Objects and Classes

Objects and Classes

- **Object -** Objects have states and behaviors.

  E.g. : A dog has states - color, name, breed

  behaviors -wagging, barking, eating.

  - An object is an instance of a class.
  - Real-world objects : Cars, Dogs, Humans.
- **Class -** A class can be defined as a template/blue print of object.

Objects and Classes

- **Class :** A class is a user-defined data type with a template that serves to define its properties.

  Syntax:

  ```
  class Classname [extends superclassname]
  {
      [field declaration;]
      [methods declaration;]
  }
  ```

  Where, field declaration is

  ```
  datatype var1, var2,…varn
  ```

  Method declaration is

  ```
  returntype functionname (parameterlist)
  {
      function body
  }
  ```

  Here, parameter list is

  ```
  datatype var1, datatype var2,…datatype varn
  ```

- The methods and variables defined within a class are called *members of the class.*

Example

```
class Sample{
  int a,b;
  void getdata(int x, int y){
      a=x;
      b=y;
      System.out.println("sum is :"+(a+b));
  }
}
class Test{
  public static void main(String args[]){
      Sample s = new Sample();
      s.getdata(10,20);
  }
}
```

Output:

```
Sum is :30
```

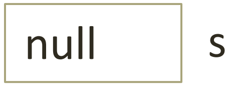
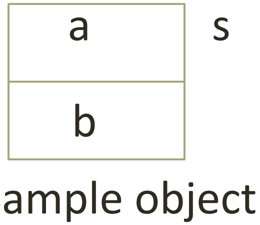
- Sample and Test two .class will be created.

Types of variables

- A class can have 3 types of variables:
- **Local variables:**
  - Variables defined inside methods, constructors or blocks are called local variables.
  - The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables:**
  - Instance variables are variables within a class but outside any method.
  - These variables are instantiated when the class is loaded.
  - Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables:**
  - Class variables are variables declared with in a class, outside any method, with the static keyword.

Creating object

```
Sample s;               //declare the object
s= new Sample();        //instantiate the object
```

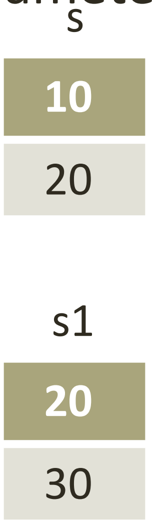
Action	Statement	Result
Declaration	Sample s;	 null s
Instantiate	s = new Sample();	 a s b Sample object

Accessing class members

- Objectname.variablename =value;
- Objectname.methodname(parameter-list);

s.x =10;
s.y = 20;

s1.x =20;
s1.y =30;



- Sample s = new Sample();
- Sample s1 =new Sample();
- Both object s and s1 have their independent memory locations.
- s object occupies 8 bytes (4 bytes * 2) which contains 10 and 20 respectively.
- s1 object occupies 8 bytes (4 bytes * 2) which contains 20 and 30 respectively.

Example

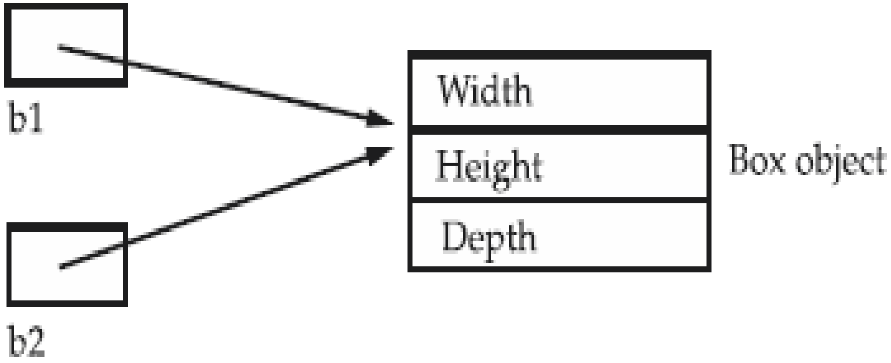
```
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
   public static void main(String args[]) {
          Box mybox = new Box();
          double vol;
          // assign values to mybox's instance variables
          mybox.width = 10;
          mybox.height = 20;
          mybox.depth = 15;
          // compute volume of box
          vol = mybox.width * mybox.height * mybox.depth;
          System.out.println("Volume is " + vol);
   }
}
```

