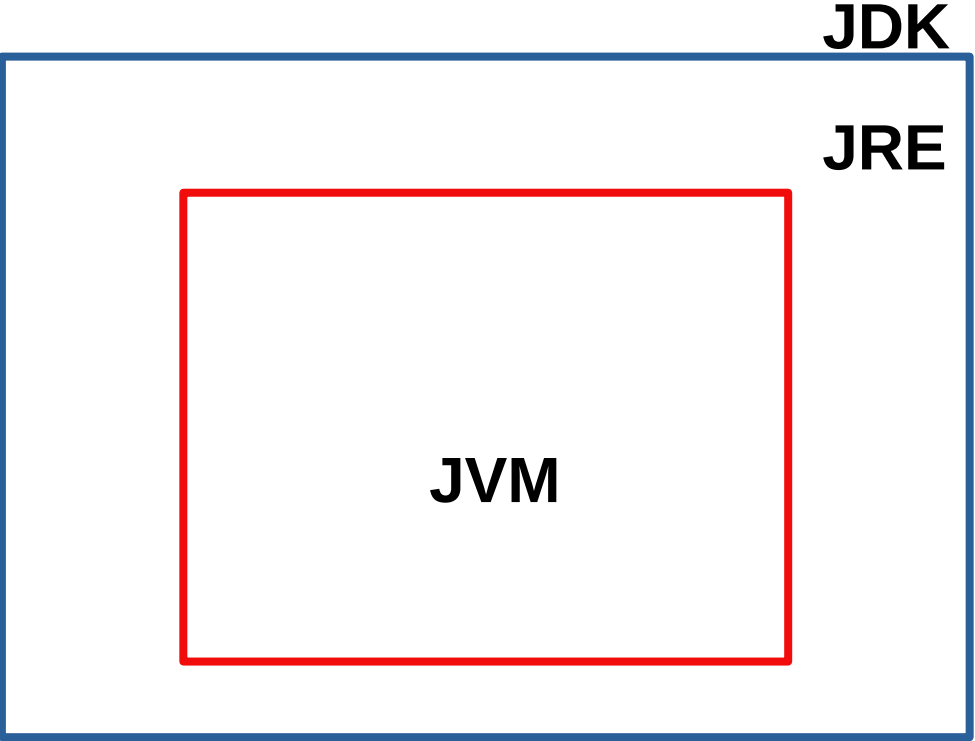
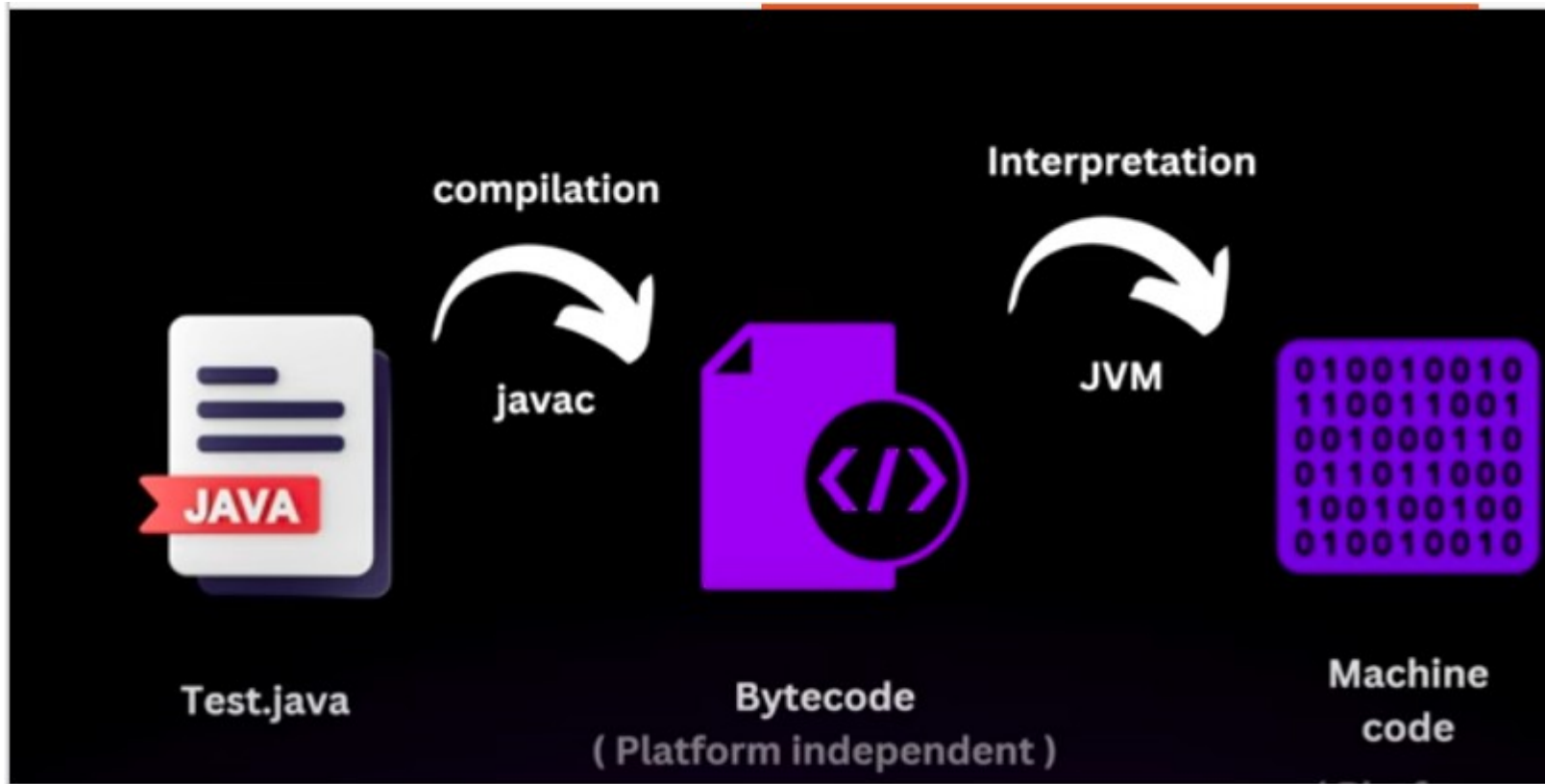


# CORE JAVA

1. Opend JDK Installation  
<https://adoptium.net/en-GB/>
2. Environment Path Setting.





# Understanding The Skeleton Of a Java Program

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

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

This is the main method. In Java, the main method is the entry point of a program. It has a specific signature:

**public:** Access modifier indicating that the method can be accessed from outside the class.

**static:** Indicates that the method belongs to the class rather than an instance of the class.

**void:** Specifies that the method does not return any value.

**main:** The name of the method.

**String[] args:** The method accepts an array of strings as parameters. This is where command-line arguments can be passed to your program.

# Primitive Data types

Integral number

// byte

// short

// int

// long

Decimal number

// float

// double

Characters

boolean

# Aithmetic operators

- +, -, \*, /, %
- ++
- --
- Pre , Post increment
-

# Arithmetic operators

```
public static void main(String[] args) {  
    int yourSalary = 1000;  
    int deduction = 500;  
    int monthlyTotal = yourSalary - deduction;  
    int yearlyTotal = monthlyTotal * 12;  
    int perChild = yearlyTotal / 3;  
    System.out.println(perChild);  
}
```



# String

strings are **immutable**, means their values cannot be changed once they are created.

