Object Oriented Programming with Java

Sandeep Kulange

Agenda

- Coding Convention
- Reference in Java
- java.lang.Object class
- toString() method
- Final keyword
- Static field
- Static Initialization Block
- Instance Initializer Block
- Static method
- Singleton class

Coding Convention

- **Pascal Case Coding/Naming Convention:**
  - Example
    1. System
    2. StringBuilder
    3. NullPointerException
    4. IndexOutOfBoundsException

  - In this case, including first word, first character of each word must in upper case.

  - We should use this convention for:
    1. Type Name(Interface, class, Enum, Annotation)
    2. File Name

Coding Convention

- **Camel Case Coding/Naming Convention:**
  - Example
    1. main
    2. parseInt
    3. showInputDialog
    4. addNumberOfDays
  - In this case, excluding first word, first character of each word must in upper case.
  - We should use this convention for:
    1. Method Parameter and Local variable
    2. Field
    3. Method
    4. Reference

Coding Convention

- **Naming Convention for package:**
  - We can specify name of the package in uppercase as well as lower-case. But generally it is mentioned in lower case.
  - Example
    1. java.lang
    2. java.lang.reflect
    3. java.util
    4. java.io
    5. java.net
    6. java.sql

Coding Convention

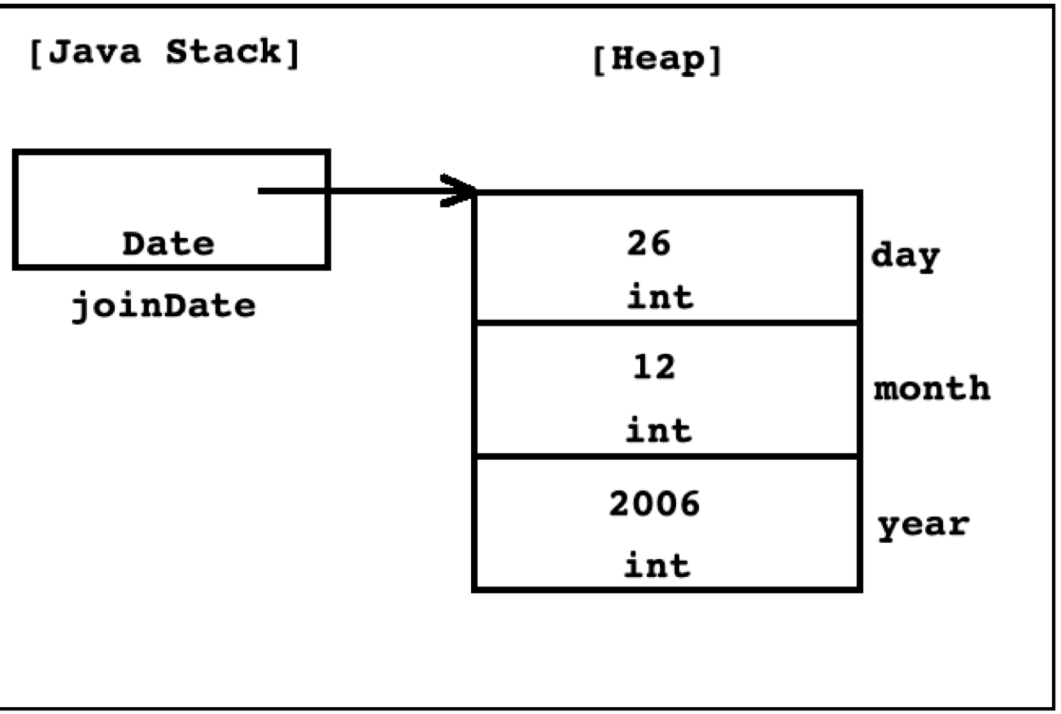
- **Naming Convention for constant variable and enum constant:**
    - Example

```
1. public static final int SIZE;
2. enum Color{ RED, GREEN, BLUE }
3. Name of the final variable and name of the enum constant should be in upper case.
```

Reference

- Local reference variable get space on Java Stack.

```
class Date{
    private int day;
    private int month;
    private int year;
}
class Program{
    public static void main(String[] args) {
        Date joinDate = new Date( 26,12,2006);
        //joinDate : Method Local Variable
    }
}
```



- In above code joinDate is method local reference variable hence it gets space on Java Stack.

