Day 3

Constructor

- It is a method of a class which is used to initialize instance.
- Due to following reason constructor is considered as special method of a class:
    1. Its name is same as class name.
    2. It doesn't have any return type
    3. It is designed to call implicitly.
    4. It is designed to call once per instance.
- We can use any access modifier on constructor.
- Types of constructor.
    1. Parametereless constructor (User Defined Default constructor).
    2. Parameterized constructor
    3. Default constructor (Compiler Defined default constructor)
- Since static field / class-level variable do not get space inside instance, we should not initialize it in constructor.
- In java, we can call constructor from another constructor. It is called constructor chaining.
- For constructor chaining we should use this statement. "this" statement must be first statement inside constructor body.
- Using constructor chaining, we can reduce developers effort.

Getter and Setter

- Method of a class which is used to modify state of instance is called mutator/modifier/setter method

```
public void setReal(int real)
{
    this.real = real;
}
public void setImag(int imag)
{
    this.imag = imag;
}
```

- Method of a class which is used to read state of instance is called inspector/selector/getter method.

```
 public int getReal()
{
    return this.real;
}
public int getImag()
{
    return this.imag;
}
```

NullPointerException

- Using null object, if we try to access any member of the class then JVM throws Null Pointer exception.

```
Complex c1 = null;
c1.printRecord(); //NullPointerException
```

- Solution

```
Complex c1 = null;
c1 = new Complex(); // to avoid NullPointerException
c1.printRecord(); //Ok
```

- or

```
Complex c1 = new Complex();
c1.printRecord(); //Ok
```

NumberFormatException

- If string does not contain parsable numeric value then parseXXX() method throws NumberFormatException.

```
 String str = "abc";
int number = Integer.parseInt(str);//NumberFormatException
System.out.println("Number : "+number);
```

System Date

- Using java.time.LocalDate

```
LocalDate ldt = LocalDate.now();
int day = ldt.getDayOfMonth();
int month = ldt.getMonthValue();
int year = ldt.getYear();
System.out.println(day+" / "+month+" / "+year);
```

- Using java.util.Calendar

```
 Calendar c = Calendar.getInstance();
//int day = c.get( Calendar.DAY_OF_MONTH);
int day = c.get( Calendar.DATE);
int month = c.get( Calendar.MONTH ) + 1;
int year = c.get( Calendar.YEAR );
System.out.println(day+" / "+month+" / "+year);
```

- Using java.util.Date

```
Date date = new Date();
int day = date.getDate();
int month = date.getMonth() + 1;
int year = date.getYear() + 1900;
System.out.println(day+" / "+month+" / "+year);
```

- If we want to format date then we should use java.text.SimpleDateFormat class.

```
Integer n1 = new Integer( );    //Not OK
Integer n2 = new Integer( 10 );    //OK
Integer n3 = new Integer( "10" );    //OK
```

Object Class

- It is a non final and concrete class declared in java.lang package.
- It is root of java class hierarchy.
- It is also called as ultimate base class or super cosmic base class or root class.
- java.lang.Object is super class of all the classes(not interfaces) in java language.
- It is introduced in jdk 1.0.
- java.lang.Object class do not contain nested type and field.
- Object class contain only parameterless constructor.

```
Object o1 = new Object("SunBeam");  //Not OK
Object o2 = new Object();  //OK
```

- Object class contain 11 methods(5 non final + 6 final).

1. public String toString();
2. public boolean equals(Object obj);
3. public native int hashCode();
4. protected native Object clone() throws CloneNotSupportedException;
5. protected void finalize() throws Throwable;
6. public final native Class<?> getClass();
7. public final void wait() throws InterruptedException

8. public final native void wait(long timout) throws InterruptedException;
9. public final void wait(long timout, int nanos) throws InterruptedException;
10. public final native void notify();
11. public final native void notifyAll();

Boxing:

- Process of converting state of instance of value type into reference type is called boxing.

```
int number = 125;
Integer n1 = Integer.valueOf(number);   //Boxing
```

```
int number = 125;
String strNumber =  String.valueOf(number); //Boxing
```

- If boxing is done implicitly then it is called auto boxing.

```
int number = 125;
Object obj = number;
//Object obj = Integer.valueOf(number);
```

UnBoxing

- Process of converting, state of instance of reference type into value type is called unboxing.

```
String str = "125";
int number = Integer.parseInt(str);//UnBoxing
```

```
Integer n1 = new Integer("125");
int n2 = n1.intValue(); //UnBoxing
```

- If unboxing is done implicitly then it is called auto unboxing.

```
Integer n1 = new Integer("125");
int n2 = n1; //Auto UnBoxing
//int n2 = n1.intValue(); //UnBoxing
```

Static Concept

- In java, we not declare local variable static.

