OOP_First
Created by
github.com/farzadafi

- **What is OOP?**
  - OOP stands for Object-Oriented Programming.
  - Procedural programming
    - is about writing procedures or methods that perform operations on the data,
  - while object-oriented programming is about creating objects that contain both data and methods.
  - OOP is Closer to the problem-solving space to real life
  - Object-oriented programming has several advantages over procedural programming
    - OOP is faster and easier to execute
    - OOP provides a clear structure for the programs
    - OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
    - OOP makes it possible to create full reusable applications with less code and shorter development time
  - main concepts of object-oriented programming
    - Inheritance
    - Encapsulation
    - Abstraction
    - Polymorphism
- **Class**
  - class and object
    - problem
      - until now, we understand something about Strings, Integers, char, and ..., but please think and answer this question, if you write a program like an online market, and this app has some users that enter your market and buy something, how to define this user? with Integer? with String? with what???
        - To solve this problem, we define class :) and use from string, integer, bool, for create a person for example, or other object :)
          - Subtopic 1
    - class is a blueprint or template for creating objects. It defines the properties and behaviors that objects of that class will have.
    - Everything in Java is associated with classes and objects, along with its attributes and methods.
    - define
      - To create a class, use the keyword class
      - a class should always start with an uppercase first letter, and that the name of the java file should match the class name.

```java
public class Main {
    //todo define field or method
}
```

- person class

```java
class Person {
```

- scanner class

```java
public final class Scanner
```

- object
  - is an instance of a class that encapsulates data (known as fields or attributes) and behavior (known as methods or operations).
  - Objects are the fundamental building blocks of OOP, and they are used to represent real-world entities or concepts in software.
  - In Java, objects are created using the new operator followed by the name of the class and any arguments needed by the class constructor.
  - define for person class

```java
Person farzad = new Person();
Person naser = new Person();
Person alireza = new Person();
```

- define for scanner class

```java
Scanner scanner = new Scanner(System.in);
```

- class VS object
  - a class is a template for objects, and an object is an instance of a class.
  - car is class
    volvo, mazda, 206 are object from car class
  - fruit is class
    apple, mango are object form fruit class

- class Attribute
  - are variables that hold the state or data of an object.
  - They are defined within a class
  - also known as fields
  - person class

```java
class Person {
    String firstname = "farzad"; // attribute
```

- Accessing Attributes
  - You can access attributes by creating an object of the class, and by using the dot syntax (.)

```java
Person farzad = new Person();
System.out.println(farzad.firstname); // output is farzad
```

- Modify Attributes
  - You can also modify attribute values

```java
Person farzad = new Person();
farzad.firstname = "amin";
System.out.println(farzad.firstname); // output is amin
```

- Multiple Objects
  - If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other

```java
Person farzad = new Person();
Person naser = new Person();
farzad.firstname = "amin";
System.out.println(farzad.firstname); // output is amin
System.out.println(naser.firstname); //output is farzad
```

- Multiple Attributes
  - You can specify as many attributes as you want

```java
class Person {
    String firstname = "farzad"; // attribute
    String lastname; // attribute
    Integer age; //attribute
```

- method
  - A method is a block of code which only runs when it is called.
  - You can pass data, known as parameters, into a method.
  - Methods are used to perform certain actions, and they are also known as functions.
  - Why use methods?
    - To reuse code: define the code once, and use it many times.
  - A method must be declared within a class.
  - It is defined with the name of the method, followed by parentheses ()
  - Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:
  - define

```java
void eat() {
    System.out.println("Eat something :)");
}
```

- eat() is the name of the method
- void means that this method does not have a return value. You will learn more latter about this.
- Call a Method
  - To call a method in Java,first create a object from that class and write the object name and then . and then method's name followed by two parentheses () and a semicolon;

```java
Person person = new Person();
person.eat(); //output is Eat something
```

- A method can also be called multiple times:

```java
Person person = new Person();
person.eat(); //output is Eat something
person.eat(); //output is Eat something
person.eat(); //output is Eat something
```

- Parameters and Arguments
  - Information can be passed to methods as parameter.
  - Parameters act as variables inside the method.
  - Parameters are specified after the method name, inside the parentheses.
  - You can add as many parameters as you want, just separate them with a comma.
  - define

```java
void eat(String firstname) {
    System.out.println(firstname + "eat something :)");
```