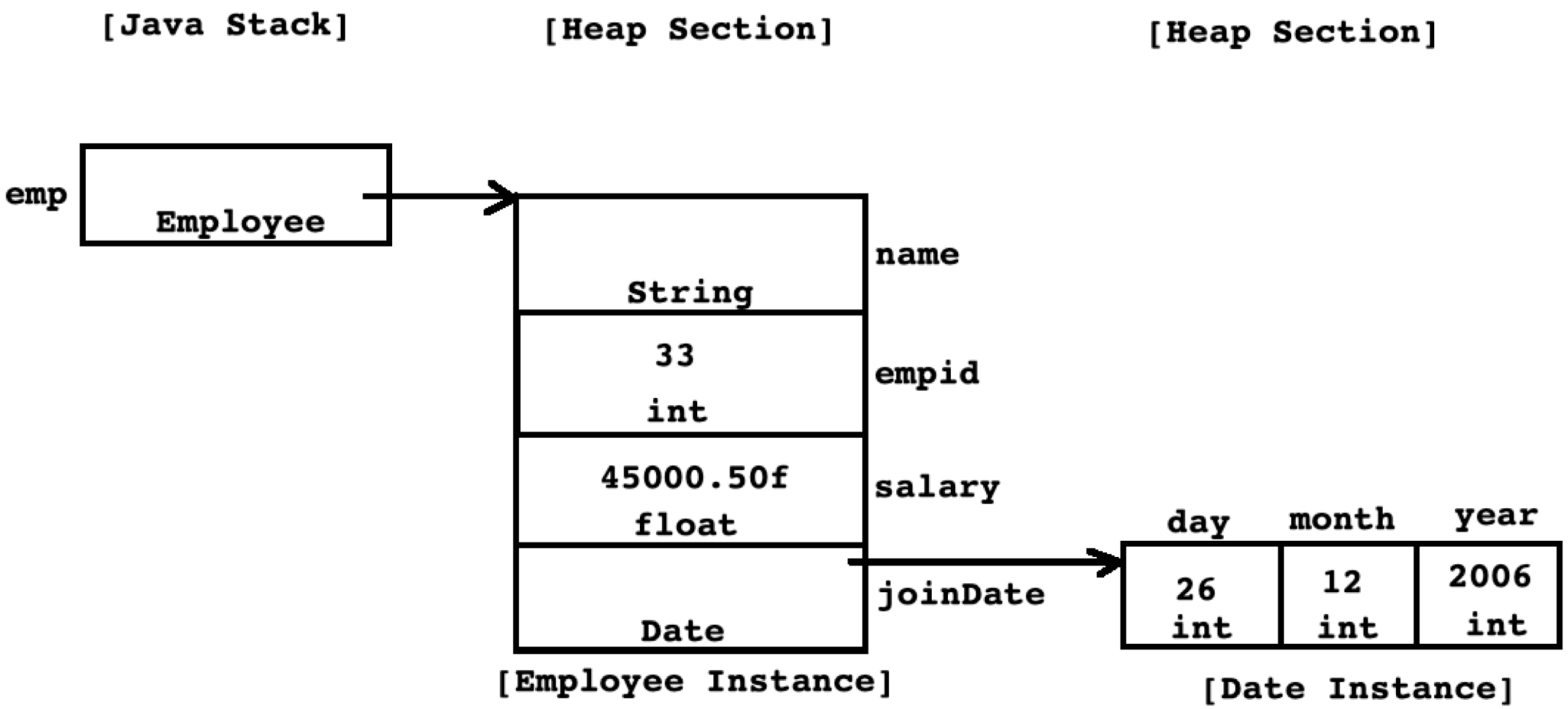
Reference

- Class scope reference variable get space on heap.

```
class Employee{
    private String name;
    private int empid;
    private float salary
    private Date joinDate;  //joinDate : Field
    public Employee( String name, int empid, float salary, Date joinDate ){
        this.name = name;
        this.empid = empid;
        this.salary = salary;
        this.joinDate = joinDate;
    }
}
class Program{
    public static void main(String[] args) {
        Employee emp = new Employee( "Sandeep",33, new Date(26, 12, 2006) );
        //emp : Method Local Variable
    }
}
```

- In above code, emp is method local reference variable hence it gets space on Java Stack. But joinDate is field of Employee class hence it will get space inside instance on Heap.

Reference


[Java Stack]
[Heap Section]
[Heap Section]
emp
Employee
String
name
33
int
empid
45000.50f
float
salary
Date
joinDate
day
month
year
26
int
12
int
2006
int
[Employee Instance]
[Date Instance]

Object class

- It is a non final and concrete class declared in java.lang package.
- In java all the classes(not interfaces)are directly or indirectly extended from java.lang.Object class.
- In other words, java.lang.Object class is ultimate base class/super cosmic base class/root of Java class hierarchy.
- Object class do not extend any class or implement any interface.
- It doesn't contain nested type as well as field.
- It contains default constructor.
  - `Object o = new Object("Hello");  //Not OK`
  - `Object o = new Object( );  //OK`

- Object class contains 11 methods.