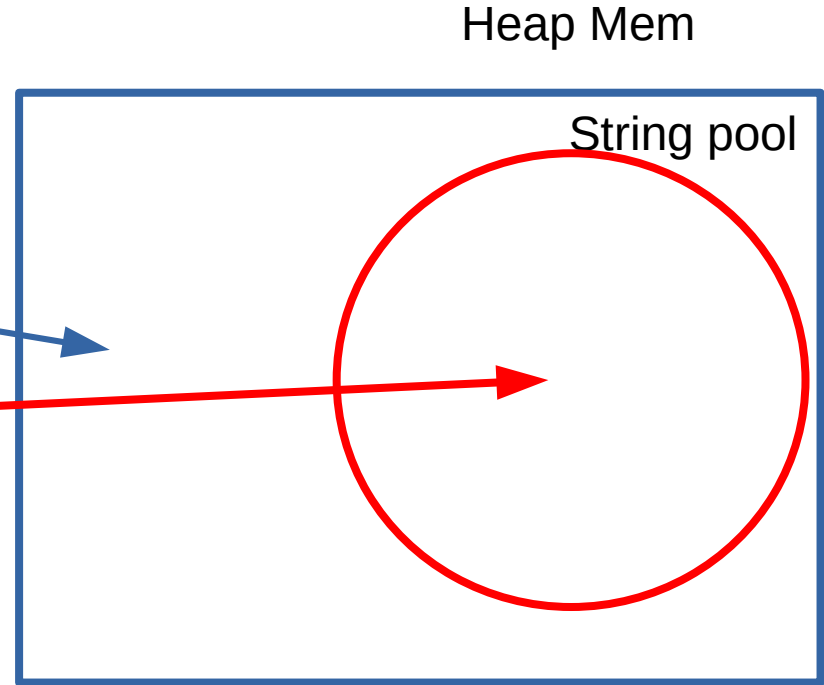
which means that once a string object is created, its content cannot be changed. If we try to change a string, a new string object is created instead.

# String

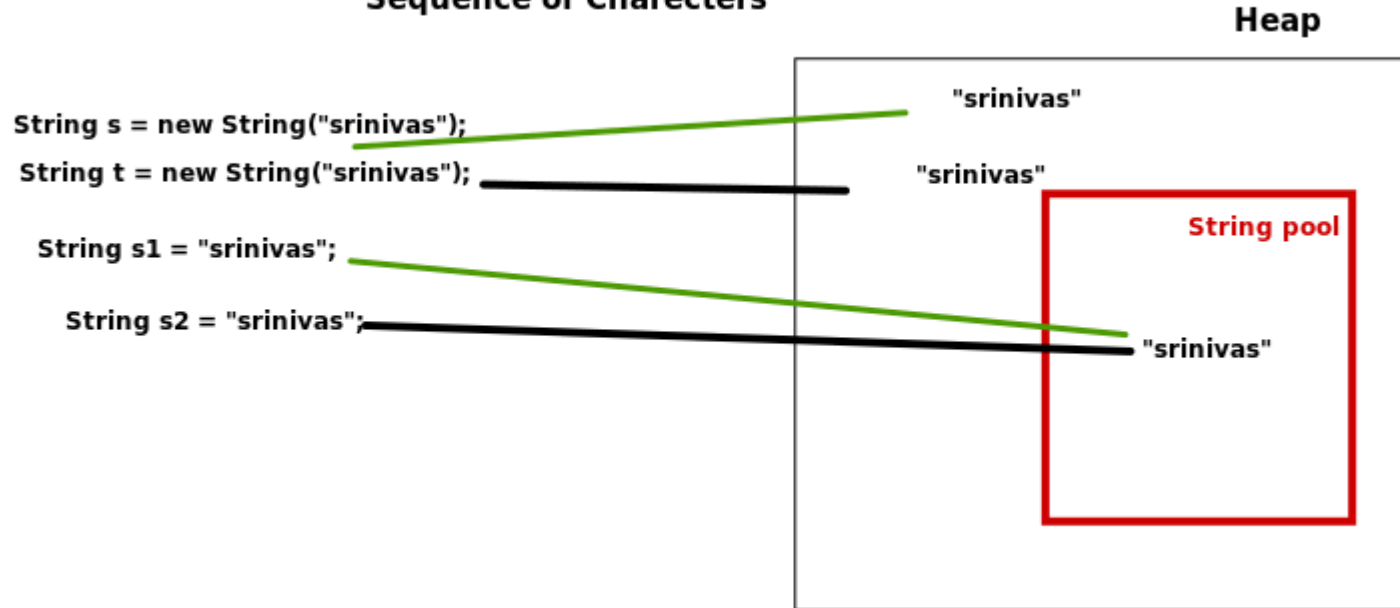
Non primitive data type

`String a = new String("Ram");`

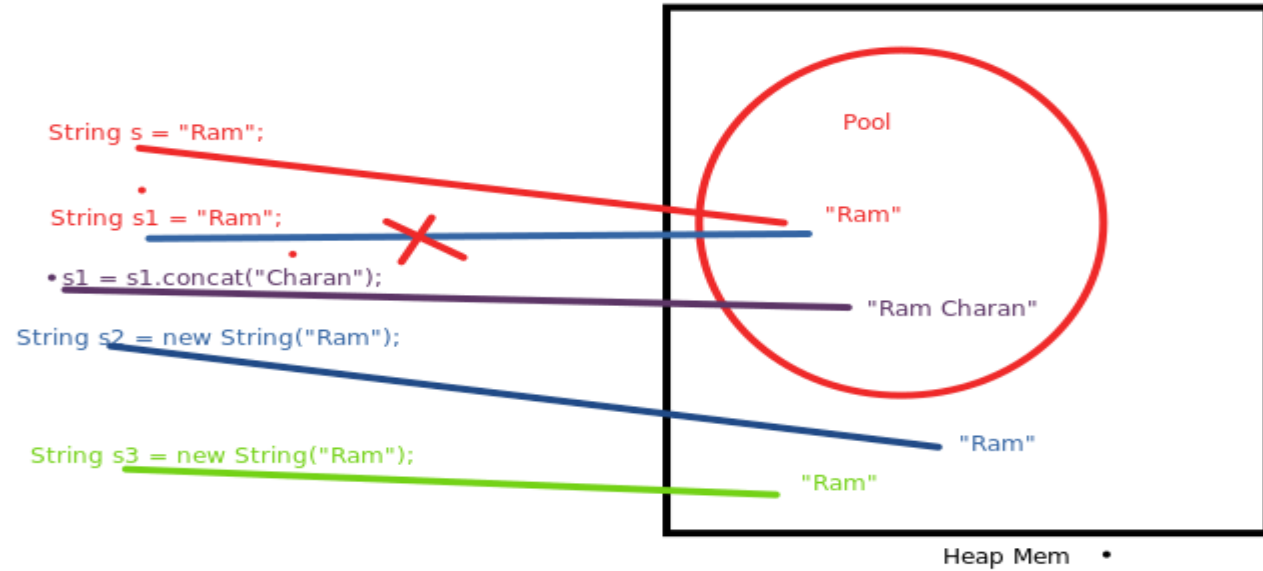
`String b = "Ram";`



**String : It is a non primitive data type,  
Sequence of Charecters**



# String Pool



# Conditional statements

Logical operators:

&& logical And

|| logical OR

! logical Not.