- call

```java
Person person = new Person();
person.eat( firstname: "farzad"); //output is farzad Eat something
```

- When a parameter is passed to the method, it is called an argument. So, from the example above: firstname is a parameter, while farzad, is argument.
- Multiple Parameters
  - You can have as many parameters as you like:
    - defind

```java
void eat(String firstname, String foodName) {
    System.out.println(firstname + "eat " + foodName);
}
```

- call

```java
Person person = new Person();
person.eat( firstname: "farzad", foodName: "milk"); //output is farzad Eat milk
```

- Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.
- Create a Method
  - Return Values
    - The void keyword, used in the examples above, indicates that the method should not return a value.
    - If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method
    - for example some times you want to create a method that accept two variable and calcualte sum of them and print this value inside of this method and not save this result to other variable like this

```java
void sum(int number1,int number2){
    System.out.println("result is " + (number1 + number2));
}

public static void main(String[] args) {
    Person person = new Person();
    person.sum(1, 2); //output is result is 3
}
```

- and the another time you want sum method get you result, and not print result inside of body on method, and you want to use that result like this:

```java
int sum(int number1,int number2){
    int result = number1 + number2;
    return result;
}

public static void main(String[] args) {
    Person person = new Person();
    int result = person.sum(1, 2);
    System.out.println(result + 10); //output is 13
}
```

- this sum method what to return an int, for this reason we have to write for return type int, if return a String, we have to write String in return type

```java
String sum(int number1,int number2){
    int result = number1 + number2;
    return String.valueOf(result);
}

public static void main(String[] args) {
    Person person = new Person();
    String result = person.sum(1, 2);
    System.out.println(result + 10); //output is 13
}
```

- if return type different kind return type you write on signature method, you will get comile time error :)

```java
String sum(int number1,int number2){
    int result = number1 + number2;
    return true;
}
```

- you can write any code in a mehtod, pay attention, a method must be litte, little, and little :)
- Method Overloading
  - With method overloading, multiple methods can have the same name with different parameters:

```java
int sum(int number1, int number2) {
    int result = number1 + number2;
    return result;
}

int sum(int number1, int number2, int number3){
    int result = number1 + number2 + number3;
    return result;
}

public static void main(String[] args) {
    Person person = new Person();
    int result = person.sum(1, 2);
    int result1 = person.sum(1, 2, 3);
}
```

- Multiple methods can have the same name as long as the number and/or type of parameters are different.

```java
int sum(int number1, int number2) {
    int result = number1 + number2;
    return result;
}

String sum(int number1, int number2){
    int result = number1 + number2;
    return String.valueOf(result);
}

public static void main(String[] args) {
    Person person = new Person();
    int result = person.sum( number1: 1, number2: 2);
    String result1 = person.sum( number1: 1, number2: 2);
}
```

- Recursion
  - Recursion is the technique of making a function call itself.
  - This technique provides a way to break complicated problems down into simple problems which are easier to solve.
  - Recursion may be a bit difficult to understand.
  - The best way to figure out how it works is to experiment with it.
  - example

```java
int sum(int number) {
    if (number > 0) {
        return number + sum( number: number - 1);
    } else {
        return 0;
    }
}
```

- **Java Scope**
  - In Java, variables are only accessible inside the region they are created. This is called scope.
  - Method Scope
    - Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared

```java
int sum(int number1, int number2) {
    int variableTest;
    int result = number1 + number2;
    return result;
}

String sum(int number1, int number2, int number3){
    variableTest = 10; //this variable defines in scope above sum method
    // because we cant access it in this scope :)
    int result = number1 + number2 + number3;
    return String.valueOf(result);
}
```

- Block Scope
  - A block of code refers to all of the code between curly braces {}
    - Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared
    - A block of code may exist on its own

```java
int sum(int number1, int number2) {
    {
        int number = 10;
    }
    System.out.println(number);
    return number1 + number2;
}
```

- or it can belong to an if, while or for statement.

```java
for (int i = 0; i < 10; i++) {
    int number = i + 10;
}
System.out.println(number);
// we cant access this number, because
// defines no for loop scope
```

```java
if(true){
    int number = 10;
}
System.out.println(number);
```