Output:

```
Volume is 3000.0
```

Assigning Object Reference variables

```
Box b1 = new Box();
Box b2 = b1;
```



```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

- b1 has been set to null, but still b2 points to the original object.

Adding a method to the Box class

```
// This program includes a method inside the box class.
class Box {
        double width;
        double height;
        double depth;
        // display volume of a box
        void volume() {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
```

```
class BoxDemo1 {
    public static void main(String args[]) {
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            // assign values to mybox1's instance variables
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;
            /* assign different values to mybox2's instance variables */
            mybox2.width = 3;
            mybox2.height = 6;
            mybox2.depth = 9;
            // display volume of first box
            mybox1.volume();
            // display volume of second box
            mybox2.volume();
            }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Returning a value

```
class Box {
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

```
class BoxDemo2 {
   public static void main(String args[]) {
   Box mybox1 = new Box();
   double vol;
   // assign values to mybox1's instance variables
   mybox1.width = 10;
   mybox1.height = 20;
   mybox1.depth = 15;
   // get volume of first box
   vol = mybox1.volume();
   System.out.println("Volume is " + vol);
   //System.out.println("Volume is " + mybox1.volume());
   }
}
```

2 things to remember

- The type of data returned by a method must be compatible with the return type specified by the method.
- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Adding a method that takes arguments

```
class Box {
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume() {
          return width * height * depth;
          }
    // sets dimensions of box
    void setDim(double w, double h, double d) {
         width = w;
         height = h;
         depth = d;
    }
}
```

```
class BoxDemo3 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

Steps for creating an object

- 3 steps when creating an object from a class:
  - **Declaration:** A variable declaration with a variable name with an object type.
    - Box mybox ;
    - Any attempt to use s at this point will give compile time error.
  - **Instantiation:** The 'new' key word is used to create the object.
    - Box mybox = new Box();
  - **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.
    - Box mybox = new Box(1.2,2.4,3.1);

Constructors

- A constructor initializes an object immediately upon creation.
- It has same name as the class in which it resides.
- Syntactically similar to a method.
- No return type, not even void.
  - Because implicit return type of class constructor is class type.
- Replace setter methods with a constructor.

Example

```
//Box uses a constructor to initialize the dimensions of a box.
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
    // compute and return volume
    double volume() {
    return width * height * depth;
    }
}
```

Con’t

```
class BoxDemo4 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

Output:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

- Default constructor automatically initializes all instance variable to 0.

Parameterized Constructors

```
// Box uses a parameterized constructor to initialize the dimensions of a box.
  class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
     Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
      }
      // compute and return volume
      double volume() {
        return width * height * depth;
      }
  }
```

Con’t

