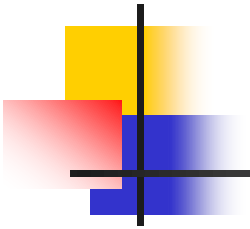
03. Class concepts – (Java)

Class fundamentals
Methods
Constructors
Access Modifiers
Inner Classes

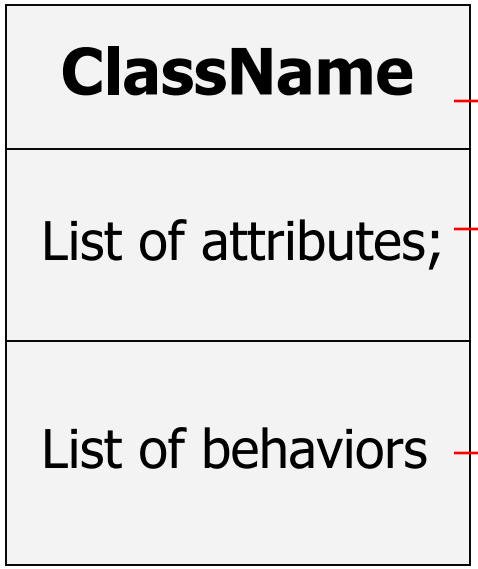


Class Fundamentals

- What is a Class?
  - A Class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

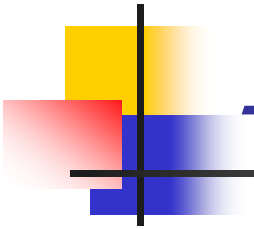
The General Form of a Class

UML Class Diagram



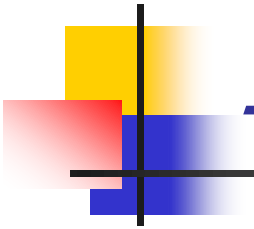
Java Representation

```
class ClassName {
    type instance-variable;
    type methodName(paramenter-list) {
        // method body.
    }
}
```



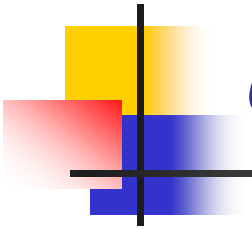
Instance Variable

- Any item of data that is associated with a particular object. Each object has its own copy of the instance variables defined in the class. Also called a field.



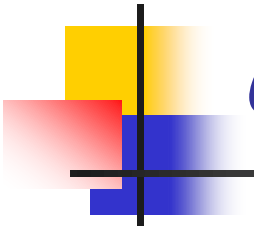
Instance Method

- Any method that is invoked with respect to an instance of a class. Also called simply a method.



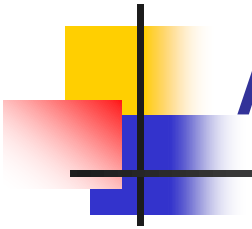
class variable

- A data item associated with a particular class as a whole--not with particular instances of the class. Class variables are defined in class definitions. Also called a *static field*.



class method

- A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class. Also called a *static method*.



A Simple Class

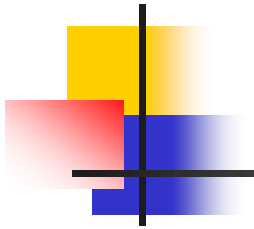
```
class Box {
    double width;
    double height;
    double depth;
}
```