- **continue Class**
  - Constructors
    - A constructor in Java is a special method that is used to initialize objects.
    - The constructor is called when an object of a class is created.
    - define
      - Note that the constructor name must match the class name,
      - it cannot have a return type (like void).
      - Also note that the constructor is called when the object is created.
      - please define it with Public keyword, just write and not attention to it, after will speack about this public.
      - example

```java
class Person {
    String firstname;
    String lastname;

    public Person() { //constructor
        firstname = "farzad";
        lastname = "afshar";
    }
```

- All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.
- Parameters
  - Constructors can also take parameters, which is used to initialize attributes.
  - You can have as many parameters as you want
  - exmample

```java
public Person(String name1, String name2) { //constructor
    firstname = name1;
    lastname = name2;
}
```

- this keyword
  - The this keyword refers to the current object in a method or constructor.
  - The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).

```java
class Person {
    String firstname;
    String lastname;

    public Person(String firstname, String lastname) { //constructor
        firstname = firstname; //shadowing occur here :|
        lastname = lastname; //shadowing occur here :|
    }

    public static void main(String[] args) {
        Person person = new Person("farzad", "afshar");
        System.out.println(person.firstname); //null
        System.out.println(person.lastname); //null
    }
}
```

with this please

- this can also be used to
  - invoke current class constructor
  - invoke current class method

```java
void testThisInMethod() {
    System.out.println(this.firstname + " " + this.lastname);
}
```

- Return the current class object

```java
Person testThisInReturnClassObject() {
    this.firstname = "farzad";
    this.lastname = "afshar";
    return this;
}
```

- **Inheritance**
  - In Java, it is possible to inherit attributes and methods from one class to another.
  - We group the "inheritance concept" into two categories:
    - superclass (parent) - the class being inherited from
    - subclass (child) - the class that inherits from another class
  - To inherit from a class, use the extends keyword.
  - real senario
    - please consider a situation, you want to create a college application, and in this app, you have some users like master, student, and clerk, these entity has some same attributes like full name, age, and national code, you write these same attribute in all classes? if we don't use from inheritance answer is yes, but with inheritance this work is awful :), you create a class with a name's person and write all the same attributes to it, this class is a superclass, and master, student, and clerk extend from person class, and these class are subclass this is good :)
      - you can access to field

```java
class Person {
    String firstname;
    String lastname;
}

class Clerk extends Person{

    // you can access to all fields
    // define on parent class
    String fullName() {
        return firstname + lastname;
    }
}
```

- you can access to method

```java
class Person {
    String firstname;
    String lastname;

    String fullname() {
        return firstname + lastname;
    }
}

class Clerk extends Person{

    String fullNameTest() {
        return fullname();
    }
}
```

- Why And When To Use "Inheritance"?
  - - It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.
- override
  - In Java, the override keyword is used to indicate that a method in a subclass is intended to override a method with the same name and signature in its superclass.
  - When a method is overridden in a subclass, the subclass provides its own implementation of the method, which is used instead of the implementation in the superclass when the method is called on an instance of the subclass.
  - The @Override annotation can be used before the method declaration to indicate that the method is intended to override a superclass method.
  - This can help prevent mistakes where a developer accidentally creates a new method with a similar name, but different signature, instead of overriding the intended method.
  - law of override
    - The method in the subclass must have the same name as the method in the superclass.
    - The method in the subclass must have the same parameter types as the method in the superclass.
    - The method in the subclass must have the same return type (or a covariant return type) as the method in the superclass.
    - The access level for the overridden method in the subclass cannot be more restrictive than the access level in the superclass. For example, if the overridden method in the superclass is public, the access level for the overridden method in the subclass must also be public or less restrictive (e.g., protected or package-private).
    - The overridden method in the subclass cannot throw checked exceptions that are not compatible with the exceptions thrown by the method in the superclass. In other words, the overridden method can only throw the same exceptions or subclasses of those thrown by the method in the superclass.
    - If the method in the superclass is marked as final, it cannot be overridden in the subclass.
    - If the method in the superclass is marked as static, it cannot be overridden in the subclass. Instead, the subclass can define a static method with the same name, but this is not considered method overriding.
  - When overriding a method, it's recommended to use the @Override annotation to indicate the intention to override a superclass method. This can help catch errors where a method is intended to be overridden, but has a slightly different signature due to a typo or other mistake.
- abstract
  - an abstract class is a class that cannot be instantiated directly, and is often used as a base class for other classes.
  - An abstract class can contain abstract methods, which are declared but not implemented in the abstract class. Subclasses of the abstract class must provide implementations for these abstract methods.
  - a real scenario
    - please consider a situation you must write a college application, in this app you have some clerck and some master, but the method of calculating the salary of the clerk and salary of the master is different, Based on this structure, you create an Employee class and add a method with name calculateSalary(), this method should be an abstract method, and then create two class with name Clerck and Master that should be overriden calculateSalary() method and write own implementation

```java
abstract class Employee {
    String fullname;
    int salary;

    abstract int calculateSalary(int value);
}

class Clerk extends Employee {

    @Override
    int calculateSalary(int overtime) {
        return salary + overtime;
    }
}

class Master extends Employee {

    @Override
    int calculateSalary(int unit) {
        return salary + (unit * 200_000);
    }
}
```

