=t;
    }
    public T getInfo()
    {

        return t;
    }
}
```

Example:

```
public class Ex2
{
    public static void main(String[] args)
    {
        Show<Float> s=new Show<Float>(10.5f);
        System.out.println(s.getInfo());
        Show<Integer> s1=new Show<Integer>(10);
        System.out.println(s1.getInfo());
    }
}
class Show<T extends Number>
{
    T t;
    Show(T t)
    {
        this.t=t;
    }
    public T getInfo()
    {
        return t;
    }
}
```

Random class

Random class in Java?

- In Java, Random class is a part of **java.util** package.

- The generation of random numbers takes place by using an instance of the Java Random Class.

- This class provides different methods in order to produce random numbers of type boolean, integer, double, long, float, etc.

Constructors used in a Java Random class

This class contains two constructors that are mentioned below:

- **Random():** this constructor helps in creating a new random generator
- **Random(long seed):** this constructor helps in creating a new random generator using specified seed

Methods

Method	Functionality
nextDouble()	Returns the next pseudo-random number that is a double value between the range of 0.0 to 1.0.
nextBoolean()	Returns the next pseudo-random which is a Boolean value from random number generator sequence
nextFloat()	Returns the next pseudo-random which is a float value between 0.0 to 1.0
nextInt()	Returns the next pseudo-random which is an integer value from random number generator sequence
nextInt(Int n)	Returns the next pseudo-random which is an integer value between 0 and the specified value from random number generator sequence
nextBytes(byte[] bytes)	Generates random bytes and places them into a byte array supplied by the user
Longs()	Returns an unlimited stream of pseudorandom long values
nextGaussian()	Helps in returning the next pseudo-random, Gaussian (precisely) distributed double value with mean 0.0 and standard deviation 1.0 from this random number generator’s sequence

Example:

```
import java.util.Random;
public class RandomDemo
{
    public static void main(String[] args) {
        Random rnd=new Random();
        System.out.println("Random boolean :"+rnd.nextBoolean());
        byte b[]=new byte[10];
        rnd.nextBytes(b);
        System.out.print("Random bytes   : ");
        for (byte n:b)
        {
            System.out.print(n+" ");
        }
    }
}
```

Output:

```
Random boolean :false
Random bytes   : -36 74 119 70 -65 117 -58 127 121 -5
```

StringTokenizer

StringTokenizer

- The string tokenizer class allows an application to break a string into tokens.
- The tokenization method is much simpler than the one used by the StreamTokenizer class
- The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.
- An instance of StringTokenizer behaves in one of two ways, depending on whether it was created with the returnDelims flag having the value true or false:

StringTokenizer methods

int	countTokens()Calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception.
boolean	hasMoreElements()Returns the same value as the hasMoreTokens method.
boolean	hasMoreTokens()Tests if there are more tokens available from this tokenizer's string.
Object	nextElement()Returns the same value as the nextToken method, except that its declared return value is Object rather than String.
String	nextToken()Returns the next token from this string tokenizer.
String	nextToken(String delim)Returns the next token in this string tokenizer's string.

Example:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens())
{
    System.out.println(st.nextToken());
}
```

prints the following output:

```
this
is
a
test
```

- **StringTokenizer** is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.

- It is recommended that anyone seeking this functionality use the split method of String or the java.util.regex package instead.

- The following example illustrates how the **String.split()** can be used to break up a string into its basic tokens:

Example:

```
String[] result = "this is a test".split("\\s");
for (int x=0; x<result.length; x++)
    System.out.println(result[x]);
```

Output:

```
this
is
a
test
```

Scanner Class

Scanner class

- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings.
- It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint.
- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read entire line of Strings we can use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

Methods

boolean hasNext( )
boolean hasNext(String pattern)
boolean hasNextBigInteger( )
boolean hasNextBoolean( )
boolean hasNextByte(int radix)
boolean hasNextFloat( )
boolean hasNextInt(int radix)
boolean hasNextLong( )
boolean hasNextShort( )
String next( )
boolean nextBoolean( )
double nextDouble( )
String nextLine( )
short nextShort( )

boolean hasNext(Pattern pattern)
boolean hasNextBigDecimal( )
boolean hasNextBigInteger(int radix)
boolean hasNextByte( )
boolean hasNextDouble( )
boolean hasNextInt( )
boolean hasNextLine( )
boolean hasNextLong(int radix)
boolean hasNextShort(int radix)
String next(String pattern)
byte nextByte( )
int nextInt( )
long nextLong()

