5.2: Features of Object orientation

IT1406 - Introduction to Programming

Level I - Semester 1

5.2. Features of Object orientation

Abstraction

- An essential element of object-oriented programming is *abstraction*.
- A powerful way to manage abstraction is through the use of hierarchical classifications.
- This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.

5.2. Features of Object orientation

Abstraction

- Hierarchical abstractions of complex systems can also be applied to computer programs.
- The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to do something.
- This is the essence of object-oriented programming.

5.2. Features of Object orientation

The Three OOP Principles

- All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are
  - Encapsulation
  - Inheritance
  - polymorphism

5.2. Features of Object orientation

Encapsulation

- *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

5.2. Features of Object orientation

Encapsulation

- Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class.

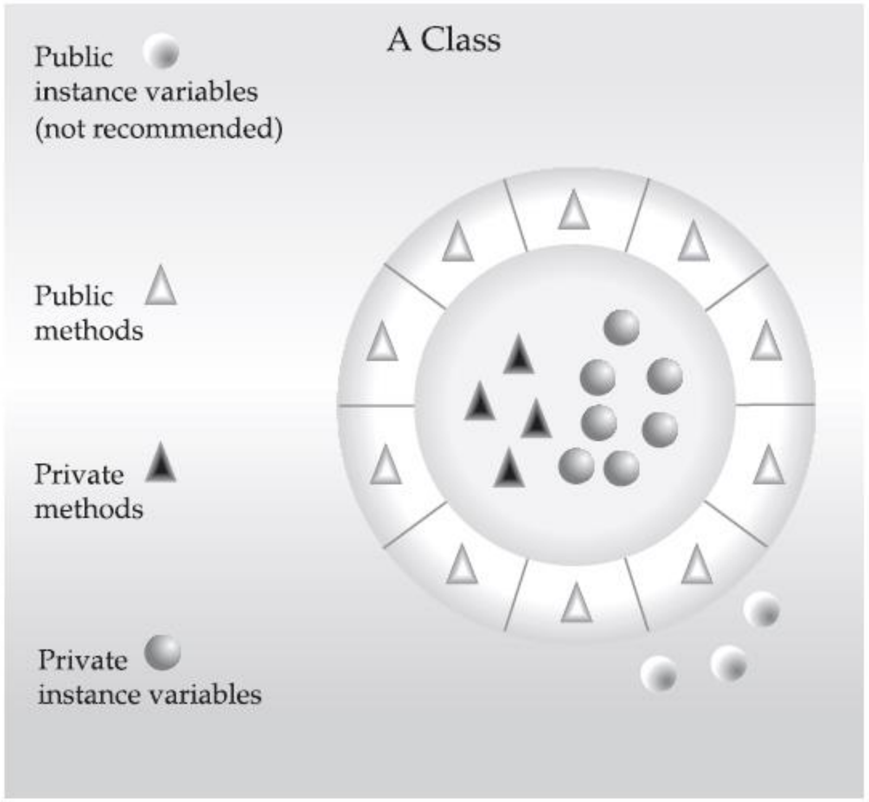
5.2. Features of Object orientation

Inheritance

- *Inheritance* is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.
- For example, a Golden Retriever is part of the classification dog, which in turn is part of the *mammal* class, which is under the larger class *animal*. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

5.2. Features of Object orientation

Inheritance


A Class
Public instance variables (not recommended)
Public methods
Private methods
Private instance variables

Encapsulation: public methods can be used to protect private data.

