

Core Java

Arrays

2-D/Multi-dimensional array

```
double[][] arr = new double[2][3];  
  
double[][] arr = new double[][]{ { 1.1, 2.2, 3.3 }, { 4.4, 5.5, 6.6 } };  
  
double[][] arr = { { 1.1, 2.2, 3.3 }, { 4.4, 5.5, 6.6 } };
```

- Internally 2-D arrays are array of 1-D arrays. "arr" is array of 2 elements, in which each element is 1-D array of 3 doubles.
- Individual element is accessed as `arr[i][j]`.

Ragged array

- Ragged array is array of 1-D arrays. Each 1-D array in the ragged array may have different length.

```
int[][] arr = new int[4][];  
arr[0] = new int[] { 11 };  
arr[1] = new int[] { 22, 33 };  
arr[2] = new int[] { 44, 55, 66 };  
arr[3] = new int[] { 77, 88, 99, 110 };  
for(int i=0; i<arr.length; i++) {  
    for(int j=0; j<arr[i].length; j++) {  
        System.out.print(arr[i][j] + ", ");  
    }  
}
```

Variable Arity Method

- Methods with variable number of arguments. These arguments are represented by ... and internally collected into an array.

```
public static int sum(int... arr) {  
    int total = 0;  
    for(int num: arr)  
        total = total + num;  
    return total;  
}  
public static void main(String[] args) {  
    int result1 = sum(10, 20);  
    System.out.println("Result: " + result1);
```

```

        int result2 = sum(11, 22, 33);
        System.out.println("Result: " + result2);
    }
}

```

- If method argument is `Object... args`, it can take variable arguments of any type.
- Pre-defined methods with variable number of arguments.
 - PrintStream class: `PrintStream printf(String format, Object... args);`
 - String class: `static String format(String format, Object... args);`

Method overloading

- Methods with same name and different arguments in same scope - Method overloading.
- Arguments must differ in one of the follows
 - Count

```

static int multiply(int a, int b) {
    return a * b;
}
static int multiply(int a, int b, int c) {
    return a * b * c;
}

```

- Type

```

static int square(int x) {
    return x * x;
}
static double square(double x) {
    return x * x;
}

```

- Order

```

static double divide(int a, double b) {
    return a / b;
}
static double divide(double a, int b) {
    return a / b;
}

```

- Constructors have same name (as of class name) and different arguments. This is referred as "Constructor overloading".
- Note that return type is NOT considered in method overloading. Following code cause error.

```

static int divide(int a, int b) {
    return a / b;
}
static double divide(int a, int b) {
    return (double)a / b;
}

```

```

int result1 = divide(22, 7);
double result2 = divide(22, 7);
// collecting return value is not mandatory
divide(22, 7);

```

Method arguments

- In Java, primitive values are passed by value and objects are passed by reference.
- Pass by reference -- Stores address of the object. Changes done in called method are available in calling method.

```

public static void testMethod(Human h) {
    h.setHeight(5.7);
}
public static void main(String[] args) {
    Human obj = new Human(40, 76.5, 5.5);
    obj.display(); // age=40, weight=76.5, height=5.5
    testMethod(obj);
    obj.display(); // age=40, weight=76.5, height=5.7
}

```

- Pass by value -- Creates copy of the variable. Changes done in called method are not available in calling method.

```

public static void swap(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
public static void main(String[] args) {
    int num1 = 11, num2 = 22;
    System.out.printf("num1=%d, num2=%d\n", num1, num2);
    swap(num1, num2);
    System.out.printf("num1=%d, num2=%d\n", num1, num2);
}

```

- Pass by reference for value/primitive types can be simulated using array.

Command line arguments

- Additional data/information passed to the program while executing it from command line -- Command line arguments.

```
terminal> java pkg.Program Arg1 Arg2 Arg3
```

- These arguments are accessible in Java application as arguments to main().

```
package pkg;
class Program {
    public static void main(String[] args) {
        // ... args[0] = Arg1, args[1] = Arg2, args[2] = Arg3
    }
}
```

Object/Field initializer

- In C++/Java, constructor is used to initialize the fields.
- In Java, field initialization can be done using
 - Field initializer
 - Object initializer
 - Constructor
- Example:

```
class InitializerDemo {
    int num1 = 10;
    int num2;
    int num3;

    InitializerDemo() {
        num3 = 30;
    }
    // ...

    public static void main(String[] args) {
        InitializerDemo obj = new InitializerDemo();
        System.out.printf("num1=%d, num2=%d, num3=%d\n", num1, num2, num3);
    }
}
```

final variables

- In Java, const is reserved word - but not used.
- Java has final keyword instead. It can be used for