Object class

- Consider the following code:

```
class Person{
}
class Employee extends Person{
}
```

- In above code, java.lang.Object is direct super class of class Person.

- In case class Employee, class Person is direct super class and class Object is indirect super class.

Methods Of Object class

```
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( void )throws Throwable

6. public final native Class<?> getClass( );
7. public final void wait( )throws InterruptedException
8. public final native void wait( long timeout)throws InterruptedException
9. public final void wait( long timeout, int nanos)throws InterruptedException
10.public final native void notify( );
11.public final native void notifyAll( );
```

toString( ) method

- It is a non final method of java.lang.Object class.
- Syntax:
  - **public String toString( );**
- If we want to return state of Java instance in String form then we should use toString() method.
- Consider definition of toString inside Object class:

```
public String toString() {
    return this.getClass().getName() + "@" + Integer.toHexString(this.hashCode());
}
```

toString( ) method

- If we do not define toString() method inside class then super class's toString() method gets call.
- If we do not define toString() method inside any class then Object class's toString() method gets call.
- It return String in following form:
  - **F.Q.ClassName@HashCode**
  - **Example : test.Employee@6d06d69c**
- If we want state of instance then we should override toString() method inside class.
- The result in toString method should be a concise but informative that is easy for a person to read.
- It is recommended that all subclasses override this method.

Final variable

- In java we do not get const keyword. But we can use final keyword.
- After storing value, if we don't want to modify it then we should declare variable final.

```
public static void main(String[] args) {
    final int number = 10;  //Intialization
    //number = number + 5;    //Not OK
    System.out.println("Number  :   "+number );
}
```

```
    public static void main(String[] args) {
        final int number;
        number = 10;    //Assignment
        //number = number + 5;    //Not OK
        System.out.println("Number  :   "+number );
    }
```

Final variable

- We can provide value to the final variable either at compile time or run time.

```
public static void main(String[] args) {
    Scanner sc = new Scanner( System.in);
    System.out.print("Number    :   ");
    final int number = sc.nextInt();    //OK
    //number = number + 5;    //Not OK
    System.out.println("Number   :   "+number );
}
```

Final field

- Once initialized, if we don't want to modify state of any field inside any method of the class(including constructor body) then we should declare field final.

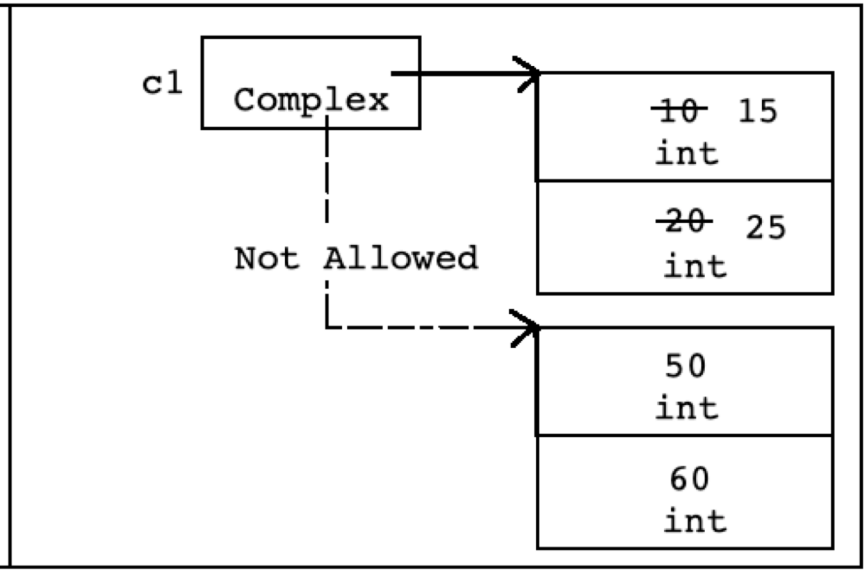
```
class Circle{
    private float area;
    private float radius = 10;
    public static final float PI = 3.142f;
    public void calculateArea( ){
        this.area = PI * this.radius * this.radius;
    }
    public void printRecord( ){
        System.out.println("Area    :   "+this.area);
    }
}
```

- If we want to declare any field final then we should declare it static also.