- **Packages**
  - A package in Java is used to group related classes.
  - Think of it as a folder in a file directory.
  - We use packages to avoid name conflicts, and to write a better maintainable code.
  - Packages are divided into two categories:
    - Built-in Packages (packages from the Java API)
      - The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
        - The library contains components for managing input, database programming, and much much more.
        - The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.
      - The library is divided into packages and classes.
      - Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
      - To use a class or a package from the library, you need to use the import keyword:
        - import package.name.Class;   // Import a single class
        - import package.name.*;   // Import the whole package
      - Import a Class
        - If you find a class you want to use, for example, the Scanner class, which is used to get user input, write the following code:
        - import java.util.Scanner;
        - In the example above, java.util is a package, while Scanner is a class of the java.util package.
        - To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation.
      - Import a Package
        - There are many packages to choose from.
        - In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.
        - To import a whole package, end the sentence with an asterisk sign (*)
        - The following example will import ALL the classes in the java.util package:
          - import java.util.*;
    - User-defined Packages (create your own packages)
      - create new package in intellij
      - the convention for name of a pakcage is should start with lowercase and write in camelCase convention :)
- **Java Modifiers**
  - By now, you are quite familiar with the public keyword in all main method :)
    - public void main(String[] args) {}
  - The public keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.
  - We divide modifiers into two groups:
    - Access Modifiers
      - controls the access level
        - For classes, you can use either
          - public
            - The class is accessible by any other class
          - default
            - This is used when you don't specify a modifier.
            - The class is only accessible by classes in the same package.
        - For attributes, methods and constructors
          - public
            - The code is accessible for all classes
          - private
            - The code is only accessible within the declared class
            - just in that class, you define code, and you can access it
          - protected
            - the code is accessible in the same package and subclasses.
          - default
            - The code is only accessible in the same package.
            - This is used when you don't specify a modifier.
    - Non-Access Modifiers
      - do not control access level, but provides other functionality
        - For classes, you can use either
          - final
            - The class cannot be inherited by other classes
          - abstract
            - The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.
        - For attributes and methods, you can use the one of the following
          - final
            - Attributes and methods cannot be overridden/modified
          - static
            - Attributes and methods belongs to the class, rather than an object
          - abstract
            - Can only be used in an abstract class, and can only be used on methods.
            - The method does not have a body, for example abstract void run();.
            - The body is provided by the subclass (inherited from).
          - transient
            - Attributes and methods are skipped when serializing the object containing them
          - synchronized
            - Methods can only be accessed by one thread at a time
          - volatile
            - The value of an attribute is not cached thread-locally, and is always read from the " main memory"
- **final keyword**
  - the final keyword can be used with variables, methods, and classes to indicate that they cannot be modified once they are initialized.
    - variable
      - When used with variables, the final keyword creates a constant, which means the value cannot be changed after it has been initialized.
      - This is often used to define values that should not be modified, such as mathematical constants, or to ensure that a variable is initialized only once.
      - final int max = 100;
    - method
      - the final keyword indicates that the method cannot be overridden by a subclass.
      - This is useful when a method should not be changed by a subclass, for example, because it is a crucial part of a class's functionality.
      - final void print() {}
    - class
      - When used with classes, the final keyword indicates that the class cannot be subclassed.
      - This can be useful when a class should not be extended, for example, because it is a complete and final implementation of a certain functionality.
- **static keyword**
  - the static keyword is used to create a static variable or method that belongs to the class itself, rather than to individual instances of the class.
  - When a variable or method is declared as static, it means that there will only be one copy of that variable or method in memory, regardless of how many instances of the class are created.
    - This is because the variable or method is associated with the class itself, rather than with any specific instance of the class.
  - Some common uses of the static keyword in Java include
    - Creating static constants
      - If you have a constant value that is used throughout your program, you can declare it as a static final variable in a class. This makes it easy to access the constant from anywhere in the program without having to instantiate an object of the class.
        - convention for name of final statifc variable is upperCase and delemiter of word is underline(_) like:
          - static final FIRST_NAME = "farzad";
    - Creating static methods
      - If you have a method that performs some operation that is not specific to any particular instance of a class, you can declare it as a static method. This allows you to call the method directly on the class, without having to create an instance of the class first.
        - when you want to access to a static variable or a method, you dont need to create a object, just write class name and then . and then write name of that fields or method name like this

```java
class Person {
    static String firstname;

    public static void main(String[] args) {
        System.out.println(Person.firstname);
    }
}
```