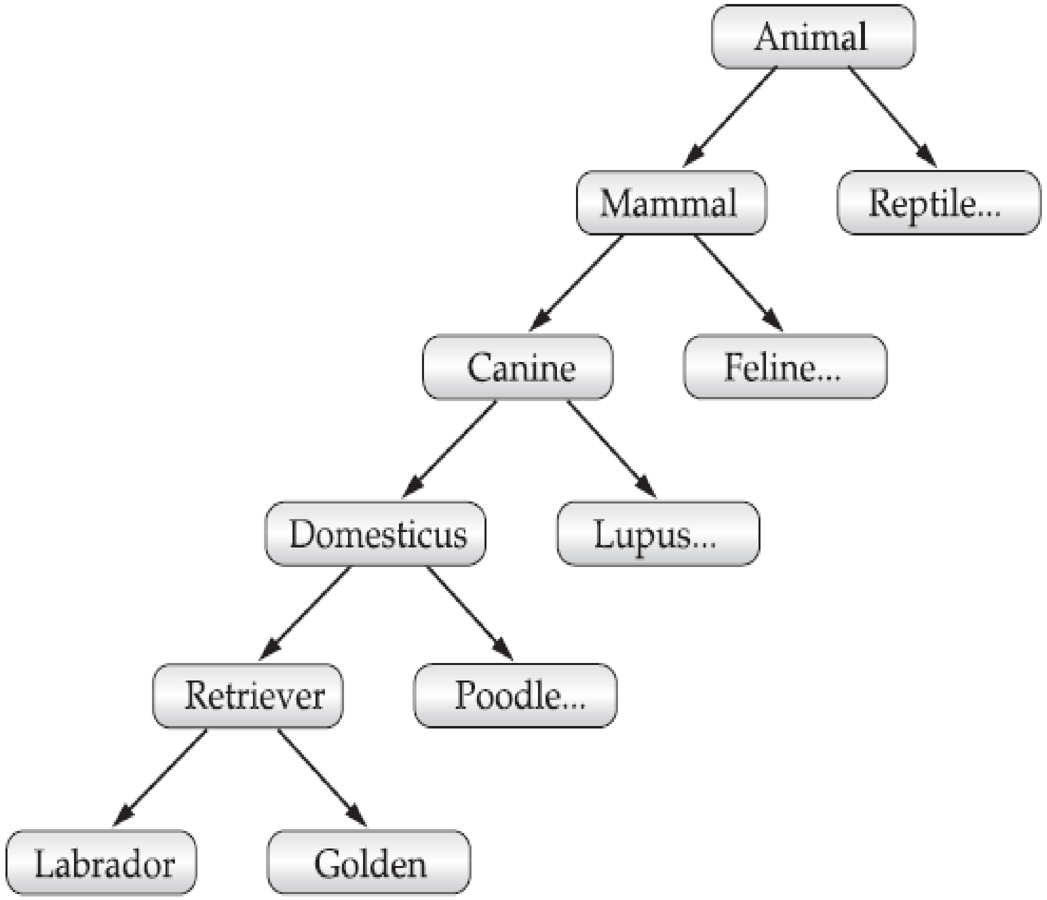
5.2. Features of Object orientation

Inheritance

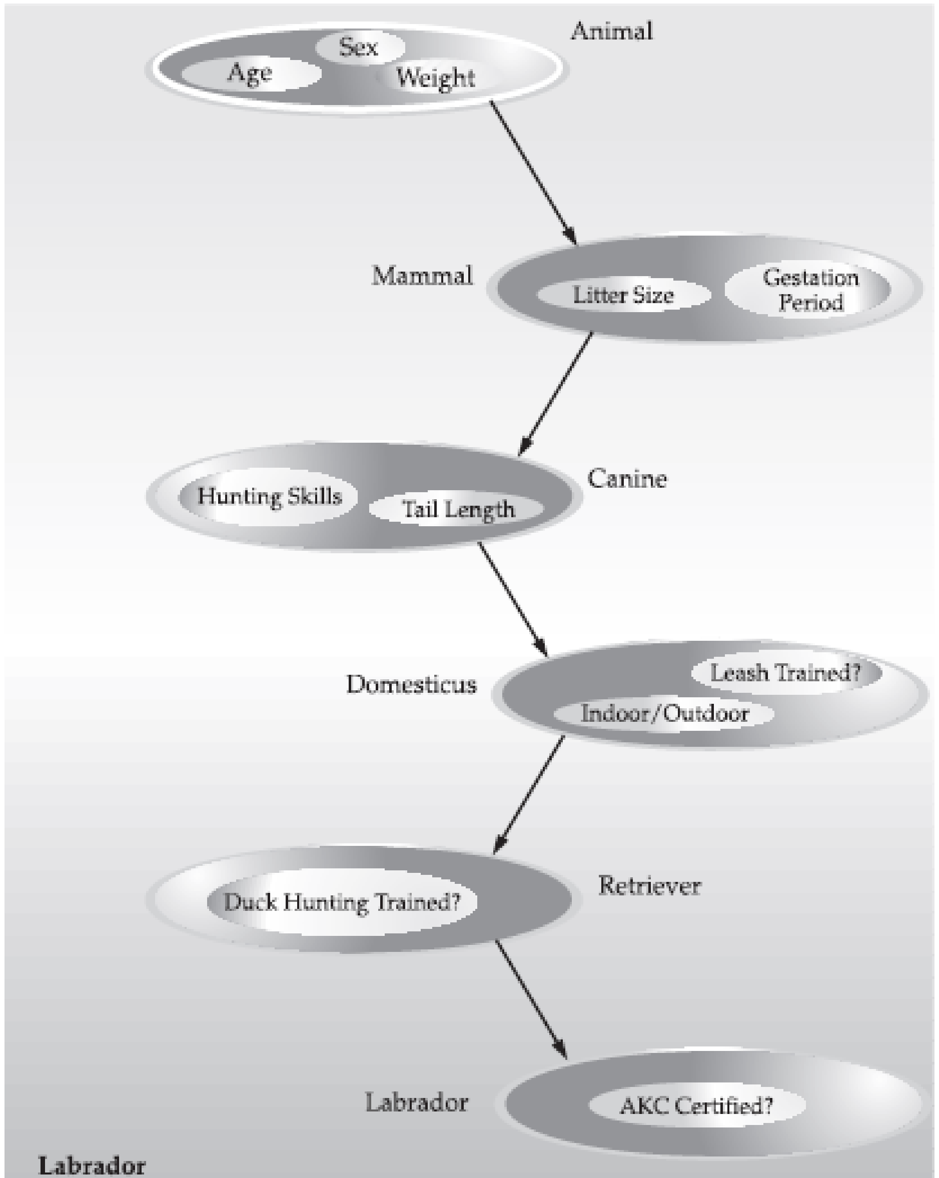
- Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization. This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically.
- A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

5.2. Features of Object orientation

Inheritance



Inheritance



Labrador

Age	**Gestation Period**	**Leash Trained?**
Sex	**Hunting Skills**	**Duck Hunting Trained?**
Weight	**Tail Length?**	**AKC Certified?**
Litter Size	**Indoor / Outdoor?**	

Labrador inherits the encapsulation of all its superclasses

5.2. Features of Object orientation

Inheritance Basics

- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.
// Create a superclass.
class A {
    int i, j;
    void showij() {
        System.out.println("i and j: " + i + " " + j);
    }
}
// Create a subclass by extending class A.
class B extends A {
    int k;
    void showk() {
      System.out.println("k: " + k);
    }
    void sum() {
      System.out.println("i+j+k: " +
      (i+j+k));
    }
}
class SimpleInheritance {
    public static void main(String args
    []) {
      A superOb = new A();
      B subOb = new B();
      superOb.i = 10;
      superOb.j = 20;
      System.out.println("Contents of
      superOb: ");
      superOb.showij();
      System.out.println();
      // The superclass may be used by
      itself.
      /* The subclass has access to all
      public members of its superclass. */
      subOb.i = 7;
      subOb.j = 8;
      subOb.k = 9;
      System.out.println("Contents of
      subOb: ");
      subOb.showij();
      subOb.showk();
      System.out.println();
      System.out.println("Sum of i, j and
      k in subOb:");
      subOb.sum();
    }
}
```

5.2. Features of Object orientation

- The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

- As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**.
- Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

5.2. Features of Object orientation

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.
- A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.
- A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses. Each subclass can precisely tailor its own classification.

5.2. Features of Object orientation

Member Access and Inheritance

- For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box {
    int color; // color of box
    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}
```

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

5.2. Features of Object orientation

- Example,

```
class RefDemo {
        public static void main(String args[]) {
                BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
                Box plainbox = new Box();
                double vol;
                vol = weightbox.volume();
                System.out.println("Volume of weightbox is " + vol);
                System.out.println("Weight of weightbox is " + weightbox.weight);
                System.out.println();
                // assign BoxWeight reference to Box reference
                plainbox = weightbox;
                vol = plainbox.volume(); // OK, volume() defined in Box
                System.out.println("Volume of plainbox is " + vol);
                // The following statement is invalid because plainbox does not define a weight member.
                // System.out.println("Weight of plainbox is " + plainbox.weight);
        }
}
```

5.2. Features of Object orientation

- Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.

5.2. Features of Object orientation

Using super

super has two general forms.

- The first calls the superclass’ constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

5.2. Features of Object orientation

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of **super**:

```
super(arg-list);
```

- Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

5.2. Features of Object orientation

Using super to Call Superclass Constructors

- To see how **super( )** is used, consider this improved version of the **BoxWeight** class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
    double weight; // weight of box

    // initialize width, height, and depth using super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
}
```

5.2. Features of Object orientation

- Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box** constructor to be called, which initializes **width**, **height**, and **depth** using these values.
- **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.
- In the preceding example, **super( )** was called with three arguments. Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments.

5.2. Features of Object orientation

A Second Use for super

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

  super.member

- Here, *member* can be either a method or an instance variable.
- This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
class A {
   int i;
}
// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the i in A
    B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
}
void show() {
   System.out.println("i in superclass: " + super.i);
   System.out.println("i in subclass: " + i);
   }
}
class UseSuper {
   public static void main(String args[]) {
      B subOb = new B(1, 2);
      subOb.show();
}
}
```

5.2. Features of Object orientation

A Second Use for super

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

- Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

5.2. Features of Object orientation

Creating a Multilevel Hierarchy

- Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.
- For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**.