Object Creation - instantiation

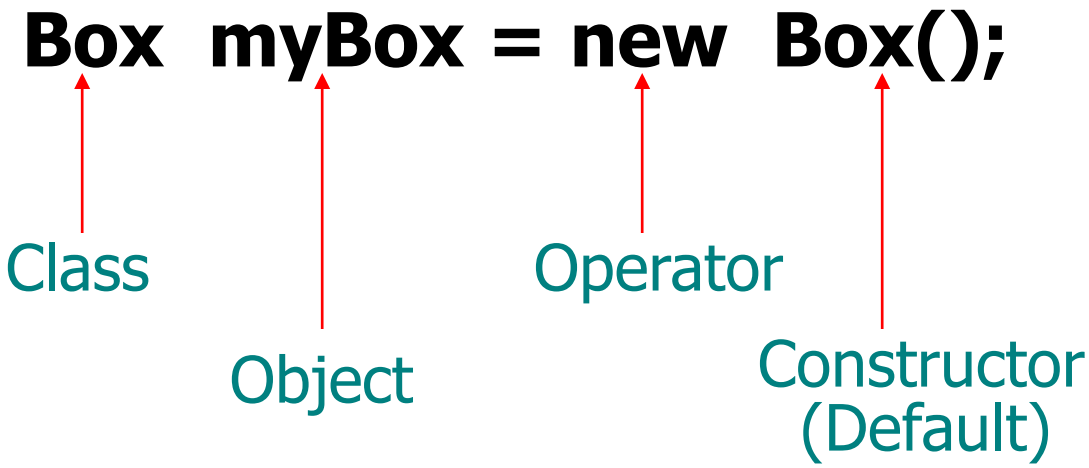
```
class Box {
        double width;
        double height;
        double depth;
}
```

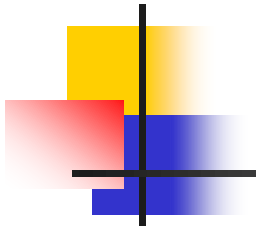
```
public class BoxDemo {
    public static void main(String args[]) {
        Box myBox = new Box();
        double vol;
        myBox.width = 10;
        myBox.height = 20;
        myBox.depth = 15;
        vol = myBox.width * myBox.depth
            * myBox.height;
        System.out.println("Volume is:" +vol);
    }
}
```

This way we can create any number of objects of Box say myBox1, myBox2, myBox3, …



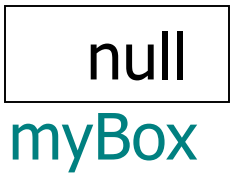
Class Instantiation Statement:



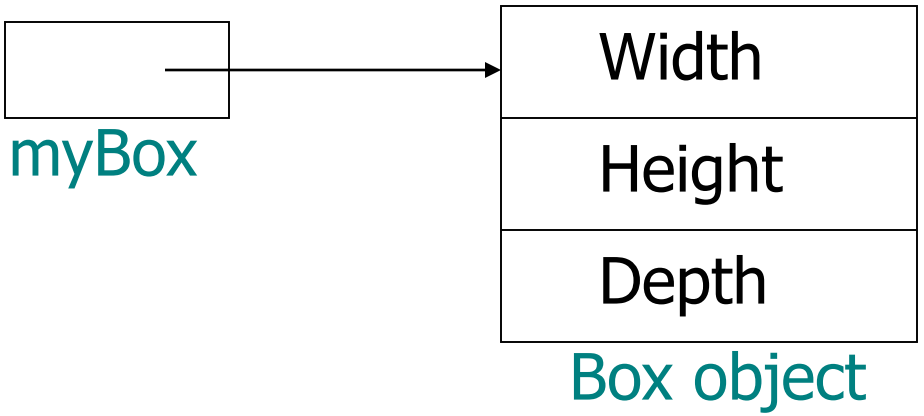


Instantiation is a 2 step process:

Box myBox;



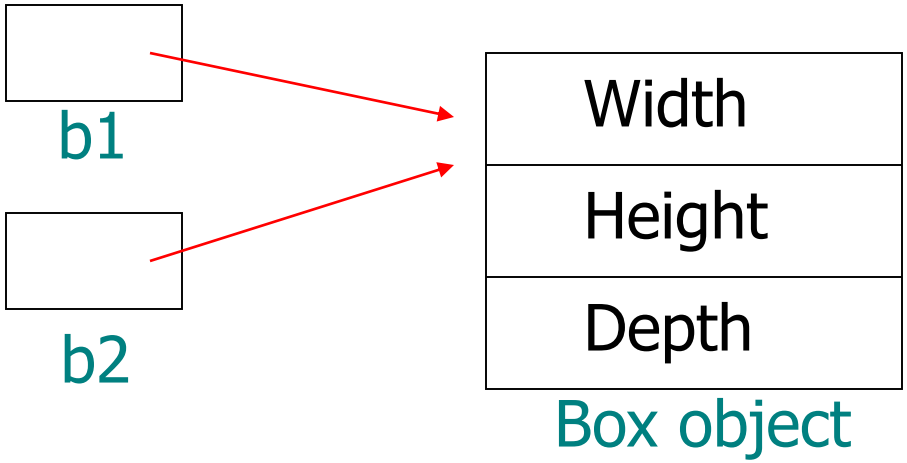
myBox = new Box();



