OOPs with Java
Static & final

References:

- Java: The Complete Reference, Book & other online resources

Access Control

- There are four types of Java access modifiers:
- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access control : more examples

```
class A{
private int data=40;
private void msg()
{
System.out.println("Hello java");
}
}

public class Simple{
public static void main(String args[]){
  A obj=new A();
  System.out.println(obj.data);//Compile Time Error
  obj.msg();//Compile Time Error
  }
}
```

```java
class A{
private A()
{
}//private constructor
void msg()
{
System.out.println("Hello java");}
}
public class Simple
{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

```
class Data {
    private String name;

    // getter method
    public String getName() {
        return this.name;
    }
    // setter method
    public void setName(String name) {
        this.name= name;
    }
}
public class Main {
    public static void main(String[] main){
        Data d = new Data();

        // access the private variable using the getter and setter
        d.setName("accessing the private variable:name");
        System.out.println(d.getName());
    }
}
```

```
class Test {
 // private variables
 private int age;
 private String name;
 // initialize age
 public void setAge(int age) {
   this.age = age;
 }
 // initialize name
 public void setName(String name) {
   this.name = name;
 }
 // access age
 public int getAge() {
   return this.age;
 }
 // access name
 public String getName() {
   return this.name;
 }
}
```

```
class Main {
  public static void main(String[] args) {
    // create an object of Test
    Test test = new Test();
    // set value of private variables
    test.setAge(24);
    test.setName(“xyz");
    // get value of private variables
    System.out.println("Age: " + test.getAge());
    System.out.println("Name: " + test.getName());
  }
}
```

o/p

- Age: 24
- Name: xyz

Static keyword

Static keyword

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static.**
- **When a member is declared static, it can be accessed** before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static.**

- When objects of its class are declared, no copy of a **static variable is made. Instead, all instances of the class** share the same **static variable.**
- Methods declared as **static have several restrictions:**
  - They can only directly call other **static methods.**
  - They can only directly access **static data.**
  - They cannot refer to **this or super in any way.**

```
class SimpleStaticExample
{
    // This is a static method
    static void myMethod()
    {
        System.out.println("myMethod");
    }

    public static void main(String[] args)
    {
          /* You can see that we are calling this
           * method without creating any object.
           */
           myMethod();
    }
}
```

If you need to do computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded.

```
// Demonstrate static variables, methods,
and blocks.
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block
initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(42);
  }
}
```

o/p

```
Static block initialized.
   x = 42
   a = 3
   b = 12
```

if you wish to call a **static method from outside** its class, you can do so using the following general form:

classname.method( )

Here, *classname is the name of the class in which the* ***static method is declared.***

A **static variable can be accessed in the same way—by use of the dot** operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

```
class StaticDemo {
  static int a = 42;
  static int b = 99;

  static void callme() {
    System.out.println("a = " + a);
  }
}

class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme();
    System.out.println("b = " +
StaticDemo.b);
  }
}
```

o/p

```
a = 42
b = 99
```

Final keyword

- A field can be declared as **final. Doing so prevents its contents from being modified,** making it, essentially, a constant
- This means that you must initialize a **final field when** it is declared. You can do this in one of two ways:
  - First, you can give it a value when it is declared.
  - Second, you can assign it a value within a constructor.
- n addition to fields, both method parameters and local variables can be declared **final.**
- It is a common coding convention to choose all uppercase identifiers for **final fields**
- **Example**
  - `final int FILE_NEW = 1;`

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

Final Variable $\longrightarrow$ To create constant variables

Final Methods $\longrightarrow$ Prevent Method Overriding

Final Classes $\longrightarrow$ Prevent Inheritance

```
class Main {
  public static void main(String[] args) {

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

```
class Main{
   //Blank final variable
   final int MAX_VALUE;

   Main(){
      //It must be initialized in constructor
      MAX_VALUE=100;
   }
   void myMethod(){
      System.out.println(MAX_VALUE);
   }
   public static void main(String args[]){
      Main obj=new  Main();
      obj.myMethod();
   }
}
```

Java Garbage Collection

In java, 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 of Garbage Collection

- 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.

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 as:

```
protected void finalize(){}
```

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.

```
public static void gc(){}
```