Calendar class

Calendar class

- Calendar class in Java is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc.
- It inherits Object class and implements the Comparable, Serializable, Cloneable interfaces.
- As it is an Abstract class, so we cannot use a constructor to create an instance. Instead, we will have to use the static method **Calendar.getInstance()** to instantiate and implement a sub-class..
- java.util.Calendar is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object.
- Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar.
- Currently, java.util.GregorianCalendar for the Gregorian calendar is supported in the Java API.

Methods

- Calendar provides no public constructors.
- **static Calendar getInstance( ) :** Returns a Calendar object for the default locale and time zone.
- **int get(int calendarField) :** Returns the value of one component of the invoking object.
- The component is indicated by calendarField.
- Examples of the components that can be requested are Calendar.YEAR, Calendar.MONTH, Calendar.MINUTE, and so forth.

- **final Date getTime( ) :** Returns a Date object equivalent to the time of the invoking object.

- **final void set(int year, int month, int dayOfMonth) :** Sets various date and time components of the invoking object.

- **final void setTime(Date d) :** Sets various date and time components of the invoking object. This information is obtained from the Date object d.

METHOD	DESCRIPTION
abstract void add(int field, int amount)	It is used to add or subtract the specified amount of time to the given calendar field, based on the calendar's rules.
int get(int field)	It is used to return the value of the given calendar field.
abstract int getMaximum(int field)	It is used to return the maximum value for the given calendar field of this Calendar instance.
abstract int getMinimum(int field)	It is used to return the minimum value for the given calendar field of this Calendar instance.
Date getTime()	It is used to return a Date object representing this Calendar's time value.</td

Example:

```
import java.util.*;
public class Calendar1 {
    public static void main(String args[])
    {
        Calendar c = Calendar.getInstance();
        System.out.println("The Current Date is:" + c.getTime());
    }
}
```

Output:

```
The Current Date is:Tue Aug 28 11:10:40 UTC 2022
```

Example:

```
import java.util.*;
public class Calendar3 {
    public static void main(String[] args)
    {
        // creating calendar object
        Calendar calendar = Calendar.getInstance();

        int max = calendar.getMaximum(Calendar.DAY_OF_WEEK);
        System.out.println("Maximum number of days in a week: " + max);

        max = calendar.getMaximum(Calendar.WEEK_OF_YEAR);
        System.out.println("Maximum number of weeks in a year: " + max);
    }
}
```

Output:

```
Maximum number of days in a week: 7
Maximum number of weeks in a year: 53
```

Example:

```
import java.util.Calendar;

public class Ex20
{
    public static void main(String[] args)
    {
        String[]
month={"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
        Calendar calendar=Calendar.getInstance();
        System.out.println("Calendar Type:"+calendar.getCalendarType());
        System.out.println("Time Zone :"+calendar.getTimeZone().getID());
        System.out.print("Date:");
        System.out.print(month[calendar.get(Calendar.MONTH)]);
        System.out.print(" "+calendar.get(Calendar.DAY_OF_MONTH));
        System.out.print(" "+calendar.get(Calendar.YEAR));
    }

}
```

