# Core Java

- Array
    - Object arrays
    - 2-D arrays
    - Ragged arrays
    - Variable arity method
- Method overloading
- Method arguments - Call by value/reference
- Command line arguments
- Arrays class -- Helper methods

## 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 accesses 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 mandetory
  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 memebers".
- 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

```java
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.

```java
// static field
static double price = 5000.0;
// static final field -- constant
static final double PI = 3.142;
```