Relation Operators: return boolean values

>, <, >=, <=, ==, !=

```
int marks = 99;  
// marks >= 90 A  
// marks >= 75 B  
// marks >= 60 C
```

```
if (marks >= 90) {  
    print("Grade A");  
} else if (marks >= 75) {  
    print("Grade B");  
} else if (marks >= 60) {  
    print("Grade C");  
} else {  
    print("Grade D");  
}
```

# If else & switch

```
int day = 3;
if (day == 1) {
    System.out.println("Monday");
} else if (day == 2) {
    System.out.println("Tuesday");
} else if (day == 3) {
    System.out.println("Wednesday");
} else if (day == 4) {
    System.out.println("Thursday");
} else if (day == 5) {
    System.out.println("Friday");
} else if (day == 6) {
    System.out.println("Saturday");
} else if (day == 7) {
    System.out.println("Sunday");
} else {
    System.out.println("Invalid day");
}
```

```
switch (day){
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
}
```

# Loops

- While
- Do while
- For
- 

```
// syntax
while(condition) {
    System.out.println("Hello world!");
}
```

```
//Print hello world 10 times
int i=0;
while(i < 10) {
    System.out.println("Hello world!");
    i = i+1; // increment i by 1 every time
}
```



```
| int i=0;  
while (i < 10) {  
    System.out.println("Hello world");  
    i = i+1;  
}
```

```
for (int j = 0; j < 10; j++) {  
    System.out.println("Hello world");  
}
```

```
while (condition) {  
    // Code to execute  
}
```

Example:

```
int i = 1;  
while (i <= 5) {  
    System.out.println(i);  
    i++;  
}
```

```
for (initialization; condition; increment/decrement) {  
    // Code to execute  
}
```

Example:

```
for (int i = 1; i <= 5; i++) {  
    System.out.println(i);  
}
```

Feature	<code>while</code> Loop	<code>for</code> Loop
Use Case	When the termination condition depends on logic or user input.	When you know the loop bounds (start and end).
Initialization	Done outside the loop.	Done within the loop declaration.
Condition Checking	Only a condition is checked before each iteration.	Initialization, condition, and increment are all in one line.
Readability	Good for complex conditions or dynamic termination.	Clear and concise for loops with fixed bounds.

# Practice programs

## Examples

- Print \* patterns
- Count number of digits in a number
- Ex: 46734633 ans: 8 digits
- Factorial of a number.

# Nested Loops

\*

\*\*

\*\*\*

\*\*\*\*

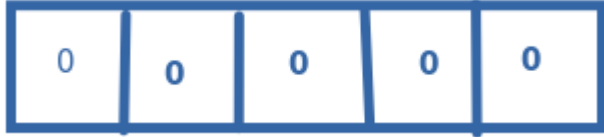
\*\*\*\*\*

# Arrays

- Collection of same kind of data.
- `Int x = 1;`
- `Int y = 2;`
- `Int z = 3;`
- `Int[] ar = {1,2,3};`
  - The array object (ar) is allocated on the
  - heap because arrays are objects in Java

```
Int[] arr = new int[5];
```

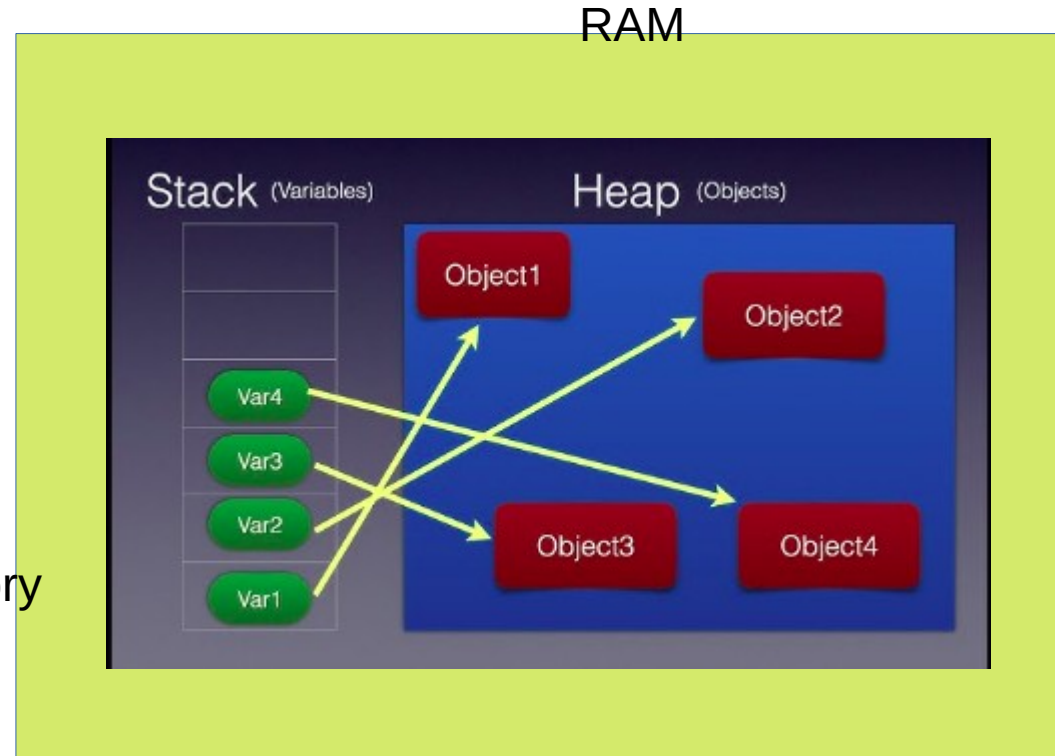
Created array obj of size 5 in Heap memory  
And arr in Stack memory.