fields

```java
class Person {
    static String firstname;

    static String getFirstName() {
        return firstname;
    }

    public static void main(String[] args) {
        System.out.println(Person.getFirstName());
    }
}
```

method

- Creating static blocks
  - If you need to perform some initialization that is common to all instances of a class, you can use a static block. A static block is executed only once, when the class is loaded into memory, and it can be used to initialize static variables or perform other setup tasks.

```java
class Person {
    static String firstname;
    static String lastname;
    static String fullname;

    static { //this is static block
        firstname = "Farzad";
        lastname = "Afshar";
        String str = firstname + lastname;
        fullname = str;
    }
```

- **Encapsulation**
  - The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users.
  - To achieve this, you must
    - declare class variables/attributes as private
    - provide public get and set methods to access and update the value of a private variable
  - Getter and Setter
    - You learned that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public get and set methods.
    - The get method returns the variable value
    - the set method sets the value.
    - Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case

```java
class Employee {
    private String fullname;

    // Getter
    public String getFullname() {
        return fullname;
    }

    //Setter
    public void setFullname(String fullname) {
        this.fullname = fullname;
    }
}
```

- The get method returns the value of the variable fullname.
- fThe set method takes a parameter (new fullname) and assigns it to the fullname variable. The this keyword is used to refer to the current object.
- However, as the name variable is declared as private, we cannot access it from outside this class

```java
Employee employee = new Employee();
employee.fullname = "farzad"; //fullname is private
```

- Instead, we use the getFullname() and setFullname() methods to access and update the variable:

```java
Employee employee = new Employee();
employee.setFullname("farzad");
final String fullname = employee.getFullname();
```

- Why Encapsulation?
  - Better control of class attributes and methods
  - Class attributes can be made
    - read-only (if you only use the get method)
    - write-only (if you only use the set method)
    - Flexible: the programmer can change one part of the code without affecting other parts
    - Increased security of data
      - please consider a situation you have a variable with a grade name, this grade is between 0 and 20 number, and you must prevent the user from assigning a number less than 0 or more than 20, this is controlled in the setter method with an if condition :)
- **Reference Data Types in Java**
  - Java provides two types of data types
    - primitive
      - The primitive data types are predefined in Java that serves as a fundamental building block
    - reference
      - the reference data type refers to where data is stored.
      - In Java, non-primitive data types are known as reference types.
      - In other words, a variable of class type is called reference data type.
      - It contains the address (or reference) of dynamically created objects.
      - For example, if Demo is a class and we have created its object d, then the variable d is known as a reference type.
      - It refers to objects.
      - It is not pre-defined
      - It is created by the programmer if required.
      - The reference types hold the references of objects.
      - All reference types are a subclass of type java.lang.Object. It provides access to the objects stored in the memory.