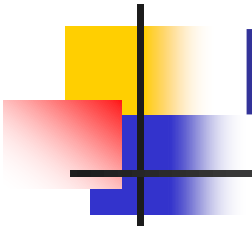
Object Reference Assignment

Box b1 = new Box();

Box b2 = b1;



- Both b1 and b2 refer to the same object and not two distinct objects, but they are not linked in any other way.

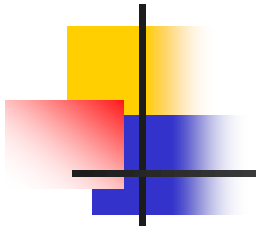


Methods

- Method signature:

```
return_type methodName(parameter_list) {
    // method body
}
```

- ***return_type*** specifies the type of data returned by the method. If the method does not return a value, its return type must be ***void***.
- ***methodName*** – any legal identifier other than the keywords.



- ***parameter_list*** – sequence of type and identifier pairs separated by commas.
- Methods that have a return type other than void return a value to the calling routine using a return statement as given below:

  return value;

  Here value is the value returned.

Box Class - Adding a method

```
class Box  {
        double width;
        double height;
        double depth;

        //  Instance method.
        void getVolume()  {
                System.out.print("Volume is: ");
                System.out.println(width*height*depth);
        }
}
```

Instance variables

```
public class BoxDemo  {

    public static void main(String args[])  {

        Box myBox1 = new Box();
        Box myBox2 = new Box();

        myBox1.width = 10;
        myBox1.height = 20;
        myBox1.depth = 15;

        myBox2.width = 3;
        myBox2.height = 6;
        myBox2.depth = 9;

        myBox1.getVolume();

        myBox2.getVolume();
    }
}
```

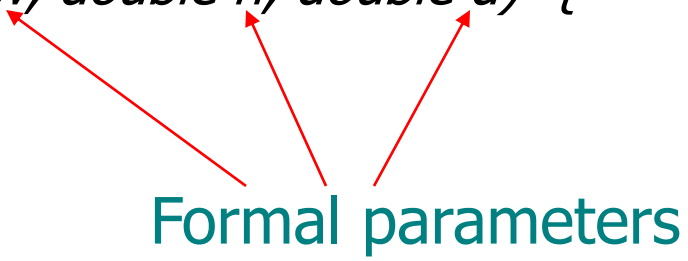
Method returning a value

```
class Box  {
        double width;
        double height;
        double depth;

        //  Instance method.
        double getVolume()  {
                double volume = width*height*depth;
                return volume;
        }
}
```

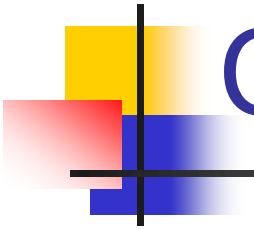
```
public class BoxDemo  {

    public static void main(String args[])  {

        Box myBox1 = new Box();
        double volume;

        myBox1.width = 10;
        myBox1.height = 20;
        myBox1.depth = 15;

        volume = myBox1.getVolume();
        System.out.println("Volume is: " + volume);

    }
}
```

Method that takes a parameter

```
class Box  {
        double width;
        double height;
        double depth;

        //  Instance method.
        double getVolume()  {
                return width*height*depth;
        }

        void setDim( double w, double h, double d)  {
                width = w;
                height = h;
                depth = d;
        }
}
```