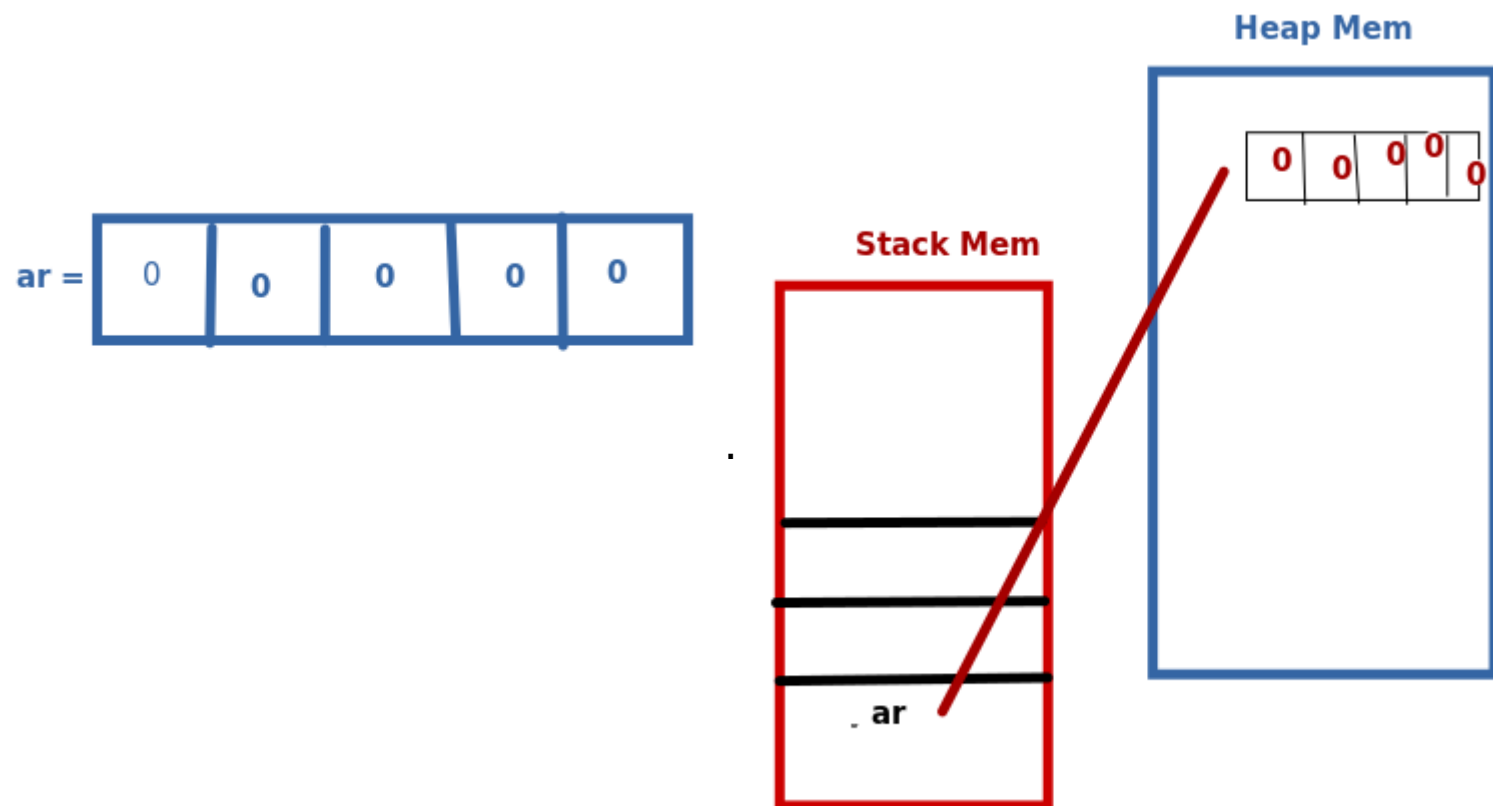


2 types of memory

- 1) Stack
  - 2) Heap
- Stored in RAM

JVM creates & manages these memory







# 2D Array

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

arr:

```
+---+---+---+  
| 0 | 0 | 0 |   arr[0]  
+---+---+---+  
| 0 | 0 | 0 |   arr[1]  
+---+---+---+  
| 0 | 0 | 0 |   arr[2]  
+---+---+---+
```

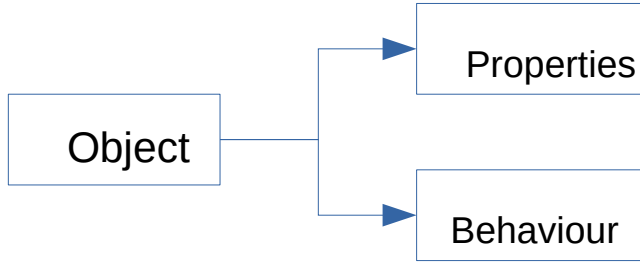
```
int[][] arr = new int[3][3];  
int[][] nums = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9}  
};
```

# Methods/Functions

- Syntax:
- Accessmodifier

# OOPs

- Object Oriented Programing.
- Oops – is a pattern/model/way of programing.
- Deals with Classes & Objects.
- Class – blue print
- Object – real world entity.



- With help of Blueprint(class).
- We create objects.

# Oops - principles

- **Encapsulation** – wrapping up of data (fields/attributes/variables) and methods(functions) into a single unit (class) and restricting direct access to them. Instead, access is provided through getter and setter methods.
- This promotes **Security**.
- **Inheritance** - that allows a class (child/subclass) to inherit the properties and behaviors of another class (parent/superclass). This promotes code **reuse**.
- The **extends** keyword is used for class inheritance.
- Java does not support multiple class inheritance to avoid ambiguity.
- Constructors are not inherited, but the parent class constructor can be called using **super()**.
- **Method Overriding** allows a subclass to provide a specific implementation of a method from the parent class.

# Constructor

- A constructor is a special method in Java used to initialize objects. It has the same name as the class and does not have a return type (not even void). It is automatically called when an object is created.
- Types of constructors:
  - 1) Default Constructor (No arguments)
- \*\*\* Constructor overriding concept is not there.

If you do not write any constructor, Java will automatically provides a Default constructor for your class.

- Ex: class Developer {  
    public Developer() {  
        sout("This is default constructor.");  
    }  
}
- }
- Developer dev = new Developer(); -> default constructor.
-

- 2) Parameterised Constructor(With arguments).

- Ex: class Employee{  
    private int id;  
    private String name;  
    public Employee(int id, String name) {  
        this.id = id;  
        this.name = name;  
        sout("This is parameterised constructor.");  
- }  
    class Main{  
        psvm() {  
            Employee emp = new Employee(101, "Ram");  
        }  
- }  
}

\* **this** – is current class Instance/class variable

–

# Constructor Overloading

Like methods, constructors can be overloaded (i.e., multiple constructors with the same name but different parameters).

Ex: class Employee{

private int id;

private String name;

•

public Employee(){

    this.id = 0;

    this.name = "Unknown"

}

public Employee(int id, String name){

    this.id = id;

    this.name = name;

}



# Constructor overloading with *super* keyword

- The super keyword is used to call the **parent** class constructor from a subclass.

- Class Employee { // parent

```
    public Employee() {  
        sout("Emp class constructor called..");  
    }
```

- }

- Class Developer extends Employee { // child

```
    - public Developer() {  
        super();  
        sout("Developer class constructor called..");  
    }
```

- }

- Method **vs** Constructor

Feature	Constructor	Method
Purpose	Initializes objects	Defines behavior/actions
Return Type	No return type	Must have a return type ( <code>void</code> , <code>int</code> , etc.)
Invocation	Called automatically when object is created	Called explicitly using object reference
Name	Must match class name	Can have any name
Overloading	✔ Yes	✔ Yes
Overriding	✗ No	✔ Yes

- **Polymorphism**- It allows a method behaves differently based on the object calling it). This improves flexibility, and maintainability.
- Example : Calculator class with overloaded method i.e add(.....);
- Types of polymorphism:
- 1) Compile-time Polymorphism (Method Overloading)
  - i.e same method name but different parameters.
    - add(int a, int b)
    - add(float x, float y)
    - add(int i, int j, int k)
- 2) Runtime Polymorphism (Method Overriding)
  - method overriding (same method signature in parent and child class).
  - Parent class                      Child class
  - Employee emp = new Manager();
  - emp.calculateSalary();
  - Employee emp = new Developer();
  - emp.calculateSalary();

```

class Emp { // parent
    private int id;
    private String name;
    public Emp(int id, String name){ .....}
    public double calculateSalary(){
        sout("Emp/parent class method...");
        return 10000;
    }
}

```

```

Class Dev extends Emp {
    public Dev(int id, String na){
        super(id, na);
    }
    @override
    public double calculateSalary(){
        sout("Dev/child class method...");
        return 30000;
    }
}

```

- }

## Compiler(LHS)



## JVM(RHS)



- Employee emp = new Developer();
- Super/Parent class ref = child class object

Method Overloading (Compile-time)	Method Overriding (Runtime)
Multiple methods with the same name but different parameters	Subclass redefines a method from the parent class
Compile-time (static binding)	Runtime (dynamic binding)
Different parameter list	Same method signature
✗ No	✓ Yes (must be in a subclass)

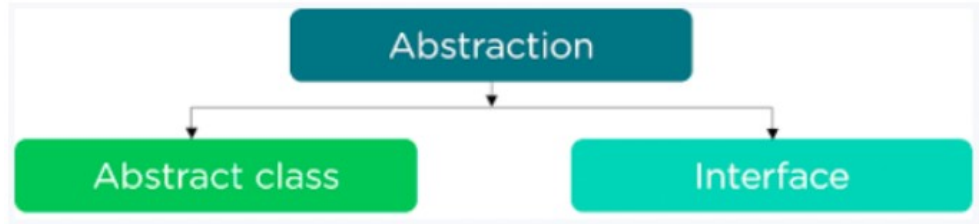
# Method overriding with *super* keyword

```
class Emp { // parent
    private int id;
    private String name;
    public Emp(int id, String name){ .....}
    public double calculateSalary(){
        sout("Emp/parent class method...");
        return 10000;
    }
}
```

```
Class Dev extends Emp {
    public Dev(int id, String na){
        super(id, na);
    }
    @override
    public double calculateSalary(){
        // call parent class salary first
        double genearlSal = super.calculateSalary();
        sout("Dev/child class method...");
        return genearlSal + 20000;
    }
}
```

# Abstraction

- Hiding implementation details and exposing only the necessary features. It helps in reducing complexity.
- abstraction is implemented using:
  - 1) Abstract Classes (abstract keyword)
  - 2) Interfaces (interface keyword)



- \* Abstract methods have no body and must be implemented by subclasses.
- \* Concrete methods have a body and can be inherited directly.