```
class BoxDemo5 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

this keyword

- this refer to the current object.
- this always reference to the object on which the method was invoked.

```
// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Instance Variable hiding

- Illegal to declare two local variable with the same name inside the same scopes.
- Collision occur between instance and local variables.
- Use of this

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Garbage collection

- Dynamically memory is allocated to objects using new, but how such objects are destroyed and their memory is released for later reallocation.
- In C++, delete operator is used.
- In Java, it handles deallocation automatically using technique garbage collection.
- The garbage collector runs periodically
  - Checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.

Finalize() Method

- In finalize() method define specific actions that must be performed before an object is destroyed by the garbage collector.
- The java runtime calls the method whenever it is about to recycle an object of that class.

Overloading Methods

- Methods having same name but their parameter declarations different.
    - i.e. must differ in the type and/or number of their parameters.
    - Return type alone is insufficient to distinguish two versions of a method.
    - Methods are said to be overloaded and the process is referred to as method overloading.
    - E.g. Overload.java
- Automatic type conversions can play a role in overload resolution.
    - E.g. Overload.java

Con’t

- Method overloading supports polymorphism “one interface, multiple methods” paradigm.
- In C, abs() – returns abs of integer
  - labs() – returns abs of long
  - fabs() – returns abs of float
- C does not support overloading, each function has to have it own name and tough 3 functions does same thing.
- In java, Math class handle all numeric types.

- Sqrt – returns square of an int and square root of float.
  - Defeats its original purpose

Overloading Constructors

- OverloadCons.java

Using Objects as Parameters

- PassObj.java
- OverloadCons2.java

Argument Passing

- Call-by-value : copies the value of an argument into the formal parameter of the subroutine.
  - Changes made to the parameter of the subroutine have no effect on the argument.
- Call-by-reference : reference to an argument is passed to the parameter.
  - Changes made to the parameter will affect the argument used to call the subroutine.
- When a primitive type is passed to a method, it is done by use of call-by-value.
- Object is implicitly passed by call-by-reference.
  - CallByValue.java
  - CallByRef.java

Returning Objects

  - RetOb.java

Recursion

- Recursion is the attribute that allows a method to call itself.
- Method that calls itself is said to be recursive.
  - Recursion.java
- When method calls itself, new variables and parameters are allocated storage on the stack.
- Recursive version may execute slowly than the iterative.
  - Coz overhead of additional function calls.
  - Call creates a new copy of variables.
- Adv of Recursion
  - Create clearer and simpler versions of several algorithms.
  - E.g. QuickSort sorting algorithm

Java Modifiers

Access Control Modifiers: default, public , protected, private

- Visible to the package (default). No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).
- Through encapsulation, we can control what parts of a program can access the members of a class.
- By controlling access, we can prevent misuse.
  - AccessTest.java
  - TestStack.java and TestStack1.java

Non-access Modifiers:

- The *static* modifier for creating class methods and variables.
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

Static (variables, methods, blocks and nested classes)

Static variables

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- The static variable gets memory only once in class area at the time of class loading.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static.
- Static variable can be accessed by calling with the class name before creating object of its class.
  - *ClassName.VariableName*
  - *Student.java*

Con’t

Static Methods

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.
  - *ClassName.methodName()*
- Methods declared as static have several restrictions:
  - The static method can not use non static data member or call non-static method directly.
  - this and super cannot be used in static context.
  - StaticByName.java

Con’t

- static block
  - Is used to initialize the static data member.
  - It is executed before main method at the time of classloading.
  - TestStatic.java
- It makes your program **memory efficient** (i.e it saves memory).

Final(variables, methods and class)

- Final variables : cannot change the value once initialized.
  - Constant variables
  - All uppercase identifiers for final variables.
  - Employee.java
- Final method : To prevent derive class or any other class to overriding the method.
- Final class : no class can be derived from that class.
  - To prevent inheritance

Nested and Inner classes

- Classes within another class known as nested classes.
- If class B is defined within class A, then B does not exist independently of A.
  - B can access all members (even private)of A but vice versa is not true
- Two types of nested classes : static and non-static

Non-static nested class

- An inner class is a non-static nested class.
- It has access to all of the variables and methods of its outer class and refer them directly in the same way that other non-static members of the outer class do.

Con’t

- Instance of Inner can be created only within the scope of class Outer.
  - Otherwise, compiler generates an error message.
  - Solution:
  - Create an instance of Inner outside of Outer by qualifying its name with Outer, as Outer.Inner.
  - InnerClassDemo.java
- Inner class within for loop
  - InnerClassDemo1.java

static nested class

- It can access static data members of outer class including private.
- static nested class cannot access non-static (instance) data member or method.
    - Outer.java

String and StringBuffer class

- String as an Object in java.
  - System.out.println(“This is String”);
  - This is String – is String constant as object.
  - String s = “I like Java”
  - String s = “I” + “like” + “Java”;
- TestString.java

```
// Demonstrate String arrays.
class StringDemo3 {
    public static void main(String args[]) {
        String str[] = { "one", "two", "three" };
        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " + str[i]);
    }
}
```

```
str[0]: one
str[1]: two
str[2]: three
```

- StringBuffer1.java
- If your text can change and will only be accessed from multiple threads, use StringBuffer because StringBuffer is synchronous.
- StringBuffer is faster than String.
- **Java SE documentation**

Varargs : Variable-Length Arguments

- A method that takes variable number of arguments is called a variable-arity method or varargs method.
  - VarArgs.java
- Varargs parameter must be last
  - int doIt(int a, int b, double c, int … v) {
- Only one varargs parameter
  - int doIt(int a, int b, double c, int … v, double … m) { //Error
  - VarArgs1.java
- Overloading vararg methods
  - VarArgs2.java
- Varargs and Ambiguity
  - Static void vaTest(int v)
  - Static void vaTest(int v, int … v)
    - vaTest(1)   //ambiguity