- Static variable is also called as class-level variable. Since class level variable must exist in class scope, local variable can not be declared as static.

```
public static void print( )
{
    static int count = 0; //Not OK
    ++ count;
    System.out.println("Count   :   "+count);
}
```

- We can not declare local variable static but we can declare field static.

```
class Test
{
    private static int count;
    public static void print( )
    {
        ++ count;
        System.out.println("Count   :   "+count);
    }
}
class Program
{
    public static void main(String[] args)
    {
        Test.print();   //1
        Test.print();   //2
        Test.print();   //3
    }
}
```

- If we want to share value of any field in all the instances of same class then we should declare that field static.
- Static field i.e class level variable get space once during class loading on Method area once per class.
- If we want to intialize static field then we should use static initializer block.
- JVM execute static initializer block during class loading.
- we can write multiple static initializer block inside class. In this case JVM execute it sequentially.

```
class Test
{
    private static int number;
    static //Static Initializer block
    {
        Test.number = 500;
    }
}
```

- If we want to access non static members of the class then we should define non static method inside class.
- Non static methods are designed to callon instance.
- If we want to access static members of the class then we should define static method inside class.
- Static methods are designed to call on class name.
- Since static methods are designed to call on class name, 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.
- Using instance, we can access non static members inside static method.

```
class Program
{
    private int num1 = 10; //Instance variable
    private static int num2 = 20; //Class level variable
    public static void main(String[] args)
    {
        //System.out.println("Num1    :   "+num1);//Error
        Program p = new Program();
        System.out.println("Num1    :   "+p.num1);//10
        System.out.println("Num2    :   "+num2);//20
    }
}
```

```
class A
{
    public static void f1( ){   }
}
class Program
{
    public static void f2( ){   }
    public static void main(String[] args)
    {
        //f1( );  //Not OK
        A. f1( );  //OK
        f2( );  //OK
        Program.f2( );  //OK
    }
}
```

- Program for instance counter

```
class InstanceCounter
{
    private static int count;
    public InstanceCounter( )
    {
        ++ InstanceCounter.count;
```

```
    }
    public static int getCount()
    {
        return InstanceCounter.count;
    }
}
```

- A class from which we can create only instance is called singleton class.

```
class Singleton
{
    private Singleton( )
    {   }
    private static Singleton instance;
    public static Singleton getInstance( )
    {
        if( Singleton.instance == null )
            Singleton.instance = new Singleton();
        return Singleton.instance;
    }
}
```

- Second implementation

```
class Singleton
{
    private static Singleton instance;
    static
    {
        Singleton.instance = new Singleton();
    }
    private Singleton( )
    {   }
    public static Singleton getInstance( )
    {
        return Singleton.instance;
    }
}
```

Final Variable

- final is keyword in java.
- if we dont want to modify state of the variable then we should declare it final.

```
public static void main(String[] args)
{
    final int number = 10;
```

```
    //++ number;  //Not OK
    System.out.println("Number  :   "+number);
}
```

- We can not declare local variable static but we can declare it final.

```
public static void main(String[] args)
{
    //final int number = 10;  //Ok
    final int number;  //Ok
    number = 10;//OK
    //number = 20;//Not OK
    System.out.println("Number  :   "+number);
}
```

- After storing value, if we dont want to modify state then we should use final keyword.

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

    Console console = System.console();
    System.out.println("Enter number    :   ");
    final int number = Integer.parseInt(console.readLine());  //Ok
    System.out.println("Number  :   "+number);
}
```

- We can provide value to the final variable at compile time as well as runtime.
- If we dont want to modify state of any field inside any method of the class then we should declare field final.
- Note : If we want to declare any field final then we should declare it static also.

```
class Math
{
    public static final double PI = 3.14;
}
```

- In java, we can not declare instance final but we can declare reference final.

```
public static void main(String[] args)
{
    final Complex c1 = new Complex(10, 20);
    c1.setReal(100);
    c1.setImag(200);
    c1.printRecord();//100,200
```

```
    //c1 = new Complex( 50, 60 ); //Not OK
}
```