Final Reference Variable

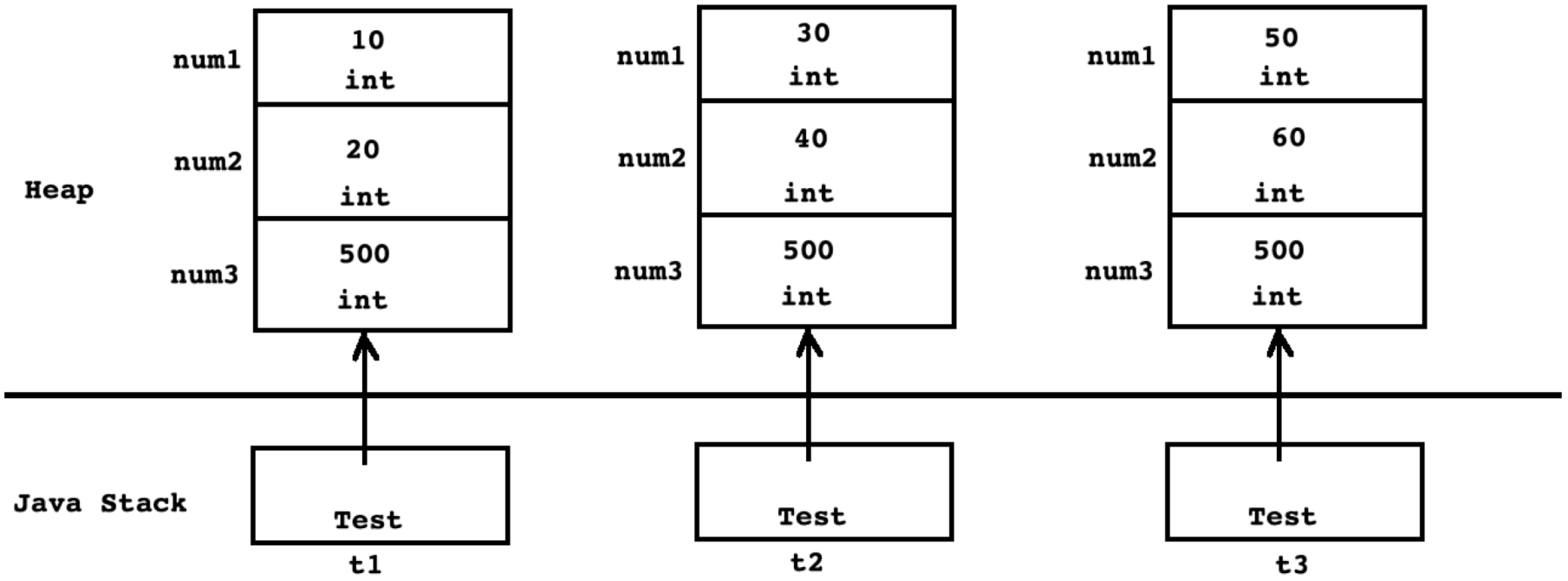
- In Java, we can declare reference final but we can not declare instance final.

```
public static void main(String[] args) {
    final Complex c1 = new Complex( 10, 20 );
    c1.setReal(15);
    c1.setImag(25);
    //c1 = new Complex(50,60);    //Not OK
    c1.printRecord( );      //15, 25
}
```