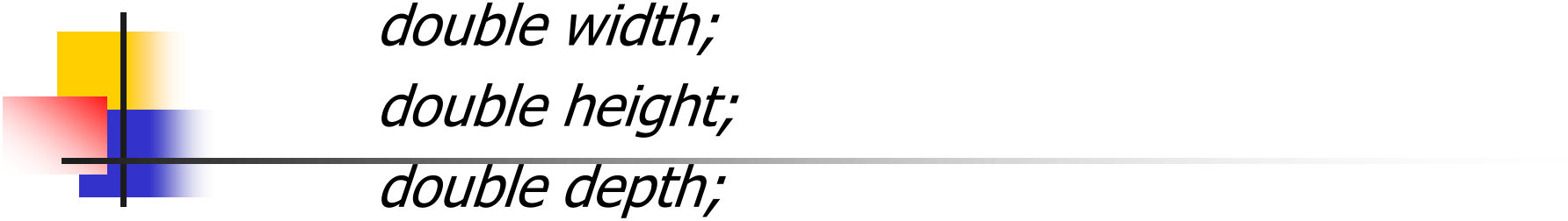


```
public class BoxDemo  {

    public static void main(String args[])  {

        Box myBox1 = new Box();
        double volume;

        myBox1.setDim(10, 20, 15);

        volume = myBox1.getVolume();
        System.out.println("Volume is: " + volume);

    }
}
```



Constructors

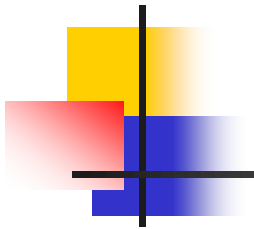
- Instead of using a separate method for initializing an object during its creation, it is more convenient and concise to initialize them automatically when they are created.
- This automatic initialization is done by a special method called a ***constructor***.
- A ***constructor*** is special because it does not have a return type, not even void.



```
class Box  {
        double width;
        double height;
        double depth;

        //  Constructor
        Box()  {
                width = 10;
                height = 10;
                depth = 10;
        }
}
```

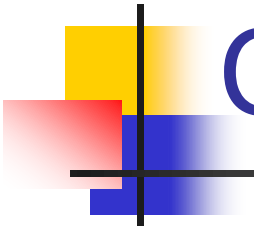
Instantiation:

```
Box myBox = new Box();
```



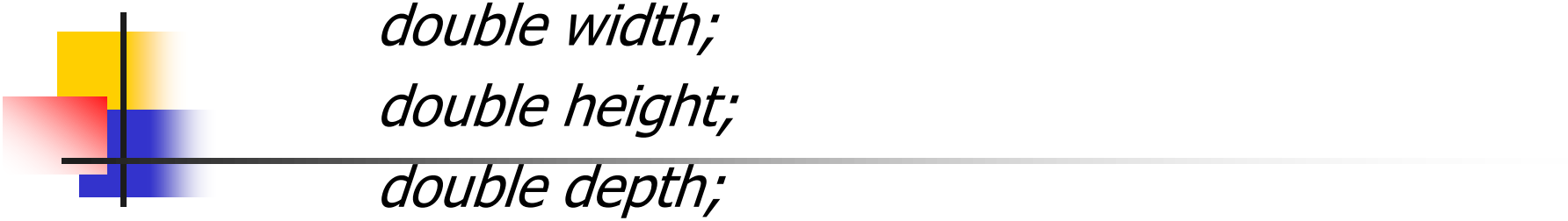
```
class Box  {
        double width;
        double height;
        double depth;

        //  Constructor
        Box(double w, double h, double d)  {
                width = w;
                height = h;
                depth = d;
        }
        //  Instance method.
        double getVolume()  {
                return width*height*depth;
        }
}
```

```
public class BoxDemo  {

    public static void main(String args[])  {

        Box myBox1 = new Box(10, 20, 15);
        Box myBox2 = new Box(3, 6, 9);
        double volume;

        volume = myBox1.getVolume();
        System.out.println("Volume is: " + volume);

        volume = myBox2.getVolume();
        System.out.println("Volume is: " + volume);

    }
}
```