- **Abstraction using class:**

- 1) can't be instantiated.(can't create object)
- 2) can contain both abstract & normal methods.
- 3) defined using abstract keyword.

Ex: abstract class Employee {  
    public abstract double calculateSalary(); // no  
    method body only declaration

- Public void displayEmpDetails() { // concrete method  
    sout("Emp details.....");  
}

Class Manager extends Employee {  
    @Override  
    - public double calculateSalary() {  
        return 100000;  
    - }  
}



# Abstraction Using Interfaces

- What is interface ?
- Only contains abstract methods. no implementation.
- A class **implements** an interface using the implements keyword.
- A class can implement multiple interfaces.
- It cannot have a constructor.
- All methods are implicitly public and abstract.
- All variables are public, static, and final (constants).
- **When you want to define a contract for multiple classes.**

# Multiple interfaces

- interface Animal {
- void eat();
- }
- 
- interface Pet {
- void play();
- }

- class Dog implements Animal, Pet {
- public void eat() {
- System.out.println("Dog is eating.");
- }
- 
- public void play() {
- System.out.println("Dog is playing.");
- }
- }
-

## Marker interface

A marker interface has no methods.

Example: Serializable, Cloneable.

```
class MyClass implements Serializable {  
    // Just a marker, no methods required  
}
```

```
interface Vehicle{  
  
    int MAX_SPEED = 4; // Implicitly  
    public, static, and final  
  
    void start(); //no body  
  
    void stop();  
  
}
```

```
Class Car implements Vehicle {  
  
    @override  
    public void start(){  
        sout("Car is starting..");  
    }  
  
    @override  
    public void stop(){  
        sout("Car is stoping..");  
    }  
}
```
















# Abstract class **vs** interface

Feature	Abstract Class ( <code>abstract</code> )	Interface ( <code>interface</code> )
Abstraction Level	Partial (0-100%)	100% abstraction (before Java 8)
Method Type	Can have both <b>abstract &amp; concrete</b> methods	Only <b>abstract methods</b> (before Java 8)
Multiple Inheritance	✗ Not supported	✓ Supported
Default Methods	✗ No	✓ Yes (Java 8+)
Static Methods	✓ Yes	✓ Yes (Java 8+)
Fields	Can have instance variables	Only <code>public static final</code> variables
Constructors	✓ Yes	✗ No

# Access modifiers

- Access modifiers in Java control the visibility and accessibility of classes, methods, and variables. They help in encapsulation, ensuring data security and integrity.
- **public**: Accessible everywhere (same class, package, subclass, world).
- **protected**:
  - Accessible within the same package.
  - Accessible in subclasses outside the package via inheritance.
  - Not accessible outside the package (unless inherited).
- **Default** (no modifier):
  - Accessible only within the same package.
  - Not accessible outside the package, even in subclasses.
- **private**: Most restrictive.
  - Accessible only within the same class.
  - Not accessible in subclasses or other classes (even in the same package).
-

# Access Modifier scopes

Access Modifier	Class	Package	Subclass (outside package)	World (outside class & package)
<code>public</code>	 Yes	 Yes	 Yes	 Yes
<code>protected</code>	 Yes	 Yes	 Yes	 No
<b>(default)</b> ( <i>No modifier</i> )	 Yes	 Yes	 No	 No
<code>private</code>	 Yes	 No	 No	 No

# static keyword

- It can be applied to variables, methods, blocks & nested classes.
- The main concept behind static is that it belongs to class not instance/object of the class.
- Ex: class Emp {  
    static int count = 0;  
}
- Attached to class & only one copy of count variable.
- Access using class name. - Emp.count;
- Static methods can't use non static data members or call non-static methods directly.  
- **this** & **super** can't be used in **static** context.
- Because this & super refers to instance/object of class.
- Static belongs to class, not belongs to any one object of that class.



# Static block

```
public class Employee {  
  
    // static variable  
    private static int count;  
  
    static {  
        System.out.println("static block...");  
        count = 0;  
        // static initialisation  
        // one time setup task  
    }  
  
    private int id;  
    private String name;  
  
    public Employee() {  
        System.out.println("constructor....");  
        count++;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    //static method  
    public static int getCount() {  
        return count;  
    }  
}
```

```
public class Test {  
  
    public static void main(String[] args) {  
  
        //    Employee e1 = new Employee();  
        //    e1.setId(1);  
        //    e1.setName("Ram");  
        //  
        //    Employee e2 = new Employee();  
        //    e2.setId(2);  
        //    e2.setName("Arjun");  
        //  
        System.out.println(Employee.count);  
  
        //    System.out.println(Employee.getCount());  
  
    }  
}
```

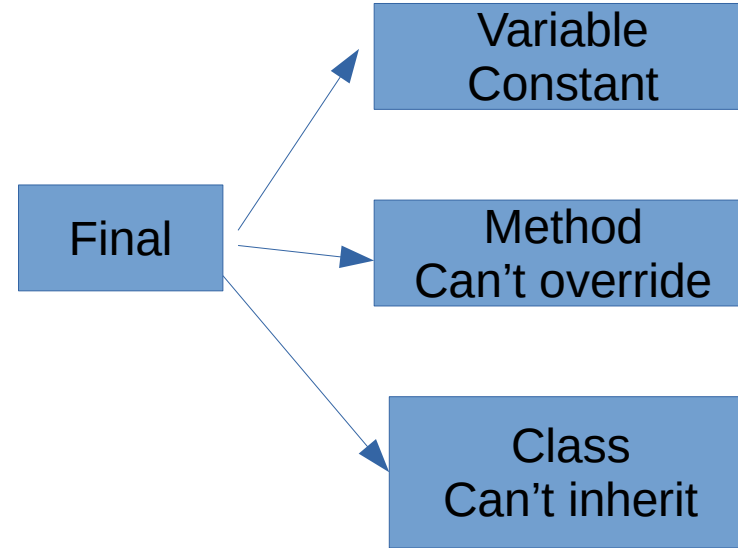
# Final keyword

Final means once assign can't change.

keyword used on

- Method
- Variable
- Class

ex : `public static final PI = 3.14`



# Inner class

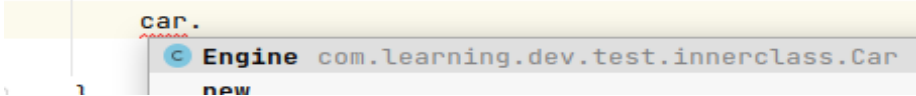
- Inner class is a class that is defined within another class.
- grouping related classes together, and providing access to members of the outer class.
- Member Inner Class
- Static Nested Class
- \*Anonymous Inner Class(nameless)