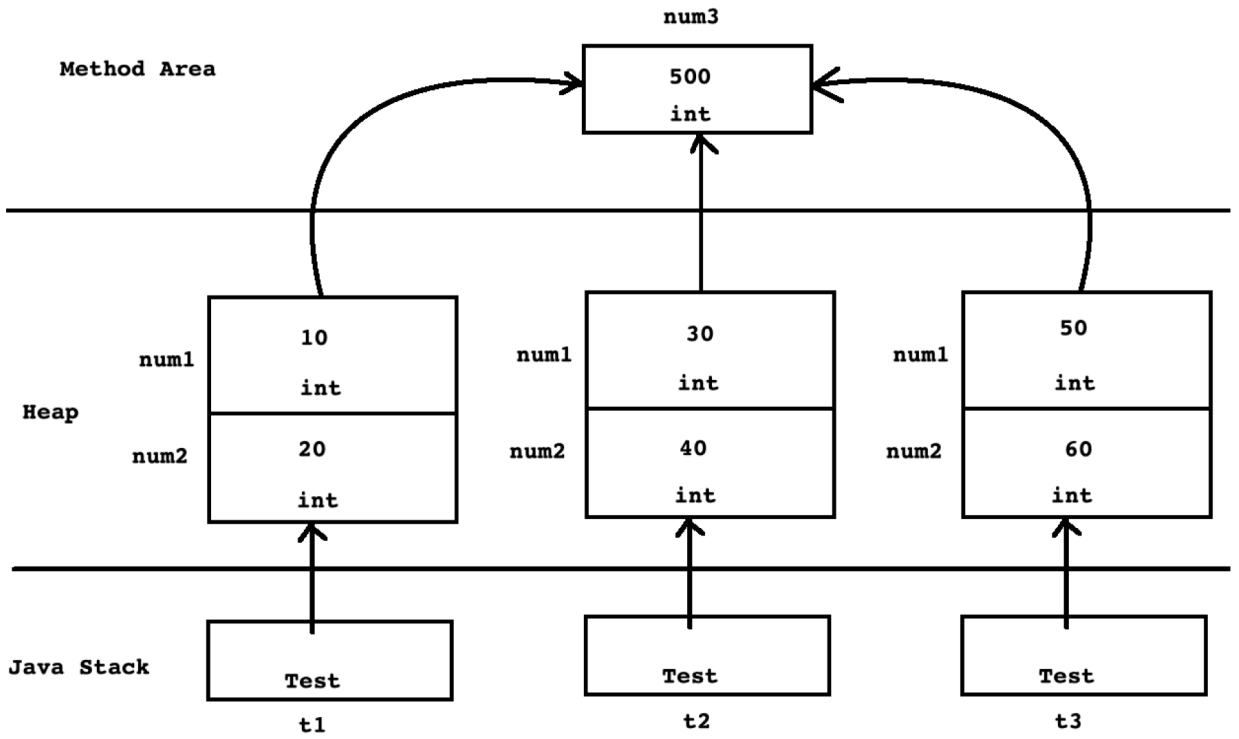
- We can declare method final. It is not allowed to override final method in sub class.
- We can declare class final. It is not allowed to extend final class.

Static Field



Static Field

- If we want to share value of any field inside all the instances of same class then we should declare that field static.



Static Field

- Static field do not get space inside instance rather all the instances of same class share single copy of it.
- Non static Field is also called as instance variable. It gets space once per instance.
- Static Field is also called as class variable. It gets space once per class.
- Static Field gets space once per class during class loading on method area.
- Instance variables are designed to access using object reference.
- Class level variable can be accessed using object reference but it is designed to access using class name and dot operator.

Static Initialization Block

- A *static initialization block* is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

```
static {
    // whatever code is needed for initialization goes here
}
```

- A class can have any number of static initialization blocks, and they can appear anywhere in the class body.
- The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.
- There is an alternative to static blocks – you can write a private static method:

Instance Initializer Block

- Normally, you would put code to initialize an instance variable in a constructor.
- There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.
- Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

```
{
    // whatever code is needed for initialization goes here
}
```

- The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

Instance Initializer Block

- A *final method* cannot be overridden in a subclass.

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        // initialization code goes here
    }
}
```

- This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

Static Method

- To access non static members of the class, we should define non static method inside class.
- Non static method/instance method is designed to call on instance.
- To access static members of the class, we should define static method inside class.
- static method/class level method is designed to call on class name.
- static method do not get this reference:
    1. If we call, non static method on instance then method get this reference.
    2. Static method is designed to call on class name.
    3. Since static method is not designed to call on instance, it doesn't get this reference.

Static Method

- this reference is a link/connection between non static field and non static method.
- Since static method do not get this reference, we can not access non static members inside static method directly. In other words, static method can access static members of the class only.
- Using instance, we can use non static members inside static method.

```
class Program{
    public int num1 = 10;
    public static int num2 = 10;
    public static void main(String[] args) {
        //System.out.println("Num1    :   "+num1);    //Not OK
        Program p = new Program( );
        System.out.println("Num1    :   "+p.num1);    //OK
        System.out.println("Num2    :   "+num2);
    }
}
```

Static Method

- Inside method, If we are going to use this reference then method should be non static otherwise it should be static.

```
class Math{
    public static int power( int base, int index ){
        int result = 1;
        for( int count = 1; count <= index; ++ count ){
            result = result * base;
        }
        return result;
    }
}
```

```
class Program{
    public static void main(String[] args) {
        int result = Math.power(10, 2);
        System.out.println("Result  :   "+result);
    }
}
```

Singleton class

- A class from which, we can create only one instance is called singleton class.
- Steps to define singleton class:
    1. Define class and declare constructor private.
    2. Define factory method and return reference of instance from it.
    3. Use getter and setter to access members.

Thank You.

[sandeepkulange@sunbeaminfo.com]