Constructors

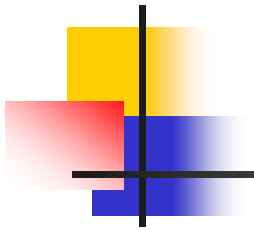
- Instead of using a separate method for initializing an object during its creation, it is more convenient and concise to initialize them automatically when they are created.
- This automatic initialization is done by a special method called a ***constructor***.
- A ***constructor*** is special because it does not have a return type, not even void.



```
class Box  {
        double width;
        double height;
        double depth;

        //  Constructor
        Box()  {
                width = 10;
                height = 10;
                depth = 10;
        }
}
```

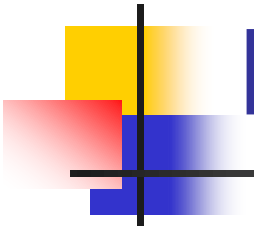
Instantiation:

```
Box myBox = new Box();
```



```
class Box  {
        double width;
        double height;
        double depth;

        //  Constructor
        Box(double w, double h, double d)  {
                width = w;
                height = h;
                depth = d;
        }
        //  Instance method.
        double getVolume()  {
                return width*height*depth;
        }
}
```

```
public class BoxDemo  {

    public static void main(String args[])  {

        Box myBox1 = new Box(10, 20, 15);
        Box myBox2 = new Box(3, 6, 9);
        double volume;

        volume = myBox1.getVolume();
        System.out.println("Volume is: " + volume);

        volume = myBox2.getVolume();
        System.out.println("Volume is: " + volume);

    }
}
```



Method Overloading

- Defining two or more methods within the same class that share the same name, as long as their parameter declarations are different is called method overloading.
- This is the way Java implements polymorphism.

```
class OverloadDemo  {

    void test() {
        System.out.println("No parameters");
    }

    void test(int a) {
        System.out.println("a: " +a);
    }
                                    // double test(int a) {. . . } - Wrong
    void test(int a, int b) {
        System.out.println("a and b: " +a + " " +b);
    }

    double test(double a) {
        System.out.println("double a: " +a);
        return a*a;
    }
}
```

```
class Overload  {

    public static void main(String args[])  {
        OverloadDemo ob = new OverloadDemo();
        double result;

        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): "
                                        +result);
    }
}
```

Constructor Overloading

- Constructors can also be overloaded.

```
Box(double w, double h, double d)  {
        width = w;   height = h;   depth = d;
}

Box()  {
        width = -1;   height = -1;   depth = -1;
 }

Box(double len)  {
        width = height = depth = len;
}
```

The Class Declaration

`public`	Class is publicly accessible.
`abstract`	Class cannot be instantiated.
`final`	Class cannot be subclassed.
`class NameOfClass`	***Name of the Class.***
`extends` *`Super`*	Superclass of the class.
`implements` *`Interfaces`*	Interfaces implemented by the class.

