

# Core Java

---

## "this" reference

- "this" is implicit reference variable that is available in every non-static method of class which is used to store reference of current/calling instance.
- Whenever any non-static method is called on any object, that object is internally passed to the method and internally collected in implicit "this" parameter.
- "this" is constant within method i.e. it cannot be assigned to another object or null within the method.
- Using "this" inside method (to access members) is optional. However, it is good practice for readability.
- In a few cases using "this" is necessary.
  - Unhide non-static fields from local fields.

```
double height; // "height" field
void setHeight(double height) { // "height" parameter
    height = height; // ???
}
```

- Constructor chaining
- Access instance of outer object.

## null reference

- In Java, local variables must be initialized before use; otherwise it raise compiler error.

```
int a;
a++; // compiler error
Human obj;
obj.walk(); // compiler error
```

- In Java, reference can be initialized to null (if any specific object is not yet available). However reference must be initialized to appropriate object before its use; otherwise it will raise NullPointerException at runtime.

```
Human h = null; // reference initialized to null
h = new Human(); // reference initialized to the object
h.walk(); // invoke the method on the object
```

```
Human h = null;
h.walk(); // NullPointerException
```

## Constructor chaining

- Constructor chaining is executing a constructor of the class from another constructor (of the same class).

```
Human(int age, double height, double weight) {  
    this.age = age;  
    this.height = height;  
    this.weight = weight;  
}  
Human() {  
    this(0, 1.6, 3.3);  
    // ...  
}
```

- Constructor chaining (if done) must be on the very first line of the constructor.

## OOP concepts

### Abstraction

- Abstraction is getting essential details of the system.
- Abstraction change as per perspective of the user.
- Abstraction represents outer view of the system.
- Abstraction is based on interface/public methods of the class.

### Encapsulation

- Combining information/state and complex logic/operations so that it will be easier to use is called as Encapsulation.
- Fields and methods are bound in a class so that user can create object and invoke methods (without looking into complex implementations).
- Encapsulation represents inner view of the system.
- Encapsulation and abstraction are complementary to each other.

### Information hiding

- Members of class not intended to be visible outside the class can be restricted using access modifiers/specifiers.
- The "private" members can be accessed only within the class; while "public" members are accessible in as well as outside the class.
- Usually fields (data) are "private" and methods (operations) are "public".

## Packages

- Packages makes Java code modular. It does better organization of the code.
- Package is a container that is used to group logically related classes, interfaces, enums, and other packages.

- Package helps developer:
  - To group functionally related types together.
  - To avoid naming clashing/collision/conflict/ambiguity in source code.
  - To control the access to types.
  - To make easier to lookup classes in Java source code/docs.
- Java has many built-in packages.
  - `java.lang` \* --> `Integer`, `System`, `String`, ...
  - `java.util` --> `Scanner`, `ArrayList`, `Date`, ...
  - `java.io` --> `FileInputStream`, `FileOutputStream`, ...
  - `java.sql` --> `Connection`, `Statement`, `ResultSet`, ...
  - `java.util.function` --> `Predicate`, `Consumer`, ...
  - `javax.servlet.http` --> `HttpServlet`, ...
- Package Syntax
  - To define a type inside package, it is mandatory write package declaration statement inside .java file.
  - Package declaration statement must be first statement in .java file.
  - Types inside package called as packaged types; while others (in default package) are unpackaged types.
  - Any type can be member of single package only.
- It is standard practice to have multi-level packages (instead of single level). Typically package name is module name, dept/project name, website name in reverse order.

```
package com.sunbeaminfo.modular.corejava;
```

```
package com.sunbeaminfo.dac;
```

- Packages can be used to avoid name clashing. In other words, two packages can have classes/types with same name.

```
package com.sunbeam.tree;

class Node {
    // ...
}

public class Tree {
    // ...
}
```

```
package com.sunbeam.list;

class Node {
    // ...
}

public class List {
    // ...
}
```

## Access modifiers (for types)

- default: When no specifier is mentioned.
  - The types are accessible in current package only.
- public: When "public" specifier is mentioned.
  - The types are accessible in current package as well as outside the package (using import).

## Access modifiers (for type members)

- private: When "private" specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
- default: When no specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
  - The members are accessible in all classes in same package (obj.member OR ClassName.member).
- protected: When "protected" specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
  - The members are accessible in all classes in same package including its sub-classes (super.member, obj.member OR ClassName.member).
  - The members are accessible in sub classes outside the package including its sub-classes (super.member).
- public: When "public" specifier is mentioned before the member field/method.
  - The members are accessible in current class (member OR this.member).
  - The members are accessible in all other classes (super.member, obj.member OR ClassName.member).
- Scopes
  - private (lowest)
  - default
  - protected
  - public (highest)

	default	private	protected	public
same class	yes	yes	yes	yes
same package subclass	yes	no	yes	yes
same package non-subclass	yes	no	yes	yes
different package subclass	no	no	yes	yes
different package non-subclass	no	no	no	yes

- 

## Array

- Array is collection of similar data elements. Each element is accessible using indexes (0 to n-1).
- In Java, array is non-primitive/reference type i.e. its object is created using new operator (on heap).
- Array size is fixed - given while creating array object. It cannot be modified later.
- Java array object holds its length and data elements.
- The array of primitive type holds values (0 if uninitialized) and array of non-primitive type holds references (null if uninitialized).
- Array of primitive types

```
int[] arr1 = new int[5];

int[] arr2 = new int[5] { 11, 22, 33 };
// error

int[] arr3 = new int[] { 11, 22, 33, 44, 55 };

int[] arr4 = { 11, 22, 33, 44 };
```

- Array of non-primitive types

```
Human[] arr5 = new Human[5];

Human[] arr6 = new Human[] {h1, h2, h3};
// h1, h2, h3 are Human references

Human[] arr7 = new Human[] {
    new Human(...),
    new Human(...),
    new Human(...)
};
```

- In Java, checking array bounds is responsibility of JVM. When invalid index is accessed, `ArrayIndexOutOfBoundsException` is thrown.
- Array types are
  - 1-D array
  - 2-D/Multi-dimensional array
  - Ragged array

## 1-D array

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

int[] arr = new int[4] { 10, 20, 30, 40 };
// error

int[] arr = new int[] { 11, 22, 33, 44, 55 };

int[] arr = { 11, 22, 33, 44 };

Human[] arr = new Human[5];

Human[] arr = new Human[] {h1, h2, h3};
```

- Individual element is accesses as `arr[i]`.

## 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 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
    }
}
```