# Member Inner class

```
public class Car {  
  
    private String model; // member variables  
    private boolean isEngineOn;  
  
    public Car(String model) { //Constructor  
        this.model = model;  
        isEngineOn = false;  
    }  
  
    class Engine { // class is also member of Car class  
        void start() {  
            if (!isEngineOn) {  
                System.out.println(model + " Engine started..");  
            } else {  
                System.out.println(model + " Engine already on..");  
            }  
        }  
  
        void stop() {  
            if (isEngineOn) {  
                isEngineOn = false;  
                System.out.println(model + " Engine Stopped..");  
            } else {  
                System.out.println(model + " Engine already off..");  
            }  
        }  
    }  
}
```

- Inner class is a member of Outer class & is associated with Outer class.

- ```
public class Test {  
    public static void main(String[] args) {  
        Car car = new Car( model: "Tata Safari");  
        car.  
    }  
}
```



- Creating Engine object with help of Car(outer class).

```
Car car = new Car("Tata Safari");  
Car.Engine engine = car.new Engine();  
engine.start();  
engine.stop();
```

# Use case of inner class

```
public class Car {  
  
    private String model; // member variables  
    private boolean isEngineOn;  
  
    public Car(String model) { //Constructor  
        this.model = model;  
        isEngineOn = false;  
    }  
  
    public String getModel() {  
        return model;  
    }  
  
    public void setModel(String model) {  
        this.model = model;  
    }  
  
    public boolean isEngineOn() {  
        return isEngineOn;  
    }  
  
    public void setEngineOn(boolean engineOn) {  
        isEngineOn = engineOn;  
    }  
}
```

```
public class Engine {  
  
    void start() {  
        if (!isEngineOn) {  
            System.out.println("Engine started..");  
        } else {  
            System.out.println("Engine already on.");  
        }  
    }  
  
    void stop() {  
        if (isEngineOn) {  
            isEngineOn = false;  
            System.out.println(model+" Engine Stopped..");  
        } else {  
            System.out.println(model+" Engine already off.");  
        }  
    }  
}
```

Cannot resolve symbol 'isEngineOn'  
Create local variable 'isEngineOn' Alt+Shift+E

# solution

```
public class Engine {  
  
    private Car car;  
  
    public Engine(Car car) { // pass car object to engine class  
        // constructor  
        this.car = car;  
    }  
  
    void start() {  
        if (!car.isEngineOn()) {  
            System.out.println(car.getModel() + " Engine started..");  
        } else {  
            System.out.println(car.getModel() + " Engine already on..");  
        }  
    }  
  
    void stop() {  
        if (car.isEngineOn()) {  
            car.setEngineOn(false);  
            System.out.println(car.getModel() + " Engine Stopped..");  
        } else {  
            System.out.println(car.getModel() + " Engine already off..");  
        }  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Car car = new Car("Tata safari");  
        Engine engine = new Engine(car);  
        engine.start();  
        engine.stop();  
    }  
}
```

\*\*\* In this case we need to write constructor in

Engine class & pass Car object. To avoid this simple we can

Write Engine as Inner class in Car class.

Then Engine will become a member of Car class.

# Static inner class

- Static inner class belongs to the outer class rather than its instance.(where as member inner class associate with instance of outer class).
- Static inner class is part of Outer class, not belongs to any instance/object of a class.
- Does not require an instance of the outer class to be instantiated.
- Cannot access non-static members (fields or methods) of the outer class directly.
- \*\*\* You can create an instance of a static inner class without an instance of the outer class.
- A real-world use case of a static inner class is when you want to create a helper class that is tightly coupled with the outer class but does not require an instance of the outer class.

```

public class Computer {

    private String modelName;
    private String brand;
    private OperatingSystem os;

    public Computer(String modelName, String brand) {
        this.modelName = modelName;
        this.brand = brand;
    }

    class OperatingSystem{
        private String osName;

        public OperatingSystem(String osName) {
            this.osName = osName;
        }

        public void displayInfo() {
            System.out.println("Computer: "+modelName+" Os: "+osName);
        }
    }

    static class USB {
        private String type;

        public USB(String type) {
            this.type = type;
        }

        public void displayInfo() {
            System.out.println("USB type: "+type);
        }
    }
}

```

```

public class StaticTest {

    public static void main(String[] args) {
        Computer computer = new Computer("abc", "Dell");
        //member inner class
        Computer.OperatingSystem os =
            computer.new OperatingSystem("xyz");
        os.displayInfo();

        // static inner class
        Computer.USB usb = new Computer.USB("type-c");
        usb.displayInfo();
    }
}

```



# Anonymous inner class

- When we think of Anonymous inner class:

when we want to implement **interface** without creating its separate implementation class, then we can think of anonymous inner class.

- Nameless inner class, because we are creating and using it on the fly, we are not naming this class.
- We know, cannot create object for interface, but here we are providing body for interface.

- Syntax:



```
ShoppingCart shoppingCart = new ShoppingCart( totalAmount: 150);
shoppingCart.processPayment(new Payment() {
    1 usage
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
});
```

The screenshot shows a Java code snippet. A yellow box highlights the anonymous inner class implementation of the `Payment` interface. A yellow arrow points to the `new Payment()` part of the `processPayment` call, and another yellow arrow points to the closing brace of the anonymous class.

```
public interface Payment {  
    void pay(double amount);  
}
```

```
public class ShoppingCart {  
  
    private double totalAmount;  
  
    public ShoppingCart(double totalAmount) {  
        this.totalAmount = totalAmount;  
    }  
  
    // taking Payment interface implemented  
    // class object as argument  
    // paymentMethod can be CC, UPI, Paypal..etc  
    public void processPayment(Payment paymentMethod) {  
        paymentMethod.pay(totalAmount);  
    }  
}
```

```
public class CreditCard implements Payment{  
  
    private String cardNumber;  
  
    public CreditCard(String cardNumber) {  
        this.cardNumber = cardNumber;  
    }  
  
    @Override  
    public void pay(double amount) {  
        System.out.println("Paid "+amount+" using Credit Card ");  
    }  
}  
  
public class AnonymousTest {  
    public static void main(String[] args) {  
  
        ShoppingCart shoppingCart = new ShoppingCart(250);  
        // CreditCard creditCard = new CreditCard("123456");  
        //CC payment  
        shoppingCart.processPayment(new Payment() {  
            @Override  
            public void pay(double amount) {  
                System.out.println("Paid "+amount+" using Credit Card ");  
            }  
        });  
  
        //UPI payment  
        shoppingCart.processPayment(new Payment() {  
            @Override  
            public void pay(double amount) {  
                System.out.println("Paid "+amount+" using UPI ");  
            }  
        });  
    }  
}
```

# Wrapper classes

## Primitive types:

### 1. Integer Types

| Data Type | Size    | Default Value | Range                   |
|-----------|---------|---------------|-------------------------|
| byte      | 1 byte  | 0             | -128 to 127             |
| short     | 2 bytes | 0             | -32,768 to 32,767       |
| int       | 4 bytes | 0             | $-2^{31}$ to $2^{31}-1$ |
| long      | 8 bytes | 0L            | $-2^{63}$ to $2^{63}-1$ |

### 2. Floating-Point Types

| Data Type | Size    | Default Value | Range                                                   |
|-----------|---------|---------------|---------------------------------------------------------|
| float     | 4 bytes | 0.0f          | $\pm 3.4\text{E}-38$ to $\pm 3.4\text{E}+38$ (approx)   |
| double    | 8 bytes | 0.0d          | $\pm 1.7\text{E}-308$ to $\pm 1.7\text{E}+308$ (approx) |

### 3. Character Type

| Data Type | Size    | Default Value | Description                                |
|-----------|---------|---------------|--------------------------------------------|
| char      | 2 bytes | '\u0000'      | Stores a single Unicode character (16-bit) |

### 4. Boolean Type

| Data Type | Size                  | Default Value | Values        |
|-----------|-----------------------|---------------|---------------|
| boolean   | 1 bit (JVM dependent) | false         | true or false |

- Wrapper class: wraps a value of primitive type in an Object.
- Ex: Integer b = 20;
- Int a = 10; -> "a" is a primitive type, "10" stores in Stack.
- Integer b = 20; -> "b" is a reference variable of class Integer, "20" stores in Heap.
- Ex: boolean isEngineOn = true;
- Boolean isEngineOn = true;
- Integer i = 10; -> **Autoboxing** : automatically converts primitive to Wrapper class,
- Integer i = Integer.valueOf(i); // not required
- Int j = i.intValue(); -> **Unboxing**.
- Can directly write – int j = i; -> same unboxing.
- Integer class has methods & static variables, explore Integer class.