`{`
`  `*`ClassBody`*
`}`	

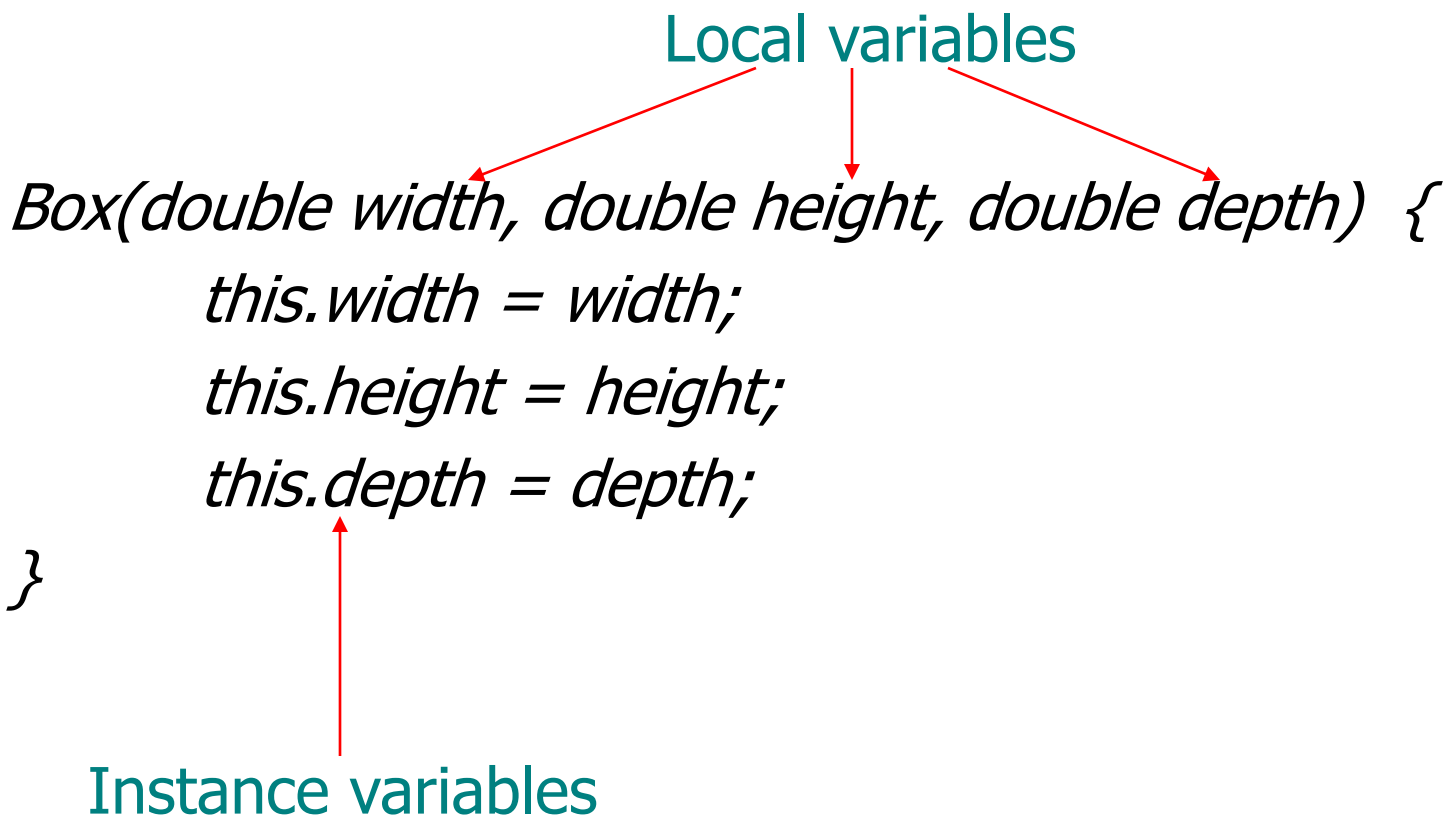
Declaring Member Variables

accessLevel	Indicates the access level for this member.
static	Declares a class member.
final	Indicates that it is constant.
transient	This variable is transient.
volatile	This variable is volatile.
type name	The type and name of the variable.

