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
```

# Destructor



- Constructors and destructors create and destroy objects of a class, respectively.
- Destructors reverse the construction process.
- They must deallocate memory when appropriate and perform other cleanup activities like closing open file handles.
- In real-time systems, this often means commanding hardware components to known, reasonable states.
- Valves may be closed, hard disks parked, lasers deenergized, and so forth.

# Example C++ code



```
#include <iostream>
using namespace std;
class Employee {
  public:
      Employee() {
      // Constructor is defined.
      cout << "Constructor Invoked for Employee class" << endl;
      }
      ~Employee() {
      // Destructor is defined.
      cout << "Destructor Invoked for Employee class" << endl;
      }
};
int main(void) {
  // Creating an object of Employee, constructor will be called
  Employee emp;
  cout<<"Inside main(), before return 0.\n";
  return 0;
  //emp goes out of scope, destructor will be called
}
```

# Example C++ code output

```
Constructor Invoked for Employee class
Inside main(), before return 0.
Destructor Invoked for Employee class
```

# Destructor in Java : Garbage Collection

- There are no destructors in Java.
- Instead, Java uses a garbage collector to automatically deallocate memory when an object is no longer needed.
- The garbage collector is a program that runs in the background and tracks which objects are still being used and which objects can be deleted.
- In Java, there is no need for destructors because the garbage collector will automatically deallocate memory when an object is no longer needed.

# Garbage Collection



- When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no need to explicitly destroy objects.
- Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.
- Furthermore, different Java run-time implementations will take varying approaches to garbage collection.

# Garbage Collection : finalize()



- Sometimes if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called finalization.
- To add a finalizer to a class, you simply define the finalize( ) method.
- Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.
- The Java runtime calls that method whenever it is about to recycle an object of that class.

# Garbage Collection : System.gc()

```
public static void gc()
```

Request to run the garbage collector.

Invoking the 'gc' method in Java prompts the Virtual Machine to free up memory by recycling unused objects for future use.

When control returns from the method call, one can say that the Java Virtual Machine has made its “best effort” to reclaim space from all discarded objects.

# Example of garbage collection



```
class Employee {
      Employee() {
            System.out.println("Constructor Invoked");
      }
      protected void finalize(){
            System.out.println("Finalizer Invoked");
      }
}
class Main{
 public static void main(String[] args) {
  // Creating an object of Employee, constructor will be called
  Employee emp = new Employee();
  System.out.println("Inside main()");
  emp=null; // makes emp eligible for garbage collection
  System.gc();
}
}
```

# Output

```
Constructor Invoked
Inside main()
Finalizer Invoked
```

# Code Reuse



- One of the most compelling features about Java is code reuse.
- The trick is to use the classes without soiling the existing code. In this chapter you'll see two ways to accomplish this.
- The first is: You simply create objects of your existing class inside the new class.
- This is called **composition** because the new class is composed of objects of existing classes.
- You're simply reusing the functionality of the code, not its form.
- The second approach is: You create a new class as a type of an existing class. You literally take the form of the existing class and add code to it without modifying the existing class. This is called **inheritance**.

# Example of Composition



```
public class Job {
    private String role;
    private long salary;
    private int id;

    public String getRole() {
        return role;
    }
    public void setRole(String role) {
        this.role = role;
    }
    public long getSalary() {
        return salary;
    }
    public void setSalary(long salary) {
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

# Example of Composition (cont'd)

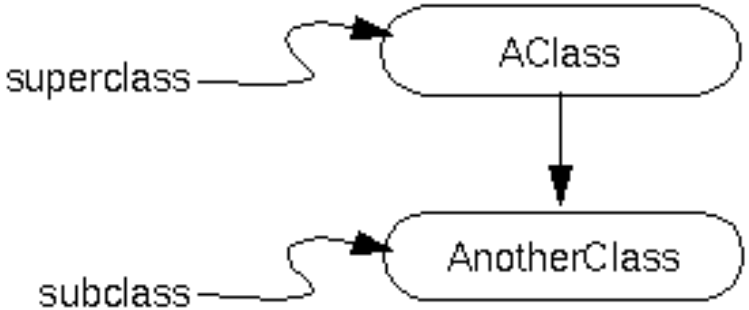


```
public class Person {

    //composition has-a relationship
    private Job job;

    public Person(){
        this.job=new Job();
        job.setSalary(1000L);
    }
    public long getSalary() {
        return job.getSalary();
    }

}
public class TestPerson {

    public static void main(String[] args) {
        Person person = new Person();
        long salary = person.getSalary();
    }

}
```

# Inheritance



- Classes can be derived from other classes.
- The derived class (the class that is derived from another class) is called a subclass or the child class
- The base class from which its derived is called the superclass or the parent class



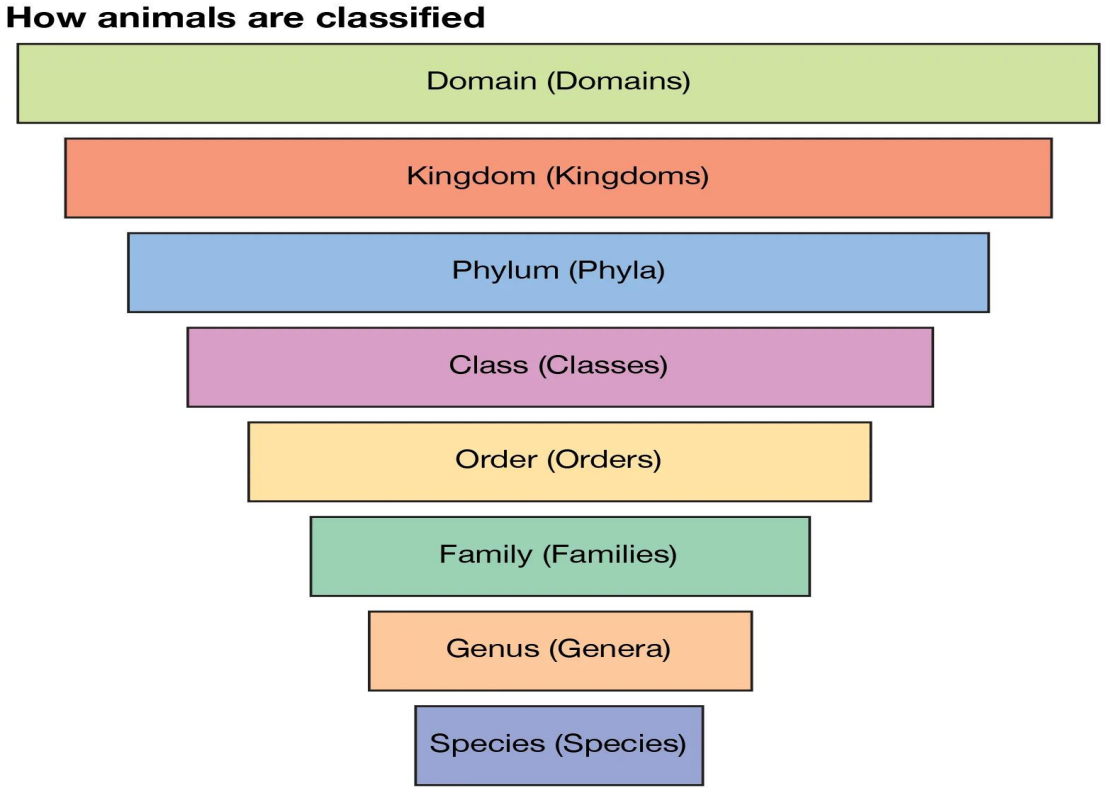
# Inheritance



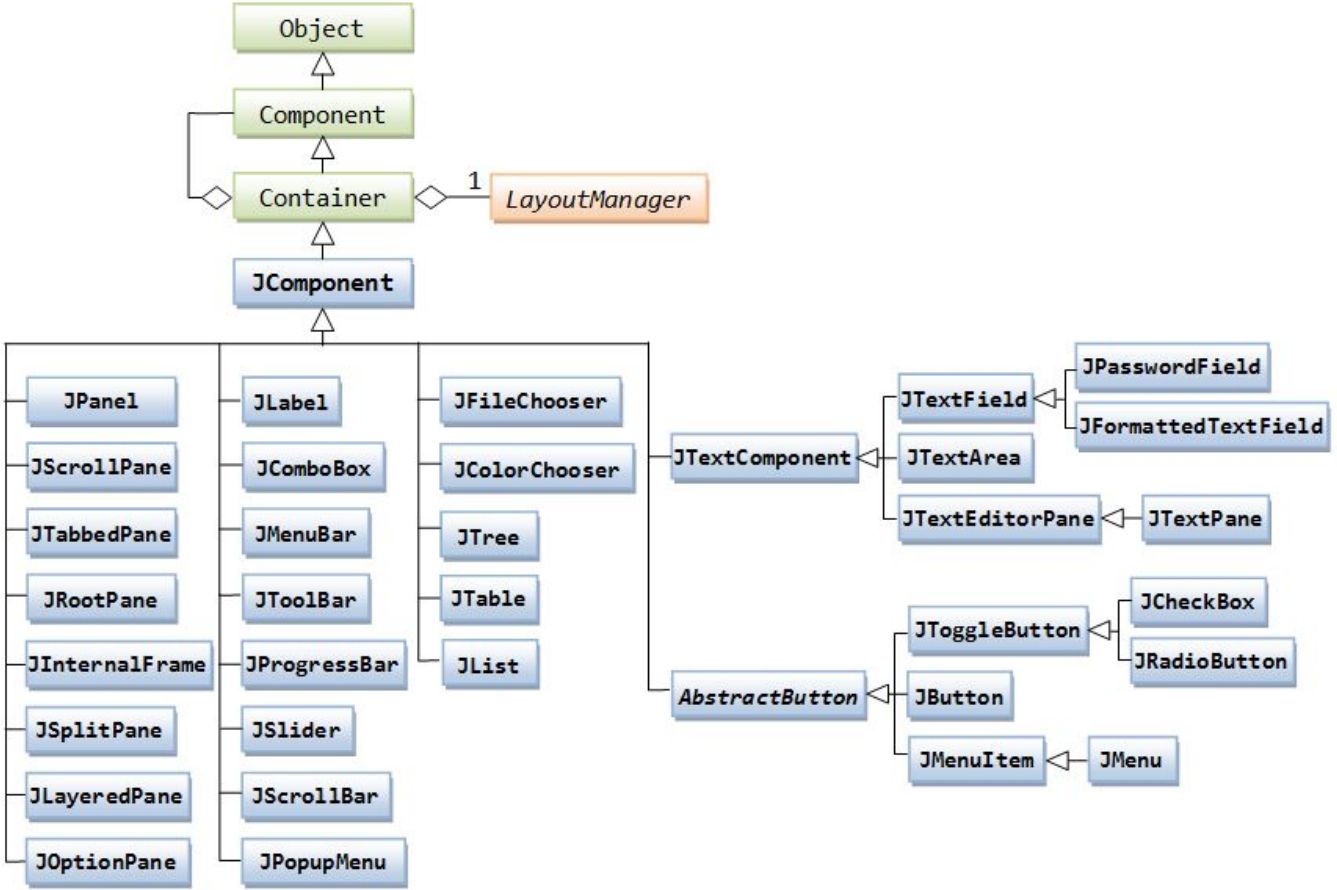
- Inheritance starts with organizing our classes in a hierarchy. Hierarchies are all over the place in the universe.
- A classic example is the "Linnaean taxonomy", from Domain to Kingdom, Phylum, Class, Order, Family, Genus, Species, with the top breaking up into plant, animal, fungi, bacteria, and protista.
- A classic example of this in programming comes in simulations and games, like our Tron Light Cycle game.
- In these cases you might start with a class, PhysicalObject. Underneath that might be StationaryObject and MovableObject. MoveableObject might break into PassiveMoveable and Vehicle.
- Vehicle could then be divided into cars and cycles, etc.
- Another common example are GUI interface objects.
- You might have GUI Objects as a class, and then Windows, Menus, Buttons, and Fields as subclasses.
- Each fields could be broken into display fields and InputFields, and InputFields can be further specialized into PasswordInputFields.

# Linnaean taxonomy





# GUI (Swing) Components



# Photo of Tron Light Cycle



# Inheritance



- The child class inherits all of the data (fields) and all of the methods from the parent.
- This should make some sense: if the child class is a special kind of the parent class (An elephant is a special kind of mammal, a cycle is a kind of vehicle, a button is a kind of GUI widget) then the child should have similar functionality and would likely need the same fields an use the same methods (elephants regulate their temperature, cycles carry passengers, button change GUI behavior).
- For some operations, nothing would be defined in the parent class because all of the child classes handle it differently. Elephants locomote differently from dolphins, so it would not makes sense for them both to inherit the same locomotion method.

# Inheritance



- In fact, in Java, all classes must be derived from some class. Which leads to the question "Where does it all begin?"
- The top-most class, the class from which all other classes are derived, is the **Object** class defined in java.lang.
- **Object** is the root of the hierarchy of classes.

# Inheritance



- An object of a subclass class has the type of the subclass class, and it also has the type of the superclass.
- An object of a derived class can be plugged in as a parameter in place of any of its ancestor classes.
- In a class hierarchy, several methods may have the same name and the same formal parameter list.
- Moreover, a reference variable of a class can refer to either an object of its own class or an object of its subclass.
- Therefore, a reference variable can invoke a method of its own class or of its subclass(es).

# super



- A subclass can refer to the field x of its superclass using the super.x
- A subclass can refer to the method f() of its superclass using the super.f()
- The subclass's constructor can invoke a constructor of its superclass using the super(...) call and supplying 0 or more parameters as required by the superclass constructor.

# Overloading



Java allows different methods to share the same name as long as their parameter lists are different.
This is called overloading.
For example, the following calculator class contains four variants of the overload method:

```
class Calculator {
      int result;
     void add(int x) { result += x; }
     void add(long x) { result += x; }
     void add(float x) { result += x; }
     void add(int x, int y) { result += x % y; }

}
```

The virtual machine performs variant selection by analyzing the number and types of arguments passed to the method:

```
Calculator calc = new Calculator();
calc.add(42);   // call int variant
calc.add(42L);  // call long variant
calc.add(42F);  // call float variant
calc.add('X');  // call int variant
```

# Overloading (2)



A subclass inherits the fields and methods of its superclass. The subclass may declare new methods. Some of these methods may overload inherited methods. For example, assume a simple floating point calculator has been declared:

```
class Calculator {
   double result;
   void add(double arg) { result += arg; }
   // etc.
}
```

We would like to add a mode that allows us to do integer arithmetic.

```
class IntCalculator extends Calculator {
   int result;
   boolean intMode = true;
   void add(int arg) {
      if (intMode) result += arg;
      else super.add(arg);
   }
   // etc.
}
```

# Overloading (3)



Study the result of executing the following test code:

```
IntCalculator calc = new IntCalculator();
calc.add(42F); // calc.super.result = 42.0
calc.add(42); // calc.result = 42
calc.intMode = false;
calc.add(42); // calc.super.result = 84.0
```

# Overloading (4)



Calc has two variants of the add method. The inherited double variant, and the declared integer variant. The compiler chose to call the inherited variant in the line:

```
calc.add(42F);
```

and chose to call the integer variants in the third and fifth lines.

Notice that the integer variant calls the inherited variant if the intMode flag is false. This is done by qualifying the call using super:

```
super.add(arg);
```

Alternatively, since the inherited result field has package scope, the integer variant could have executed:

```
super.result += arg;
```

# Another simple example



```
class OverloadDemo {
 void test() {
    System.out.println("No parameters");
 }
    // Overload test for one integer parameter.
 void test(int a) {
    System.out.println("a: " + a);
 }
    // Overload test for two integer parameters.
 void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
 }
    // Overload test for a double parameter
 double test(double a) {
     System.out.println("double a: " + a);
     return a*a;
 }
}
```

# Another simple example (cont'd)



```
class Overload {
 public static void main(String[] args) {
      OverloadDemo ob = new OverloadDemo();
      double result;
      // call all versions of test()
      ob.test();
      ob.test(10);
      ob.test(10, 20);
      result = ob.test(123.25);
      System.out.println("Result of ob.test(123.25): " + result);
 }
}
```

# Output

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

# 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
```

# Overriding



Let's make a minor change to IntCalculator that we discussed on slide 45. We simply change the parameter of add from int to double:

```
class IntCalculator extends Calculator {
      int result;
      boolean intMode = true;
      void add(double arg) {
            if (intMode) result += arg;
            else super.add(arg);
      }
      // etc.
}
```

# Overriding (2)



Study the following lines of code:

```
IntCalculator calc = new IntCalculator();
calc.add(42F);  // calc.result = 42
calc.add(42);// calc.result = 84
calc.add(3.14); // calc.result = 87
```

Clearly, `IntCalculator.add()` was called all three times. This happens because the new add method has the same signature as the inherited add method and therefore **overrides** or redeclares it rather than overloads it.

# Overloading Vs Overriding



**Overriding:** Declaring a method in a subclass with the same signature as a method in its superclass.

**Overloading:** Declaring a method with the same name, different parameters.

# Polymorphism



- Definition: Polymorphism = A basic principle of OOD defined as the ability to use the same expressions to denote different operations (in other words, the ability to associate multiple meanings to the same method name. ).
- Polymorphism is a word of Greek origin that means having multiple forms.

# Binding



- **Definition: Binding** = Associating a method definition with its invocation.
- In **early binding**, a method's definition is associated with its invocation when the code is compiled.
- In **late binding**, a method's definition is associated with the method's invocation at execution time. In late binding the method invoked is determined by the type of object to which the variable refers, NOT by the type of the reference variable.

.

# Binding



Java uses late binding for all methods except for:

- `final` methods (If a method of a class is declared `final`, it cannot be overridden with a new definition in a derived class.)
- `private` methods
- `static` methods.

# Binding Example



```
class Animal {
  public String eats() {return "???";}
  public String name() {return "Animal";}
}

class Herbivore extends Animal {
   public String eats() {return "plants";}
   public String name() {return "Herbivore";}
}

class Ruminants extends Herbivore {
   public String eats() {return "grass";}
   public String name() {return "Ruminant";}
}

class Carnivore extends Animal {
   public String eats() {return "meat";}
   public String name() {return "Carnivore";}
}
```

# Binding Example continued



```
public class Main {

  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[] args) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminants r = new Ruminants();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());
     show(paa.name(), paa.eats());
     show(h.name(), h.eats());
     show(pah.name(), pah.eats());
     show(par.name(), par.eats());
  }
}
```

# Binding Example Output



```
public static void main (String[] args) {
    Animal a = new Animal();
    Herbivore h = new Herbivore();
    Ruminants r = new Ruminants();

    Animal paa = a;
    Animal pah = h;
    Animal par = r;

    show(a.name(), a.eats());
    show(paa.name(), paa.eats());
    show(h.name(), h.eats());
    show(pah.name(), pah.eats());
    show(par.name(), par.eats());
}
}
```

