Programming in Java

this, super, static & final Keywords

Singleton class design pattern

Contents

- final keyword
- this keyword
- static keyword
- super keyword

‘final’ Keyword

- ‘final’ keyword is used to:
  - declare variables that can be assigned value only once (constant).

    final type identifier = expression;

  - prevent overriding of a method.

    final return_type methodName (arguments if any)
    {
    body;
    }

  - prevent inheritance from a class.

    final class Class_Name
    {
    class body;
    }

final variable

```
public class Test
{
    public static void main(String[] args) {
      final int x=10;
      x=x+1;//Compilation error, as we are trying to modify the final variable(constant)
      System.out.println(x);
    }
}
```

final class(Non-inheritable class)

```
final class A
{


}
class B extends A // Compilation error [A is final and non-inheritable]
{


}
public class Test
{
      public static void main(String[] args) {
        B obj=new B();
      }
}
```

final method-It is used to prevent overriding


```
class A
{
   final void show()
   {
      System.out.println("A");
   }
}
class B extends A
{
   void show()//Compilation Error:show() in B cannot override show() in A(Overridden method is final)
   {
      System.out.println("B");
   }
}
public class Test
{
      public static void main(String[] args) {
        B obj=new B();
        obj.show();
      }
}
```

Immutable class

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Following are the requirements:

- The class must be declared as final (So that child classes can't be created)
- Data members in the class must be declared as private (So that direct access is not allowed)
- Data members in the class must be declared as final (So that we can't change the value of it after object creation)
- Only getters methods for accessing the values/ or parameterized constructors must be there to assign the value
- No setters should be there, so that there is no option to change the value of instance variables.

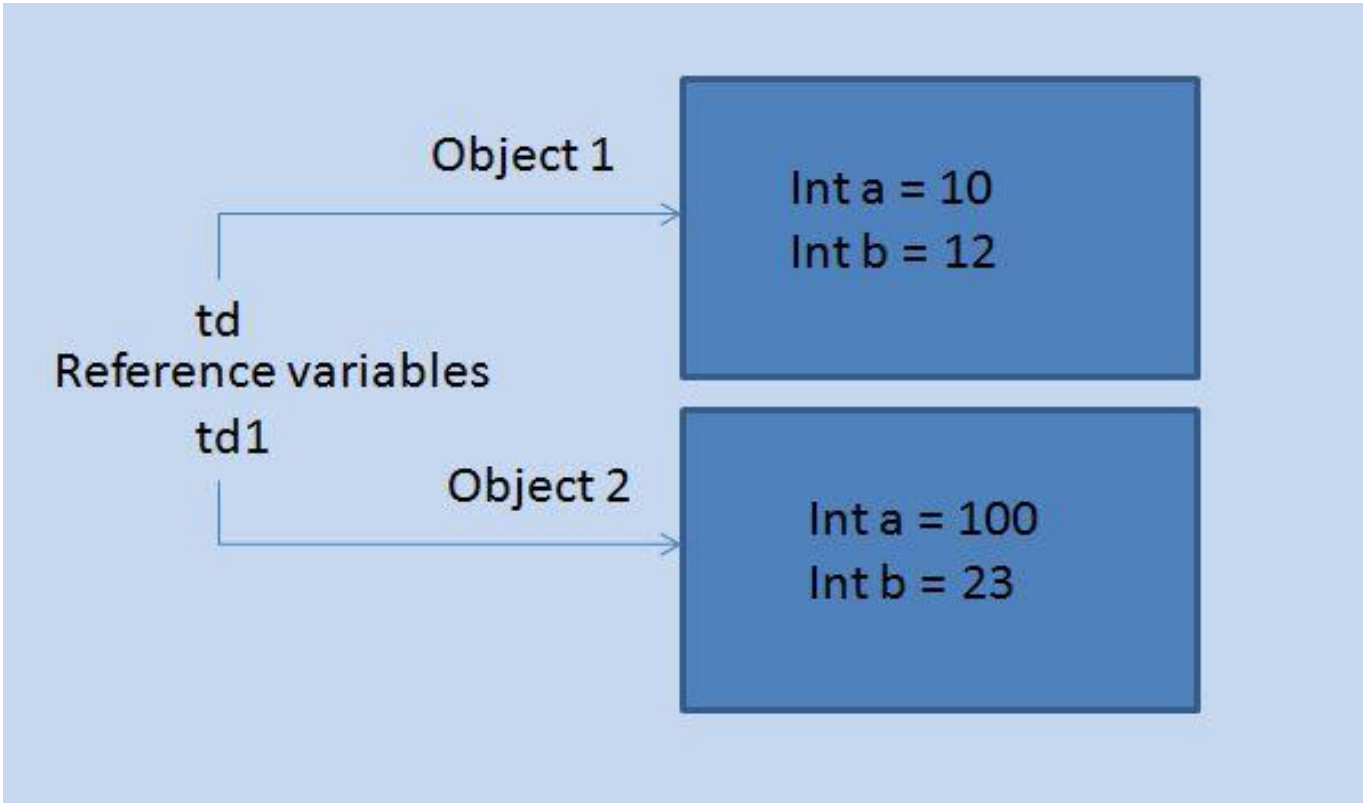
Example

CODERINDEED

```
final class Employee{
final String pancardNumber;
public Employee(String pancardNumber){
this.pancardNumber=pancardNumber;
}
public String getPancardNumber(){
return pancardNumber;
}
}
public class Main
{
   public static void main(String[] args)
   {
      Employee obj=new Employee("123");
      System.out.println(obj.getPancardNumber());
   }
}
```

‘this’ Keyword

- 'this' is used for pointing the current class instance.
- Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called.



```
class ThisDemo1{
                    int a = 0;
                    int b = 0;
                    ThisDemo1(int x, int y)
                        {
                           this.a = x;
                           this.b = y;
                         }

                    public static void main(String [] args)
                        {
                           ThisDemo1 td = new ThisDemo1(10,12);
                           ThisDemo1 td1 = new ThisDemo1(100,23);
                           System.out.println(td.a);
                           System.out.println(td.B);

System.out.println(td1.a);
                        System.out.println(td1.B);
                  }
                     }
```

Chaining of constructors using this keyword

- Chaining of constructor means calling one constructor from other constructor.