Details of a Method Declaration

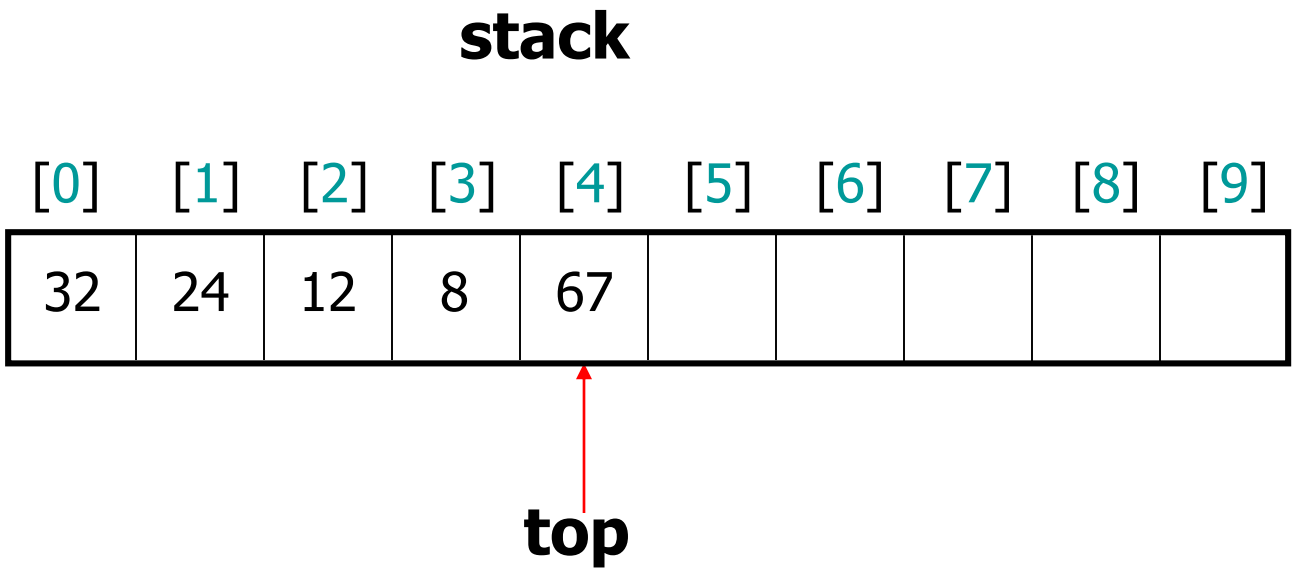
accessLevel	Access level for this method.
`static`	This is a class method.
`abstract`	This method is not implemented.
`final`	Method cannot be overridden.
`native`	Method implemented in another language.
`synchronized`	Method requires a monitor to run.
returnType methodName	The return type and method name.
`(` *paramList* `)`	The list of arguments.
`throws` *exceptions*	The exceptions thrown by this method.

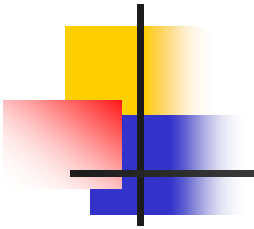
'this' keyword



Example: Stack implementation

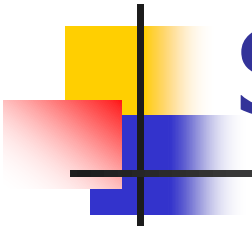
- Implement a class **Stack** with 2 basic operations – ***push*** and ***pop***.





Static Members

- A static class member can be accessed directly by the class name and doesn’t need any object. A single copy of a static member is maintained throughout the program regardless of the number of objects created.
- Static variables are initialized only once and at the start of the execution during the lifetime of a class. These variables will be initialized first before the initialization of any instance variables.



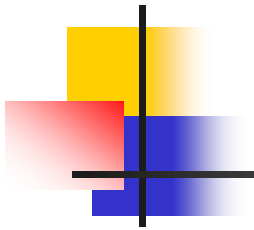
Static Members

Methods declared as static (class methods) have several restrictions:

- They can only call other static methods.
- They must only access static data.
- They cannot refer to *this* or *super* in anyway.
- These methods can be accessed using the class name rather than a object reference.
- *main()* method should be always static because it must be accessible for an application to run, before any instantiation takes place.
- When *main()* begins, no objects are created, so if you have a member data, you must create an object to access it.

Static methods/Data members

```
public class Print  {

    public static String name = "default";

    public static void printName()      {
            System.out.println(name);
    }

    public static void main(String arg[])  {
            System.out.println(Print.name);
            Print.printName();
    }
}
```



```
class TrackObj
{
        //class variable
        private static int counter = 0;

        //instance variable
        private int x = 0;

        TrackObj()
        {
                counter++;
                x ++;
        }

        //member method
        public int getX()
        {
                return x;
        }

        //class method
        public static int getCounter()
        {
                return counter;
        }
}
```