- reference data types
  - class
    - It is a set of instructions. It describes the content of the object.
  - Array
    - It provides the fixed-size data structure that stores the elements of the same type.
  - Interface
    - It is implemented by Java classes.
  - Annotations
    - It provides a way to associate metadata with program elements.
  - Enumeration
    - It is a special kind of class that is type-safe. Each element inside the enum is an instance of that enum.
- Reference vs Primitive
  - arrange with first primitive second refrences
    - Primitive
    - Refrences
  - 1
    - Examples of primitive data types are int, float, double, Boolean, long, etc.
    - Examples of reference data types are class, Arrays, String, Interface, etc.
  - 2
    - Example: int num=78;
    - Example: Demo d1;
  - 3
    - It is pre-defined in Java.
    - It is not pre-defined except the String.
  - 4
    - All primitive type begins with a lowercase letter.
    - All reference type begins with Uppercase letter.
  - 5
    - The size of a primitive type depends on the data type.
    - Non-primitive types have all the same size.
  - 6
    - We cannot invoke the method with a primitive type.
    - It is used to invoke or call methods.
  - 7
    - It cannot be null. It always has value.
    - It can be null.
  - 8
    - Its size depends on the data type.
    - JVM allocates 8 bytes for each reference variable, by default.
- Memory Allocation
  - In Java, the new keyword is used to create an instance of the class.
  - In other words, it instantiates a class by allocating memory for a new object and returning a reference to that memory.
  - Objects occupy memory in the Java heap space.
  - We can also use the new keyword to create the array object.
- Garbage Collection
  - garbage means unreferenced objects.
  - Garbage Collection is process of reclaiming the runtime unused memory automatically.
  - In other words, it is a way to destroy the unused objects.
  - To do so, we were using free() function in C language and delete() in C++.
    - But, in java it is performed automatically. So, java provides better memory management.
  - Advantage
    - It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
    - It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.
  - How can an object be unreferenced
    - By nulling the reference
      - Employee e=new Employee();
        e=null;
    - By assigning a reference to another
      - Employee e1=new Employee();
        Employee e2=new Employee();
        e1=e2;//now the first object referred by e1 is available for garbage collection
    - By anonymous object etc.
      - new Employee();
  - you dont need to use these :)
    - finalize() method
      - The finalize() method is invoked each time before the object is garbage collected.
      - This method can be used to perform cleanup processing. This method is defined in Object class
    - gc() method
      - The gc() method is used to invoke the garbage collector to perform cleanup processing.
      - The gc() is found in System and Runtime classes
      - Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.
        - System.gc();
- Copying Reference Type
  - There are two possibilities when we copy reference types, either a copy of the reference to an object is made or an actual copy (creating a new copy) of the object is made.
  - In the following example, we have assigned a reference to the object. If we made any changes in the object, it will also reflect the reference and vice-versa.

```java
Person person = new Person( firstname: "farzad");
Person person1 = person;

System.out.println(person.getFirstname());
person1.setFirstname("naser");
System.out.println(person.getFirstname());
```

- Comparing Reference Type
  - We can also compare the reference types in Java.
  - Java provides two ways to compare reference types:
    - By using the equal (==) operator
      - It compares the memory locations of the objects.
      - If the memory address (reference) of both objects is the same, the objects are equal.

```java
String str = new String( original: "farzad");
String ltr = str;
if(str == ltr)
    System.out.println("True"); //output is true
```

- Note that it does not compare the contents of the object.

```java
String str = new String( original: "farzad");
String ltr = new String( original: "farzad");
if(str != ltr)
    System.out.println("True"); //output is true
```

- By using the String.equals() Method
  - The method belongs to the String class.
  - It overrides the equals() method of the Object class.
  - example

```java
String str = new String( original: "farzad");
String ltr = new String( original: "farzad");
if(str.equals(ltr))
    System.out.println("True"); //output is true
```