# Enum

- Enum means Enumaration, is List of things.
- Ex: Days in a week, Months in a Year etc..

```
public class Day {  
    public static final String MON = "MONDAY";  
    public static final String TUE = "TUESDAY";  
    public static final String WEN = "WEDNESDAY";  
    public static final String THUR = "THURSDAY";  
    public static final String FRI = "FRIDAY";  
    public static final String SAT = "SATURDAY";  
    public static final String SUN = "SUNDAY";  
}
```

```
public static void main(String[] args) {  
    System.out.println(Day.MON);  
    System.out.println(Day.TUE);  
    System.out.println(Day.WEN);  
    System.out.println("TUESDAY");  
    System.out.println("SUNDAY");  
    System.out.println("FRIDAY");  
}
```

```
public interface Day {  
    String MON = "MONDAY";  
    String TUE = "TUESDAY";  
    String WEN = "WEDNESDAY";  
    String THUR = "THURSDAY";  
    String FRI = "FRIDAY";  
    String SAT = "SATURDAY";  
    String SUN = "SUNDAY";  
}
```

```
public enum Day {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  
}
```

MONDAY is-a Day.....

MONDAY is a instance/Obj of Day class....

Day monday = Day.MONDAY; // like Car c = new Car(); internally Moday obj will created.  
int index = monday.ordinal(); // starts with 0

```
Day[] values = Day.values();  
for(Day day : values) { // for each loop  
    System.out.println(day.name());  
}
```

Op:

```
public static void main(String[] args) {  
    Day[] values = Day.values();  
    for(Day day : values) { // for each loop  
        System.out.println(day.name());  
    }  
}
```

}

Op:

MONDAY

TUESDAY

WEDNESDAY

THURSDAY

FRIDAY

SATURDAY

SUNDAY

```
Day day = Day.MONDAY;  
switch (day) {  
    case MONDAY:  
        System.out.println("Today is Monday");  
        break;  
    case TUESDAY:  
        System.out.println("Today is Tuesday");  
        break;  
    case WEDNESDAY:  
        System.out.println("Today is Wednesday");  
        break;  
}
```

# StringBuffer&StringBuilder

## String is mutable or immutable ?

```
String s1 = "Hello";  
s1.concat("World");  
System.out.println(s1);
```

```
String s1 = new String("Hello");  
s1.concat("World");  
System.out.println(s1);
```

```
String s1 = "Hello";  
String s2 = s1.concat("World");  
System.out.println(s2);
```

```
String result = "";  
for (int i = 0; i < 100; i++) {  
    result = result + "HELLO"; // every time new string create in memory, i.e 100  
    strings  
}  
System.out.println(result);
```

To avoid creating new string every time, we can use StringBuffer/StringBuilder class.

```
//    StringBuilder sb = new StringBuilder("Hello");  
StringBuilder sb = new StringBuilder();  
sb.append("Hello");  
sb.insert(1, "Java");  
sb.replace(1, 3, "World");  
sb.delete(1, 4);  
sb.reverse();  
sb.charAt(1);  
sb.length();  
sb.substring(1, 4);
```

\*\*\* StringBuilder is not thread safe.

If we want thread safety. We should use StringBuffer.

StringBuffer provides thread safety.

Rest everything same as StringBuilder.

# Collections Framework

## What is a Collection ?

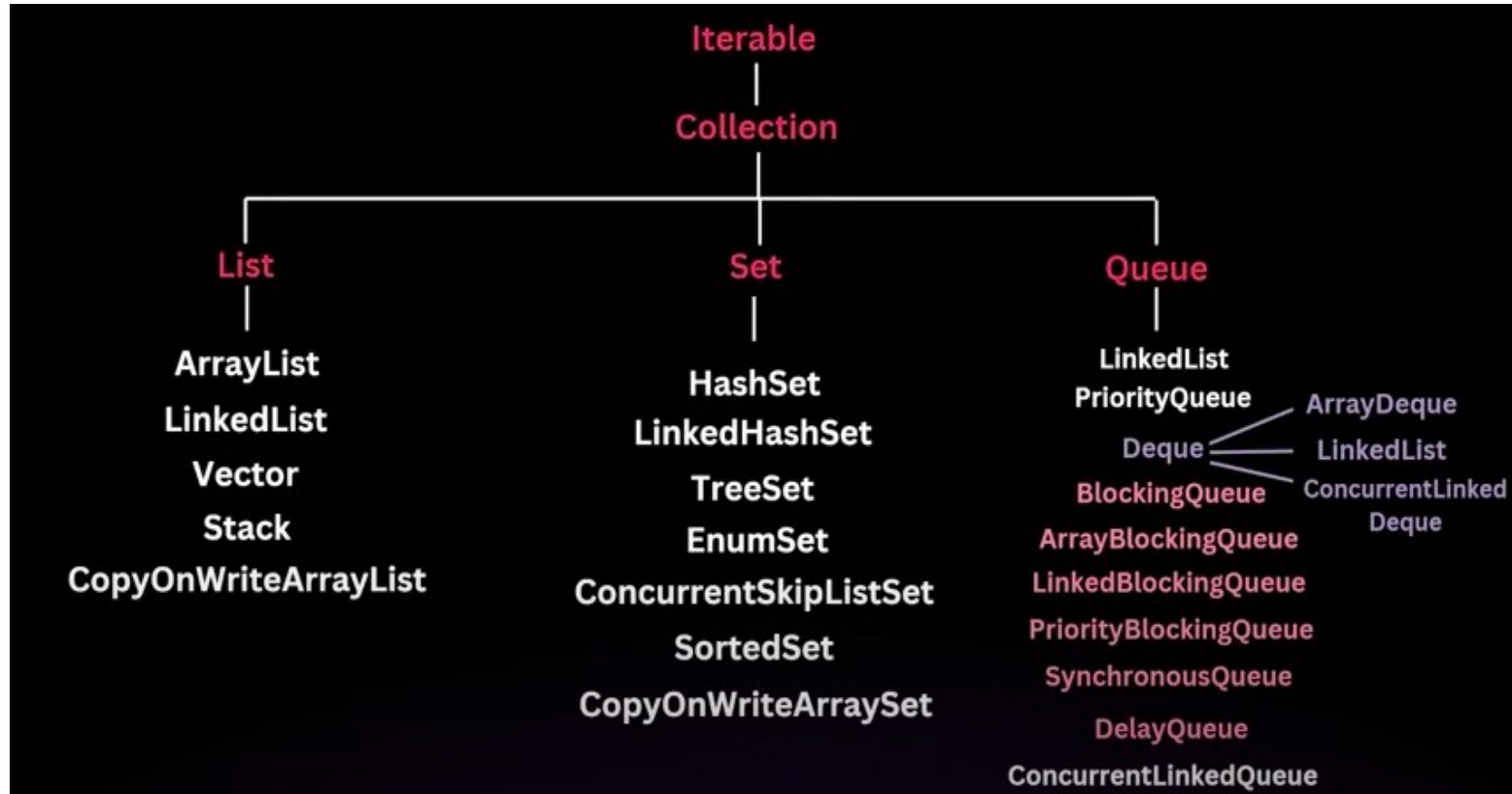
- A collection is simply an object that represents a group of objects, known as elements.
- What is collection framework – it provides a set of classes & interfaces that help in managing a group of elements.