- We can invoke the constructor of same class using ‘this’ keyword.
- Rules of constructor chaining :

➢ The this() expression should always be the first line of the constructor.

➢ There should be at-least be one constructor without the this() keyword.

Need of Constructor Chaining

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

Example

CODERINDEED

```
// Java program to illustrate Constructor Chaining
within same class Using this() keyword

public class Temp

{

        // default constructor 1

        // default constructor will call another
constructor

        // using this keyword from same class

        Temp()

        {

                // calls constructor 2

                this(5);

                System.out.println("The Default
constructor");

        }

// parameterized constructor 2

        Temp(int x)

        {

                // calls constructor 3

                this(5, 15);

                System.out.println(x);

        }

        // parameterized constructor 3

        Temp(int x, int y)

        {

                System.out.println(x * y);

        }


        public static void main(String args[])

        {

                // invokes default constructor
first

                new Temp();

        }

}
```

Output:

75

5

The Default constructor

Constructor chaining using super

- **super()** keyword to call constructor from the base class.
- Constructor chaining occurs through **inheritance**. A sub class constructor's task is to call super class's constructor first. This ensures that creation of sub class's object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.
- Note : Similar to constructor chaining in same class, **super()** should be the first line of the constructor as super class's constructor are invoked before the sub class's constructor.

Constructor Chaining using 'super'

CODERINDEED

```
class A
{
   A(int x)
   {
      System.out.println(x);
   }
}
class B extends A
{
   B(int x,int y)
   {
      super(x);
      System.out.println(x+" "+y);
   }
}
class C extends B
{
   C(int x,int y,int z)
   {
      super(x,y);
      System.out.println(x+" "+y+" "+z);
   }
}
public class Main
{
        public static void main(String args[])
        {
                C obj=new C(1,2,3);
        }
}
```

Output:

```
1
1 2
1 2 3
```

‘static’ Keyword

- used to represent class members
- Variables can be declared with the “static” keyword.

static int y = 0;

- When a variable is declared with the keyword “static”, its called a “**class variable**”.
- All instances share the same copy of the variable.
- A class variable can be accessed directly with the class, without the need to create a instance.

- Note: There's no such thing as static classs. “static” in front of class creates compilation error.

static methods

- Methods can also be declared with the keyword “static”.
- When a method is declared static, it can be used without creating an object.

```
class T2 {
            static int triple (int n)
                  {return 3*n;}
          }

class T1 {
             public static void main(String[] arg)
                  {
                     System.out.println( T2.triple(4) );
                     T2 x1 = new T2();
                     System.out.println( x1.triple(5) );
                  }
           }
```

- Methods declared with “static” keyword are called “**class methods**”.
- Otherwise they are “**instance methods**”.
- **Static Methods Cannot Access Non-Static Variables.**
- The following gives a compilation error, unless x is also static.

```
class T2 {
        int x = 3;
        static int returnIt () { return x;}
    }

class T1 {
        public static void main(String[] arg) {
                System.out.println( T2.returnIt() ); }
    }
```

‘super’ Keyword

- ‘super’ keyword is used to:
  - invoke the super-class constructor from the constructor of a sub-class.

    super (arguments if any);

  - invoke the method of super-class on current object from sub-class.

    super.methodName (arguments if any);

  - refer the super-class data member in case of name-conflict between super and sub-class data members.

    super.memberName;

Important

‘this’ and ‘super’ both are non-static.

.

Singleton Class(or Singleton Design Pattern)

- In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.
- After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created.
- So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

To design a singleton class:

- Make constructor as private.
- Write a static method that has return type object of this singleton class.

- **Singleton Design Pattern**
- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.

- The singleton class must provide a global access point to get the instance of the class.

Advantage:

Saves memory because object is not created at each request. Only single instance is reused again and again.

- Usage of Singleton Design Pattern

- Singleton pattern is mostly used in multi-threaded and database applications.

- It is used in logging, caching, thread pools, configuration settings etc.

Creating Singleton Design Pattern

To create the singleton class, we need to have static member of class, private constructor and static factory method.

- Static member: It gets memory only once because of static, it contains the instance of the Singleton class.

- Private constructor: It will prevent to instantiate the Singleton class from outside the class.

- public static factory method: This provides the global point of access to the Singleton object and returns the instance to the caller.

Example

CODERINDEED

```
// Classical Java implementation of singleton design pattern

class MySingleton
{
        private static MySingleton obj;
    public int x;
        // private constructor to force use of getInstance() to create Singleton object
        private MySingleton() {x=12;}
        public static MySingleton getInstance()
        {
                if (obj==null)
                        obj = new MySingleton();
                return obj;
        }
}
```

```
public class Main
{
    public static void main(String[] args)
    {
        MySingleton obj1=MySingleton.getInstance();
        MySingleton obj2=MySingleton.getInstance();
        System.out.println(obj1+" "+obj2);//Same reference for obj1 and obj2
        System.out.println(obj1.x+" "+obj2.x);//12 12
        obj1.x=23;
        obj2.x=45;
        System.out.println(obj1.x+" "+obj2.x);//45 45
    }
}
```

Explanation

Here we have declared getInstance() static so that we can call it without instantiating the class. The first time getInstance() is called it creates a new singleton object and after that it just returns the same object. Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation.