SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Object Oriented Programming

Kaustubh Kulkarni
Assistant Professor,
Department of Computer Engineering,
KJSCE.

Principles of Object Oriented Programming

- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Data Abstraction : Motivation

- Client/user perspective
  - Interested in what a program does, not how.
  - Minimize irrelevant details for clarity.
- Server/implementer perspective (Information Hiding)
  - Restrict users from making unwarranted assumptions about the implementation.
  - Reserve right to make changes to the class to improve performance while maintaining the same behavior from the client / user point of view.

Data Abstraction : Examples

- Queues (operations: empty, enQueue, deQueue, isEmpty)
  - array-based implementation
  - linked-list based implementation
- Tables (operations: empty, insert, lookUp, delete, isEmpty)
  - Sorted array based implementation (logarithmic search)
  - Hash-tables based implementation (ideal: constant time search)
  - AVL trees based implementation (height-balanced)
  - B-Trees based implementation (optimized for secondary storage)

Encapsulation

- Encapsulation refers to the creation of self-contained modules (classes) that bind processing functions to its data members.
- The data within each class is kept private.
- Each class defines rules for what is publicly visible and what modifications are allowed.
- ***Enables enforcing data abstraction***

Example :Without Encapsulation


```
/** A class with no encapsulation */
class BadShipping {
  public int weight;
  public String address;

  /* remaining code ommitted ... */
}

class ExploitShipping {
  public static void main (String[] args) {
    BadShipping bad = new BadShipping();
    bad.weight = -3;  // Nothing prevents me from doing this
  }
 }
```

Example :Without Encapsulation

- It's clearly a bad idea to allow people to set the shipping weight to a negative value.
- How can you change this class to prevent problems like this from happening?

Example :Without Encapsulation

- Your only choice is to make the *weight* **private** and write a method that allows the class to set limits on weight.
- But since you have already declared *weight* to be **public**, as soon as you make this 'fix', you break every class that currently uses it (by making an object and accessing *weight* with the dot operator) including those classes that are using *weight* properly!

Solution : Encapsulation

- Make *public accessor and mutator methods* to read and modify the instance variables/ data members / fields respectively.
- Keep instance variables/ data members / fields hidden using a *private* access modifier and force callers to use the accessor and mutator methods to use the instance variables/ data members / fields.

Example with encapsulation


```
/** A class with encapsulation */
class Shipping {
  // minimum shipping weight in oz.
  private static final int MIN_WEIGHT = 1;
  private int weight;

  public int getWeight () {
    return weight;
  }

  public void setWeight (int value) {
    weight = Math.max(MIN_WEIGHT, value);
  }

}

class ExploitShipping {
  public static void main (String[] args) {
    Shipping s = new Shipping();
    s.setWeight(-3);    // weight is set to MIN_WEIGHT
  }
 }
```

Accessor Methods / Getters / Selectors

- They are public methods.
- They do not modify the object's state.
- They return the current value of an attribute.
- Their method names usually start with "get" followed by the attribute name.

Mutator Methods / Setters / Modifiers

- They are public methods.
- These methods allow you to "modify" or "set" the value of an object's attribute.
- Thus they provide a way to change the object's state in a safe way performing some checks to validate the value being assigned. (see setAge() in the example)
- Their return type is void.
- Their method names usually start with "set" followed by the attribute name.

Example


```
class Person {
    private String name;
    private int age;
    // Accessor methods (getters)
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Mutator methods (setters)
    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        if (age >= 0)
            this.age = age;
        else
            System.out.println("Age cannot be negative.");
    }
}
```

Constructor

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called when the object is created, before the **new** operator completes.
- Constructors have no return type, not even void.
- This is because the implicit return type of a class' constructor is the class type itself.

Example : class Box (reworked)


```
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;
 }
}
```

Example continued : main() class

```
class BoxDemo {
 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);
 }
}
```

Example Output


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

Explanation

- As you can see, both mybox1 and mybox2 were initialized by the Box( ) constructor when they were created.
- Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both mybox1 and mybox2 will have the same volume.
- The println( ) statement inside Box( ) is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Default Constructor

- When you allocate an object, you use the following general form:
  - class-var = new classname ( );
- Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called.
- Thus, in the line
  - Box mybox1 = new Box();
- new Box( ) is calling the Box( ) constructor.
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- This is why the preceding line of code worked in earlier version of Box (discussed in previous lecture) that did not define a constructor.
- When using the default constructor, all non-initialized instance variables will have their default values.

Parameterized Constructors

- While the Box( ) constructor in the preceding example does initialize a Box object, it is not very useful—all boxes have the same dimensions.
- What is needed is a way to construct Box objects of various dimensions. The easy solution is to add parameters to the constructor.
- For example, the following version of Box defines a parameterized constructor that sets the dimensions of a box as specified by those parameters.

```
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;
}
}
```

```
class BoxDemo {
 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
```

Explanation

- Each object is initialized as specified in the parameters to its constructor.
- For example, in the following line,
  - `Box mybox1 = new Box(10, 20, 15);`
- The values 10, 20, and 15 are passed to the Box( ) constructor when **new** creates the object.
- Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15, respectively.

Note

- Once we define our own constructor, the default constructor is no longer supplied automatically.
- In the preceding example, we created a parameterized constructor.
- Now if we try to create a new object with:
  - Box mybox = new Box();
- It will result in an error, because zero parameter constructor is not defined by you and default constructor won't be automatically supplied because you created your own parameterized constructor.

```
Box mybox1 = new Box();
^
required: double,double,double
found: no arguments
reason: actual and formal argument lists differ in length
```

Constructor Overloading : Need

(Continuing with example from slides 20 - 25)

- Since Box( ) requires three arguments, it's an error to call it without them.
- This raises some important questions.
  - What if you simply wanted a box and did not care (or know) what its initial dimensions were?
  - Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions?
  - What if you want to construct a new object that is initially the same as some existing object?

```
class Box {
 double width;
 double height;
 double depth;
  // constructor used when all dimensions specified
 Box(double w, double h, double d) {
     width = w;
     height = h;
     depth = d;
 }
```

```
// constructor used when no dimensions specified
 Box() {
      width = -1; // use -1 to indicate
      height = -1; // an uninitialized
      depth = -1; // box
 }
 // constructor used when cube is created
 Box(double len) {
     width = height = depth = len;
 }
```

```
// It clones an object of type Box.
 Box(Box ob) { // pass object to constructor
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
 }
 // compute and return volume
 double volume() {
     return width * height * depth;
 }
} // class Box
```

```
class OverloadConstructor {
 public static void main(String[] args) {
 // create boxes using the various constructors
      Box mybox1 = new Box(10, 20, 15);
      Box mybox2 = new Box();
      Box mycube = new Box(7);
      Box myclone = new Box(mybox1); // create copy
of mybox1
      double vol;
```

```
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of cube is " + vol);
        // get volume of clone
        vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);
    } // main()
} // class OverloadConstructor
```

Questions?