- final variables
- final fields
- final methods
- final class
- The final local variables and fields cannot be modified after initialization.
- The final fields must be initialized any of the following.
 - Field initializer
 - Object initializer
 - Constructor
- Example:

```
class FinalDemo {  
    final int num1 = 10;  
    final int num2;  
    final int num3;  
  
    {  
        num2 = 20;  
    }  
  
    FinalDemo() {  
        num3 = 30;  
    }  
    public void display() {  
        System.out.printf("num1=%d, num2=%d, num3=%d\n", num1, num2, num3);  
    }  
  
    public static void main(String[] args) {  
        final int num4 = 40;  
  
        final FinalDemo obj = new FinalDemo();  
        obj.display();  
    }  
}
```

static keyword

- In OOP, static means "shared" i.e. static members belong to the class (not object) and shared by all objects of the class.
- Static members are called as "class members"; whereas non-static members are called as "instance members".
- In Java, static keyword is used for
 - static fields
 - static methods
 - static block
 - static import
- Note that, static local variables cannot be created in Java.

Static fields

- Copies of non-static/instance fields are created one for each object.
- Single copy of the static/class field is created (in method area) and is shared by all objects of the class.
- Can be initialized by static field initializer or static block.
- Accessible in static as well as non-static methods of the class.
- Can be accessed by class name or object name outside the class (if not private). However, accessing via object name is misleading (avoid it).

Static methods

- Methods can be called from outside the class (if not private) using class name or object name. However, accessing via object name is misleading (avoid it).
- When needs to call a method without object, then make it static.
- Since static methods are designed to be called on class name, they do not have "this" reference. Hence, cannot access non-static members in the static method (directly). However, we can access them on an object reference.
- Applications
 - To initialize/access static fields.
 - Helper/utility methods

```
import java.util.Arrays;
// in main()
int[] arr = { 33, 88, 44, 22, 66 };
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
```

- Factory method - to create object of the class

```
import java.util.Calendar;
// in main()
//Calendar obj = new Calendar(); // compiler error
Calendar obj = Calendar.getInstance();
System.out.println(obj);
```

Static field initializer

- Similar to field initializer, static fields can be initialized at declaration.

```
// static field
static double price = 5000.0;
```

static keyword

Static Method

- If we want to access non static members of the class then we should define non static method inside class.

```
class Test{
    private int num1;
    public int getNum1( ){
        return this.num1;
    }
    public void setNum1( int num1 ){
        this.num1 = num1;
    }
}
```

- If we want to access static members of the class then we should define static method inside class.

```
class Test{
    private static int num2;
    public static int getNum2( ){
        return Test.num2;
    }
    public void setNum2( int num2 ){
        Test.num2 = num2;
    }
}
```

Why static method do not get this reference?

- If we call non static method on instance then method gets this reference.
- Static method is designed to call on class name.
- Since static method is not designed to call on instance, it doesn't get this reference.

- Since static method do not get this reference, we can not access non static members inside static method.
- In other words, static method can access only static members of the class directly.
- If we want to access non static members inside static method then we need to use instance of the class.

```
class Program{
    int num1 = 10;
    static int num2 = 20;
    public static void main( String[] args ){
        //System.out.println("Num1 : "+num1); //Compiler error
        Program p = new Program();
        System.out.println("Num1 : "+p.num1); //OK: 10
        System.out.println("Num1 : "+new Program().num1); //OK: 10
    }
}
```

```

        System.out.println("Num2    :    "+num2);    //OK: 20
        System.out.println("Num2    :    "+Program.num2);    //OK:20
    }
}

```

- Inside non static method, we can access static as well as non static members directly.

```

class Test{
    private int num1 = 10;
    private static int num2 = 20;
    public void printRecord( ){
        System.out.println("Num1    :    "+this.num1);    //OK
        System.out.println("Num2    :    "+Test.num2);    //OK
    }
}

```

- Inside method, if there is a need to use this reference then we should declare method non static otherwise we should declare method static.

```

class Math{
    public static int power( int base, int index ){
        int result = 1;
        for( int count = 1; count <= index; ++ count ){
            result = result * base;
        }
        return result;
    }
}
class Program{
    public static void main(String[] args) {
        int result = Math.power(2, 3);
        System.out.println("Result  :  "+result);
    }
}

```

- Method local variable get space once per method call.
- We can declare, method local variable final but we can not declare it static.
 - static variable is also called as class level variable.
 - class level variables should exist at class scope.
 - Hence we can not declare local variable static. But we can declare field static.

```

class Program{
    private static int number;    //OK
    public static void print( ){
        //static int number = 0;    //Not OK
        number = number + 1;
    }
}

```

```

        System.out.println("Number : "+number);
    }
    public static void main(String[] args) {
        Program.print(); //1
        Program.print(); //2
        Program.print(); //3
    }
}