# Key interface in collection framework

- **Collection** : the root interface for all other collection types.
- **List**: An ordered collection that can contain duplicate elements
- (eg. ArrayList, LinkedList).
- **Set**: A collection that cannot contain duplicate elements
- (eg. HashSet, TreeSet).
- **Queue**: A collection designed for elements prior to processing
- (eg. PriorityQueue, LinkedList when used as Queue).
- **Deque**: A double-ended queue that allows insertion & removal from both ends
- (eg. ArrayDeque).
- **Map**: An interface that represents a collection of key-value pairs
- (eg. HashMap, TreeMap).

# Hierarchy:



- **Iterable:** is an root interface of collection hierarchy.
- The collection interface is the root interface of the java collection framework.it is the most basic interface that defines a group of objects known as elements.
- The collection interface is part of java.util package.
- It is a parent interface that is extended by other collection interfaces like
- List, Set and Queue.
- Since collection is an interface, it cannot be instantiated directly,rather,
- it provides a blueprint for the basic operations that are common to all collections.
- The collection is an interface defines a set of core methods thats are implemented by all classes that implements the interface.
- These methods are allow for basic operations like adding, removing & checking element is exists in the collection.
-

# List Interface

- The List interface is part of java.util package and sub-interface of the Collection interface.
- It provides a way to store an **ordered collection of elements**. (Known as sequence).
- List can contain **duplicate** elements.
- **When to use List** – When we need ordered collection, In which order elements are coming store in same order and can have duplicates.
- The List interface is implemented by several classes in the java Collection Framework, such as ***ArrayList, LinkedList, Vector and Stack.***

## ***\*\*\* Key Features of List Interface***

- **Ordered Preservation**
- **Index based access**
- **Allow duplicates**

# ArrayList

- An ArrayList is a resizable array, implementation of List interface.
- Unlike arrays in java, which have a fixed size, An arrayList can change its size dynamically as elements are added or removed.
- This flexibility makes it a popular choice when the number of elements in a list is not known in advance.
- **Internal working** : ArrayList can grow & shrink as elements added or removed.
- This dynamic resizing achieved by creating a new array when the current array is full and copying the elements to the new array.
- \*\* internally, ArrayList is implemented as an array of Object references.
- When you add elements to the ArrayList, you are essentially storing these elements in this internal array.
- \*\* When you create an ArrayList, it has an initial capacity(default 10).
- The capacity refers the size of internal array that can hold elements before needing to resize.

# Creatin an ArrayList

*//Default constructor, creates an ArrayList with initial capacity of 10*

```
ArrayList<String> list = new ArrayList<>();
```

*//creating an ArrayList with specified initial capacity*

```
ArrayList<String> listWithCapacity = new ArrayList<>{20};
```

*//creating an ArrayList from another Collection*

```
List<String> anotherList = Arrays.asList("Apple", "Banana", "Orange");
```

```
ArrayList<String> listFromCollection = new ArrayList<>(anotherList);
```

## Adding Elements

```
ArrayList<String> list = new ArrayList<>();
```

*//Adding elements to the end of the list*

```
list.add("Apple");
```

```
list.add("Banana");
```

*//Adding element at a specific index*

```
list.add(1, "Orange"); // Orange will be added at index 1, Shifting "Banana" to index2.
```

```
ArrayList<Integer> list = new ArrayList<>();
```

```
list.add(1);
```

```
list.add(2);
```

```
list.add(3);
```

```
list.add(0, 0);
```

```
System.out.println(list); op: [0, 1, 2, 3]
```

*//addAll*

```
List.addAll(...);
```

## Accessing Elements

```
ArrayList<String> fruits = new ArrayList<>();
```

```
fruits.add("Apple");
```

```
fruits.add("Banana");
```

```
fruits.add("Orange");
```

*//remove by index*

```
fruits.remove(1); // removes the element at index 1("Apple")
```

*//remove by Value*

```
fruits.remove("Apple"); // removes apple from the list
```

```
ArrayList<Integer> list = new ArrayList<>();  
// add elements  
list.add(1); // 0 index  
list.add(5); // 1  
list.add(30); // 2  
System.out.println(list);  
  
//size check  
System.out.println(list.size());  
  
//get  
System.out.println(list.get(0));  
  
// iterate  
for (int i = 0; i < list.size(); i++) {  
    System.out.println(list.get(i));  
}  
  
for (int e: list) {  
    System.out.println(e);  
}  
  
//contains - exists  
System.out.println(list.contains(5));  
System.out.println(list.contains(50));
```

```
ArrayList<Integer> list = new ArrayList<>();  
// add elements  
list.add(1); // 0 index  
list.add(5); // 1  
list.add(80); // 2  
  
// list.remove(2);  
  
// add at specific index  
list.add(2, 50);  
  
// update specific value at index  
list.set(1, 20);  
for (int e: list) {  
    System.out.println(e);  
}
```

```
ArrayList<Integer> list = new ArrayList();
System.out.println(list.size()); //0
// ArrayList size is 0, but its capacity is 10 (internal array)
// size different capacity different,

// what is capacity ?

// capacity - inside ArrayList array size

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable {

    private static final int DEFAULT_CAPACITY = 10;

    transient Object[] elementData; // array to store elements.
    private int size;

    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        }
    }
}
```

## **\*\* Adding Elements**

***When we add an element to the ArrayList, the following steps occur***

**Check capacity:** Before adding the new element, ArrayList checks if there is enough space in the internal array(**elementData**).if the array is full, it needs to be resized.

**Resize if necessary:**if the internal array is full, the ArrayList will create a new array with larger capacity(usually 1.5 times the current array) and copy the elements from the old array to new array.

**Add the element:**The new element is then added to the internal array at appropriate index, and the size is incremented.



## Resizing the Array

- **Initial capacity:** By default the initial capacity is 10. This means the internal Array can hold 10 elements before it needs to grow.
- **Growth factor:** When the internal is full, a new array is created with 1.5 times the old array.
- **Copying elements:** When resizing occurs, all the elements from the old array are copied to the new array, which is  $O(n)$  operation, where  $n$  is the number of elements in the ArrayList.

# Removing element

- ***Check Bound:*** The ArrayList first checks if the index within the valid range.
- ***Remove the element:*** The element is removed, and all elements to the right of the removed element are shifted one position to the left to fill the gap.
- ***Reduce size:*** The size is decremented by 1.

## Check for elements

```
boolean hasApple = fruits.contains("Apple"); // returns true if Apple is present
```

## Sorting an ArrayList

```
ArrayList<Integer> list = new ArrayList<>();  
list.add(10);  
list.add(3);  
list.add(7);  
list.add(11);  
list.add(5);  
Collections.sort(list); // Sort in natural order  
System.out.println(list);
```

## Access element

```
String fruit = fruits.get(0); // Returns "Apple"
```

## Remove elements

```
//Remove by index
```

```
list.remove(1); // Remove the element at index 1 – (3)
```

```
// Remove by value
```

```
List.remove(Integer.valueOf(3)); // Removes value 3
```

# Sorting list