- **Call by value and Call by reference**
  - call by value
    - Call by Value means calling a method with a parameter as value.
    - Through this, the argument value is passed to the parameter.
    - the modification done to the parameter passed does not reflect in the caller's scope
    - example

```java
class Tester {
    public static void main(String[] args) {
        int number = 30;
        increaseMethod(number);
        System.out.println("after increase outside method " + number);
    }

    public static void increaseMethod(int number) {
        System.out.println("inside method before increase " + number);
        ++number;
        System.out.println("inside method after increase" + number);
    }
}
```

```
inside method before increase 30
inside method after increase31
after increase outside method 30
```

- call by refrences
  - Call by Reference means calling a method with a parameter as a reference.
  - Through this, the argument reference is passed to the parameter.
  - the modification done to the parameter passed are persistent and changes are reflected in the caller's scope.
  - example

```java
class Tester {
    public static void main(String[] args) {
        Person person = new Person( firstname: "farzad");
        changeName(person);
        System.out.println("outside method after call " + person.getFirstname());
    }

    public static void changeName(Person person) {
        System.out.println("inside method before change " + person.getFirstname());
        person.setFirstname("naser");
        System.out.println("inside method after change " + person.getFirstname());
    }
}
```

```
inside method before change farzad
inside method after change naser
outside method after call naser
```

- fis Java "pass-by-reference" or "pass-by-value"?
  - pass-by-value
    - Pass-by-value means that the value of a variable is passed to a function/method.
  - pass-by-reference
    - Pass-by-reference means that a reference to that variable is passed to the function.
      - gives the function a way to change the contents of the variable.
  - By those definitions, Java is always pass-by-value.
  - Unfortunately, when we deal with variables holding objects we are really dealing with object-handles called references which are passed-by-value as well.
    - This terminology and semantics easily confuse many beginners.

```java
class Tester {
    public static void main(String[] args) {
        Person person = new Person( firstname: "farzad");
        changeName(person);
        System.out.println("outside method after call " + person.getFirstname());
    }

    public static void changeName(Person person) {
        System.out.println("inside method before change " + person.getFirstname());
        person = new Person( firstname: "ali");
        System.out.println("inside method after change " + person.getFirstname());
    }
```

- **Inner Classes**
  - In Java, it is also possible to nest classes (a class within a class).
  - The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
  - advantages
    - Encapsulation
      - Inner classes can be used to encapsulate related functionality within a single class, making the code more modular and easier to maintain.
    - Access to private members
      - Inner classes have access to the private members of the enclosing class, which can be useful for implementing complex algorithms and data structures.
    - Improved code organization
      - By grouping related classes together, inner classes can improve the organization and readability of your code.
    - Implementation of callbacks
      - Anonymous inner classes can be used to implement callbacks and event handlers, making it easier to write event-driven programs.
  - To access the inner class, create an object of the outer class, and then create an object of the inner class:

```java
class Person {
    public String firstname;

    class Clerk {
        public String lastname;
    }
}

class Tester {
    public static void main(String[] args) {
        Person person = new Person();
        person.firstname = "farzad";
        Person.Clerk clerk = person.new Clerk();
        clerk.lastname = "afshar";
        System.out.println(person.firstname + " " + clerk.lastname);
    }
}
```

- Private Inner Class
  - Unlike a "regular" class, an inner class can be private or protected
  - If you don't want outside objects to access the inner class, declare the class as private:
  - advantage of private inner class
    - Security
      - Private inner classes can be used to implement security features by hiding sensitive information or functionality from unauthorized access.
  - example

```java
class Person {
    private String firstname = "farzad";

    public void printFirstname() {
        Clerk clerk = new Clerk();
        clerk.Print();
    }

    private class Clerk {
        public void Print() {
            System.out.println("firstname is " + firstname);
        }
    }
}

class Tester {
    public static void main(String[] args) {
        Person person = new Person();
        person.printFirstname();
    }
```

- Static Inner Class
  - An inner class can also be static, which means that you can access it without creating an object of the outer class:
  - example

```java
class Person {
    public String firstname;

    static class Clerk {
        public String lastname;

    }
}

class Tester {
    public static void main(String[] args) {
        Person person = new Person();
        person.firstname = "farzad";
        Person.Clerk clerk = new Person.Clerk();
        clerk.lastname = "afshar";
        System.out.println(person.firstname + " " + clerk.lastname);
    }
}
```

- just like static attributes and methods, a static inner class does not have access to members of the outer class.

Presented with xmind