```

Static block

- Like Object/Instance initializer block, a class can have any number of static initialization blocks, and they can appear anywhere in the class body.
- Static initialization blocks are executed in the order their declaration in the class.
- A static block is executed only once when a class is loaded in JVM.
- Example:

```

class Program {
    static int field1 = 10;
    static int field2;
    static int field3;
    static final int field4;

    static {
        // static fields initialization
        field2 = 20;
        field3 = 30;
    }

    static {
        // initialization code
        field4 = 40;
    }
}

```

Static import

- To access static members of a class in the same class, the "ClassName." is optional.
- To access static members of another class, the "ClassName." is mandatory.
- If need to access static members of other class frequently, use "import static" so that we can access static members of other class directly (without ClassName.).

```

import static java.lang.Math.*;
class Program {
    public static double calcArea(double rad) {
        return Math.PI * rad * rad;
    }
}

```

Singleton class

- Design patterns are standard solutions to the well-known problems.
- Singleton is a design pattern.
- It enables access to an object throughout the application source code.
- Singleton class is a class whose single object is created throughout the application.
- To make a singleton class in Java
 - step 1: Write a class with desired fields and methods.
 - step 2: Make constructor(s) private.
 - step 3: Add a private static field to hold instance of the class.
 - step 4: Initialize the field to single object using static field initializer or static block.
 - step 5: Add a public static method to return the object.
- Code:

```
public class Singleton {  
    // fields and methods  
    // since ctor is declared private, object of the class cannot be created  
    // outside the class.  
    private Singleton() {  
        // initialization code  
    }  
    // holds reference of "the" created object.  
    private static Singleton obj;  
    static {  
        // as static block is executed once, only one object is created  
        obj = new Singleton();  
    }  
    // static getter method so that users can access the object  
    public static Singleton getInstance() {  
        return obj;  
    }  
}
```

```
class Program {  
    public static void testMethod() {  
        Singleton obj2 = Singleton.getInstance();  
        // ...  
    }  
  
    public static void main(String[] args) {  
        Singleton obj1 = Singleton.getInstance();  
        // ...  
    }  
}
```

```
    }  
}
```

Association

- If "has-a" relationship exist between the types, then use association.
- To implement association, we should declare instance/collection of inner class as a field inside another class.
- There are two types of associations
 - Composition
 - Aggregation
- Example 1:

```
public class Engine {  
    // ...  
}
```

```
public class Person {  
    private String name;  
    private int age;  
    // ...  
}
```

```
public class Car {  
    private Engine engine;  
    private Person driver;  
    // ...  
}
```

- Example 2:

```
public class Wall {  
    // ...  
}
```

```
public class Person {  
    // ...  
}
```

```
public class Classroom {  
    private Wall[] walls = new Wall[4];  
    private ArrayList<Person> students = new ArrayList<>();  
    // ...  
}
```

Composition

- Represents part-of relation i.e. tight coupling between the objects.
- The inner object is essential part of outer object.
 - Engine is part of Car.
 - Wall is part of ClassRoom

Aggregation

- Represents has-a relation i.e. loose coupling between the objects.
- The inner object can be added, removed, or replaced easily in outer object.
 - Car has a Driver.
 - Company has Employees.

Inheritance

- If "is-a"/"kind-of" relationship exist between the types, then use inheritance.
- Inheritance is process -- generalization to specialization.
- All members of parent class are inherited to the child class.
- Example:
 - Manager is a Employee
 - Mango is a Fruit
 - Triangle is a Shape
- In Java, inheritance is done using extends keyword.

```
class SubClass extends SuperClass {  
    // ...  
}
```

- Java doesn't support multiple implementation inheritance i.e. a class cannot be inherited from multiple super-classes.
- However Java does support multiple interface inheritance i.e. a class can be inherited from multiple super interfaces.

super keyword

- In sub-class, super-class members are referred using "super" keyword.
- Calling super class constructor

- By default, when sub-class object is created, first super-class constructor (param-less) is executed and then sub-class constructor is executed.
- "super" keyword is used to explicitly call super-class constructor.

```
class Person {
    // ...
    public Person(String name, int age) {
        // ...
    }
}
class Student extends Person {
    // ...
    public Student(String name, int age, int roll, double marks) {
        super(name, age); // calls parameterized ctor of super class -- must
be first line only
        // ...
    }
}
```

- Accessing super class members

- Super class members (non-private) are accessible in sub-class directly or using "this" reference. These members can also be accessed using "super" keyword.
- However, if sub-class method signature is same as super-class signature, it hides/shadows method of the super class i.e. super-class method is not directly visible in sub-class.
- The "super" keyword is mandatory for accessing such hidden members of the super-class.

```
class Person {
    // ...
    public String getName() {
        // ...
    }
    public int getAge() {
        // ...
    }
    public void display() {
        // display name and age
    }
}
class Student extends Person {
    // ...
    public void display() {
        System.out.println(this.getName()); // getName() is inherited from
super-class
        System.out.println(getAge()); // getAge() is inherited from super-
class
        super.display(); // Person.display() is hidden due to
Student.display()
        // must use super keyword to call hidden method of super class.
        // display roll and marks
    }
}
```

```
    }  
}
```