5.2. Features of Object orientation

When Constructors Are Executed

- When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}
// Create a subclass by extending class A.
class B extends A {
   B() {
      System.out.println("Inside B's constructor.");
   }
}
// Create another subclass by extending B.
class C extends B {
   C() {
      System.out.println("Inside C's constructor.");
   }
}
class CallingCons {
   public static void main(String args[]) {
      C c = new C();
   }
}
```

5.2. Features of Object orientation

- The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

- The constructors are executed in order of derivation.
- If you think about it, it makes sense that constructors complete their execution in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

5.2. Features of Object orientation

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // display k – this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}
```

5.2. Features of Object orientation

```
class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);
    subOb.show(); // this calls show() in B
  }
}
```

- The output produced by this program is shown here:

```
        k: 3
```

- When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.

5.2. Features of Object orientation

- If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    void show() {
        super.show(); // this calls A's show()
        System.out.println("k: " + k);
    }
}
```

5.2. Features of Object orientation

- If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

- Here, **super.show( )** calls the superclass version of **show( )**.

5.2. Features of Object orientation

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

```
// Dynamic Method Dispatch
class A {
        void callme() {
                System.out.println("Inside A's callme method");
        }
}
class B extends A {
        // override callme()
        void callme() {
                System.out.println("Inside B's callme method");
        }
}
class C extends A {
        // override callme()
        void callme() {
            System.out.println("Inside C's callme method");
        }
}
class Dispatch {
        public static void main(String args[]) {
            A a = new A(); // object of type A
            B b = new B(); // object of type B
            C c = new C(); // object of type C
            A r; // obtain a reference of type A
            r = a; // r refers to an A object
            r.callme(); // calls A's version of callme
            r = b; // r refers to a B object
            r.callme(); // calls B's version of callme
            r = c; // r refers to a C object
            r.callme(); // calls C's version of callme
        }
}
```

5.2. Features of Object orientation

- The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

- This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared.
- Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**.

5.2. Features of Object orientation

Using Abstract Classes

- You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. In the class **Triangle** it has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods.
- Java's solution to this problem is the *abstract method*.
- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass.
- To declare an abstract method, use this general form:

abstract *type name(parameter-list);*

Example:

```
abstract class A {
        abstract void callme();
        // concrete methods are still allowed in abstract classes
        void callmetoo() {
              System.out.println("This is a concrete method.");
        }
}
class B extends A {
        void callme() {
              System.out.println("B's implementation of callme.");
        }
}
class AbstractDemo {
        public static void main(String args[]) {
               B b = new B();
               b.callme();
               b.callmetoo();
        }
}
```

5.2. Features of Object orientation

Using Abstract Classes

- Abstract classes can include as much implementation as they see fit.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java’s approach to run-time polymorphism is implemented through the use of superclass references.
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

5.2. Features of Object orientation

Using final with Inheritance

Using final to Prevent Overriding

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
- Methods declared as **final** cannot be overridden.
- The following fragment illustrates **final**:

5.2. Features of Object orientation

```
class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}
class B extends A {
   void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

- Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.
- Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

5.2. Features of Object orientation

Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- Here is an example of a **final** class:

5.2. Features of Object orientation

Using final to Prevent Inheritance

```
final class A {
    //...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    //...
}
```

- As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

5.2. Features of Object orientation

The Object Class

- There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**.
- That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.
- **Object** defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

5.2. Features of Object orientation

- The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**.
- The **equals( )** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise.
- The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println( )**.

5.2. Features of Object orientation

Overloading Vararg Methods

- You can overload a method that takes a variable-length argument. For example, the following program overloads **vaTest( )** three times:

```
// Varargs and overloading.
class VarArgs3 {
    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...):"+"Number of args:"+
        v.length       +“Contents:");
    for(int x : v)
        System.out.print(x + " ");
    System.out.println();
   }
  static void vaTest(boolean ... v) {
  System.out.print("vaTest(boolean ...)"+"Number of
  args:"+v.length +"Contents:");
```

```
    for(boolean x : v)
        System.out.print(x + " ");
     System.out.println();
    }
static void vaTest(String msg, int ... v) {
      System.out.print("vaTest(String, int ...): " +msg + v.length +"
      Contents: ");
      for(int x : v)
          System.out.print(x + " ");
      System.out.println();
     }
public static void main(String args[]){
      vaTest(1, 2, 3);
      vaTest("Testing: ", 10, 20);
      vaTest(true, false, false);
      }
}
```

5.2. Features of Object orientation

- The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false
```

- This program illustrates both ways that a varargs method can be overloaded. First, the types of its vararg parameter can differ. This is the case for **vaTest(int ...)** and **vaTest(boolean ...)**.

5.2. Features of Object orientation

Polymorphism

- *Polymorphism* (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.
- More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*.

5.2. Features of Object orientation

Polymorphism, Encapsulation, and Inheritance Work Together

- When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scalable programs than does the process-oriented model.
- Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes.
- Polymorphism allows you to create clean, sensible, readable, and resilient code.