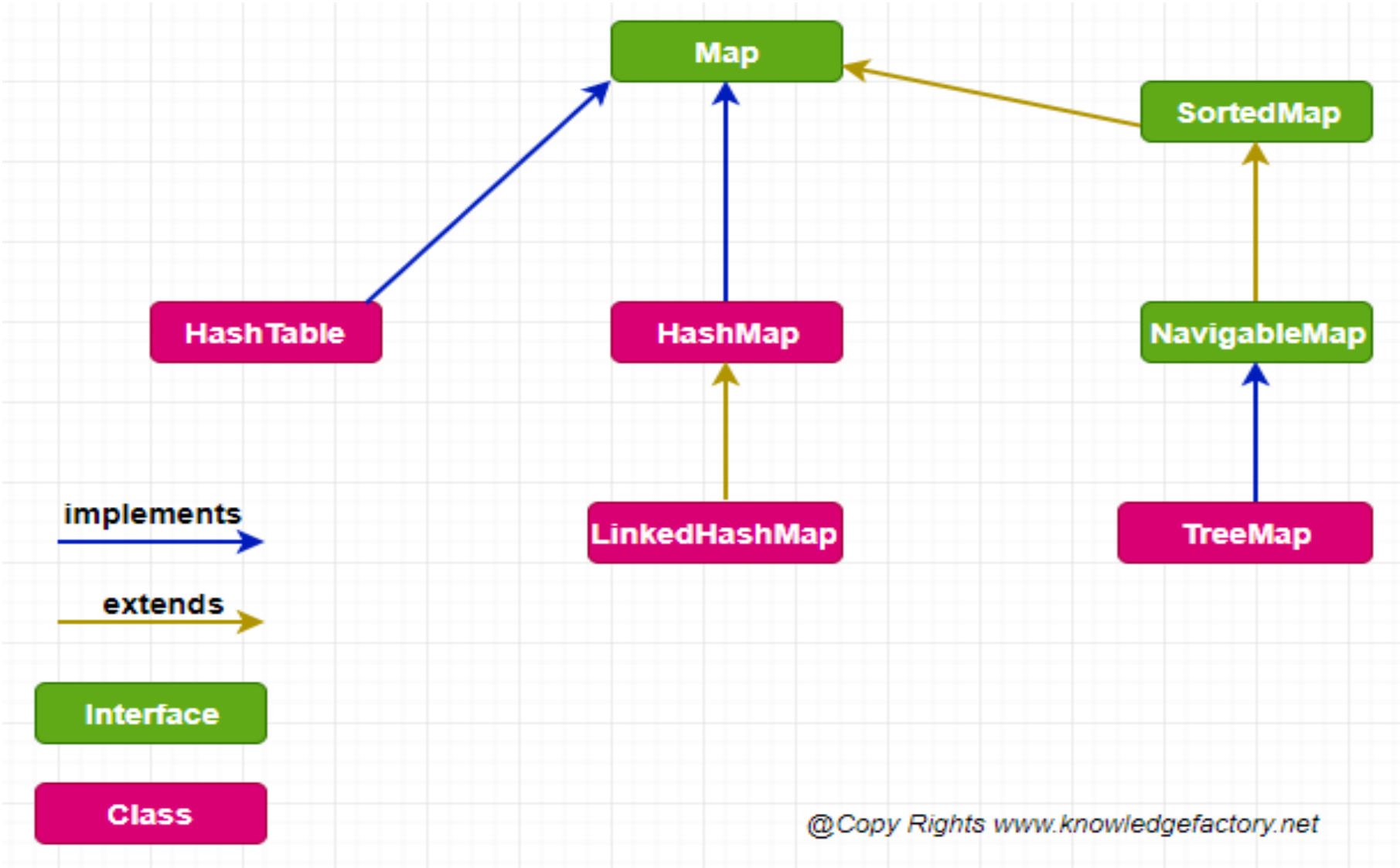
Output:

```
Calendar Type:gregory
Time Zone :Asia/Calcutta
Date:Jun 16 2022
```

MAP

Map properties

- If we want to represent a group of objects as "key-value" pair then we should use Map interface.
- Both key and value are objects only.
- Duplicate keys are not allowed but values can be duplicated
- Each key-value pair is called "one entry".
- Map interface is not child interface of Collection and hence we can't apply Collection interface methods here.
- Map interface defines the following specific methods.


Map
SortedMap
HashTable
HashMap
NavigableMap
LinkedHashMap
TreeMap
implements
extends
Interface
Class
@Copy Rights www.knowledgefactory.net

HashMap:

HashMap:

- The underlying data structure is Hashtable.
- Duplicate keys are **not allowed** but values can be duplicated.
- Insertion order is not preserved and it is based on hash code of the keys.
- Heterogeneous objects are allowed for both key and value.
- Null is allowed for keys(only once) and for values(any number of times).
- It is best suitable for **Search** operations.