```
Animal ???
Animal ???
Herbivore plants
Herbivore plants
Ruminant grass
```

# Binding Example Output Explained

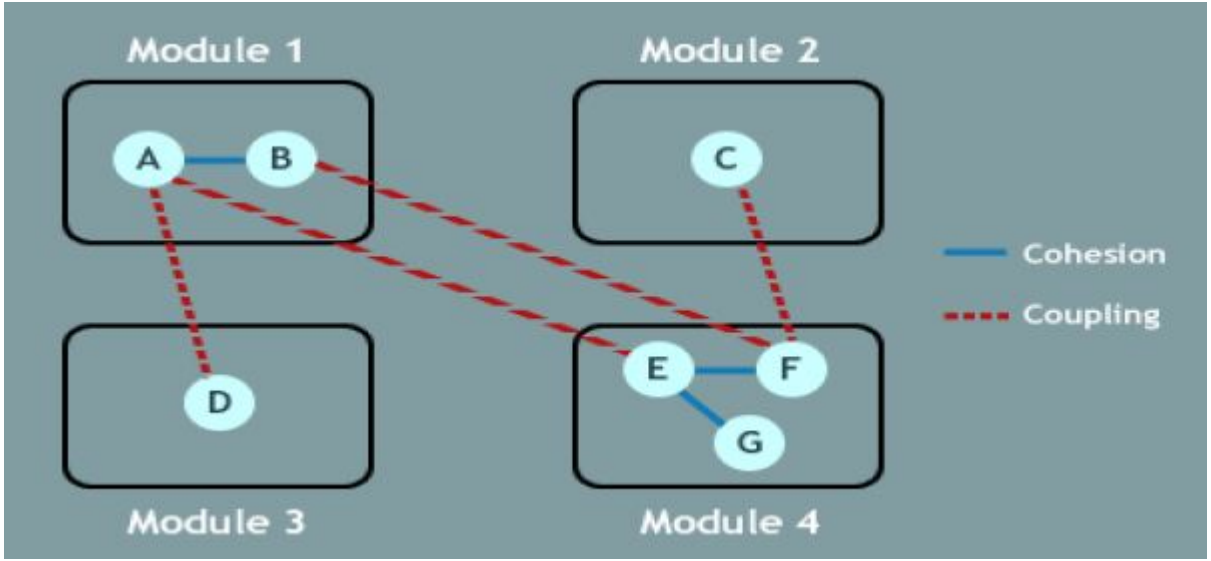
| | |
|---|---|
| `show(a.name(), a.eats());` | So, at runtime, the program follows `a` out to the heap, discovers that it actually points to an `Animal`, and invokes the `Animal.name()` function body to print “Animal”. Then it does the same for `eats()` and invokes the `Animal.eats()` function body to print “???”. |
| `show(paa.name(),paa.eats());` | `paa` is treated just like `a` because, after all, they are both pointers and pointing to the same object. |
| `show(h.name(), h.eats);` | At runtime, the program follows `h` out to the heap, discovers that it actually points to an `Herbivore`, and invokes the `Herbivore.name()` function body to print “Herbivore”. Then it does the same for `eats()` and invokes the `Herbivore.eats()` function body to print “plants”. |
| `show(pah.name(),pah.eats());` | At runtime, the program follows `pah` out to the heap, discovers that it actually points to an `Herbivore`, and invokes the `Herbivore.name()` function body to print “Herbivore”. Then it does the same for `eats()` and invokes the `Herbivore.eats()` function body to print “plants”. |
| `show(par.name(),par.eats());` | At runtime, the program follows `par` out to the heap, discovers that it actually points to a `Ruminant`, and invokes the `Ruminant::name()` function body to print “Ruminant”. Then it does the same for `eats()` and invokes the `Ruminant::eats()` function body to print “grass”. |

# Cohesion and Coupling



- **What is Cohesion?**
  - It refers to the degree to which the elements of a module are related to each other and work together to achieve a common purpose.
  - A module is cohesive if the members of the module (i.e., the public functions, variables, classes, etc.) are related.
  - A highly cohesive module or component contains elements that are closely related and have a single, well-defined responsibility.
- **What is Coupling?**
  - It refers to the degree to which one module depends on another.
  - A loosely coupled system has minimal dependencies between its modules, while a tightly coupled system has many dependencies.
  - A system with high coupling can be more difficult to modify and maintain, as changes in one module can have unexpected effects on other modules.

# Illustration: Cohesion & Coupling



# instanceof



- **instanceof** is a binary operator we use to test if an object is of a given type.
- If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true.
- It's also known as a type comparison operator because it compares the instance with the type.
- The instanceof operation has the syntax:

  <object> instanceof <class-name>

```
public class Test {

   public static void main(String args[]) {

      String name = "James";

         // following will return true since name is type of String

      boolean result = name instanceof String;

      System.out.println( result );

   }

}
```

```
class Vehicle {}
public class Car extends Vehicle {
    public static void main(String args[]) {
      Vehicle a = new Car();
      System.out.println(a instanceof Car); // true
      System.out.println(a instanceof Vehicle); // true
      Car b = new Car();
      System.out.println(b instanceof Car); // true
      System.out.println(b instanceof Vehicle); // true
      Vehicle c = new Vehicle();
      System.out.println(c instanceof Car); // false
      System.out.println(c instanceof Vehicle); // true
      /*Car d = new Vehicle();
      error: incompatible types:
      Vehicle cannot be converted to Car*/
    }
}
```

# *Questions?*