Inheritance

Types of inheritances

- Single inheritance

```
class A {  
    // ...  
}  
class B extends A {  
    // ...  
}
```

- Multiple inheritance

```
class A {  
    // ...  
}  
class B {  
    // ...  
}  
class C extends A, B // not allowed in Java  
{  
    // ...  
}
```

```
interface A {  
    // ...  
}  
interface B {  
    // ...  
}  
class C implements A, B // allowed in Java  
{  
    // ...  
}
```

- Hierarchical inheritance

```
class A {  
    // ...  
}
```

```
class B extends A {  
    // ...  
}  
class C extends A {  
    // ...  
}
```

- Multi-level inheritance

```
class A {  
    // ...  
}  
class B extends A {  
    // ...  
}  
class C extends B {  
    // ...  
}
```

- Hybrid inheritance: Any combination of above types

Up-casting & Down-casting

- Up-casting: Assigning sub-class reference to a super-class reference.
 - Sub-class "is a" Super-class, so no explicit casting is required.
 - Using such super-class reference, super-class methods overridden into sub-class can also be called.

```
Employee e = new Employee();  
Person p = e; // up-casting
```

- Down-casting: Assigning super-class reference to sub-class reference.

- Every super-class is not necessarily a sub-class, so explicit casting is required.

```
Person p1 = new Employee();  
Employee e1 = (Employee)p1; // down-casting - okay - Employee reference will  
point to Employee object
```

```
Person p2 = new Person();  
Employee e2 = (Employee)p2; // down-casting - ClassCastException - Employee  
reference will point to Person object
```

Polymorphism

- Poly=Many, Morphism=Forms
- Polymorphism is taking many forms.
- OOP has two types of Polymorphism
 - Compile-time Polymorphism / Static Polymorphism
 - Implemented by "Method overloading".
 - Compiler can identify which method to be called at compile time depending on types of arguments. This is also referred as "Early binding".
 - Run-time Polymorphism / Dynamic Polymorphism
 - Implemented by "Method overriding".
 - The method to be called is decided at runtime depending on type of object. This is also referred as "Late binding" or "Dynamic method dispatch".
 - Process of calling method of sub class using reference of super class is called as dynamic method dispatch. ####Access Modifier
 - private (lowest)
 - default
 - protected
 - public (highest)

Method overriding

- Redefining a super-class method in sub-class with exactly same signature is called as "Method overriding".
- Programmer should override a method in sub-class in one of the following scenarios
 - Super-class has not provided method implementation at all (abstract method).
 - Super-class has provided partial method implementation and sub-class needs additional code. Here sub-class implementation may call super-class method (using super keyword).
 - Sub-class needs different implementation than that of super-class method implementation.
- Rules of method overriding in Java
 - Each method in Java can be overridden unless it is private, static or final.
 - Sub-class method must have same or wider access modifier than super-class method.

```

class SuperClass {
    protected Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
    /*protected or*/ public Number calculate(Integer i, Float f) {
        // ...
    }
}

```

- Arguments of sub-class method must be same as of super-class method. The return-type of sub-class method can be same or sub-class of the super-class's method's return-type. This is called as "covariant" return-type.

```

class SuperClass {
    public Number calculate(Integer i, Float f) {
        // ...
    }
}
class SubClass extends SuperClass {
    // Double is inherited from Numer (i.e. return-type of super-class
    // method)
    public Double calculate(Integer i, Float f) {
        // ...
    }
}

```

- Checked exception list in sub-class method should be same or subset of exception list in super-class method.

```

class SuperClass {
    public void testMethod() throws IOException, SQLException {
        // ...
    }
}
class SubClass extends SuperClass {
    public void testMethod() throws IOException {
        // ...
    }
}

```

- If these rules are not followed, compiler raises error or compiler treats sub-class method as a new method.

```

class A {
    void method1() {
    }
    void method2() {
    }
}

```

```

class B extends A {
    void method1() {
        // overridden
    }
    void method2(int x) {
        // treated as new method
    }
}

```

```
// In main()
A obj = new B();
obj.method1(); // B.method1() -- overridden
obj.method2(); // A.method2() -- method2() is not overridden
```

- Java 5.0 added `@Override` annotation (on sub-class method) informs compiler that programmer is intending to override the method from the super-class.
- `@Override` checks if sub-class method is compatible with corresponding super-class method or not (as per rules). If not compatible, it raise compile time error.
- Note that, `@Override` is not compulsory to override the method. But it is good practice as it improves readability and reduces human errors.