Polymorphism

Visibility Revisited

- All variables and methods of a parent class, even private members, are inherited by its children
- As we've mentioned, private members cannot be referenced by name in the child class
- However, private members inherited by child classes exist and can be referenced indirectly
  - Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods
  - The `super` reference can be used to refer to the parent class, even if no object of the parent class exists

Parent-child class relationship

- An example of `super` reference

```java
public class Superclass {
    public void printMethod() {
        System.out.println("Printed In Superclass.");
    }
}

class Subclass extends Superclass {
    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed In Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Output:

- Printed in Superclass
- Printed in Subclass
Designing for Inheritance

- Taking the time to create a good software design can ensure long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software
- Let's summarize some of the issues regarding inheritance that relate to a good software design

Inheritance Design Issues

- Every derivation from the main class should be an is-a relationship
- Think about a potential future class hierarchy
- Design classes to be reusable and flexible
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate, i.e. “generalize” the behavior
- Override methods as appropriate to tailor or change the functionality of a child
- Add new variables to children, but don't redefine (shadow) inherited variables

Inheritance Design Issues

- An example class hierarchy

```
Employee
   
Temporary Employee
  
Permanent Employee
  
Consultant
  
Manager
  
Director
  
Consultant Manager
  
Permanent Manager
```

- Allow each class to manage its own data; use the super reference to invoke the parent's constructor to set up its data
- Even if there are no current uses for them, override general methods such as toString and equals with appropriate definitions
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation
Restricting Inheritance

- The `final` modifier can be used to cut down inheritance
  - If the `final` modifier is applied to a method, then that method cannot be overridden in any descendent classes
  - If the `final` modifier is applied to an entire class, then that class cannot be used to derive any children at all

- These are key design decisions and establish that a method or class must be used “as is” or not at all

Class Object

- In Java, all types of objects have a superclass named `Object`.
  - Every class implicitly extends `Object`.
- The `Object` class defines several methods:
  - `public String toString()`
    - Used to print the object.
  - `public boolean equals(Object other)`
    - Compare the object to any other for equality.

Comparing objects

- The `==` operator does not work well with objects.
  - `==` compares references to objects, not their contents or state.
- Example:
  ```java
  Point p1 = new Point(5, 3);
  Point p2 = new Point(5, 3);
  if (p1 == p2) { // false
    System.out.println("equal");
  }
  ```
The equals() method

- The **equals** method compares the contents / state of objects.
  - equals should be used when comparing Strings, Points, ...

  ```java
  if (str1.equals(str2)) {
      System.out.println("the strings are equal");
  }
  ```

- If you write your own class, its equals method will behave just like the == operator.

  ```java
  Point p1 = new Point(5, 3);
  Point p2 = new Point(5, 3);
  if (p1.equals(p2)) { // false
      System.out.println("equal");
  }
  ```

  - This is the behavior we inherit from class Object.

Initial flawed equals() method

- We can change this behavior by writing an equals method.
  - Ours will **override** the default behavior from class Object.
  - The method should compare the state of the two objects and return true for cases like the above.

- A flawed implementation of the equals method:

  ```java
  public boolean equals(Point other) {
      if (x == other.x && y == other.y) {
          return true;
      } else {
          return false;
      }
  }
  ```

Flaws in equals() method

- It should be legal to compare a Point to any object (not just other Point objects):

  ```java
  // this should be allowed
  Point p = new Point(7, 2);
  if (p.equals("hello")) { // false
      ...
  }
  ```

  - equals should always return false if a non-Point is passed.

equals() and the Object class

- equals() method, general syntax:

  ```java
  public boolean equals(Object <name>) {<statement(s) that return a boolean value> ;
  }
  ```

  - The parameter to equals must be of type Object.
  - Object is a general type that can match any object.
  - Having an Object parameter means any object can be passed.
Another flawed version

- Another flawed equals implementation:
  ```java
  public boolean equals(Object o) {
      return x == o.x && y == o.y;
  }
  ```

- It does not compile:
  ```java
  Point.java:36: cannot find symbol
  symbol  : variable x
  location: class java.lang.Object
  return x == o.x && y == o.y;
  ^
  ```

  The compiler is saying, 
  "o could be any object. Not every object has an x field."

Type-casting objects

- Solution: Type-cast the object parameter to a Point.
  ```java
  public boolean equals(Object o) {
      Point other = (Point) o;
      return x == other.x & & y == other.y;
  }
  ```

- Casting objects is different than casting primitives.
  - We’re really casting an Object reference into a Point reference.
  - We’re promising the compiler that o refers to a Point object.

Casting objects diagram

- Client code:
  ```java
  Point p1 = new Point(5, 3);
  Point p2 = new Point(5, 3);
  if (p1.equals(p2)) {
      System.out.println("equal");
  }
  ```

Comparing different types

- When we compare Point objects to other types:
  ```java
  Point p = new Point(7, 2);
  if (p.equals("hello")) { // should be false
      ...
  }
  ```

  Currently the code crashes:
  ```java
  Exception in thread "main"
  java.lang.ClassCastException: java.lang.String
  at Point.equals(Point.java:25)
  at PointMain.main(PointMain.java:25)
  ```

  The culprit is the line with the type-cast:
  ```java
  public boolean equals(Object o) {
      Point other = (Point) o;
  ```
The `instanceof` keyword

- We can use a keyword called `instanceof` to ask whether a variable refers to an object of a given type.

- The `instanceof` keyword, general syntax:
  
  `<variable> instanceof <type>`

- The above is a boolean expression.

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s instanceof Point</td>
<td>false</td>
</tr>
<tr>
<td>s instanceof String</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof Point</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof String</td>
<td>false</td>
</tr>
<tr>
<td>null instanceof String</td>
<td>false</td>
</tr>
</tbody>
</table>

Example:

```java
String s = "hello";
Point p = new Point();
```

Final version of `equals` method

```java
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
```

- This version correctly compares `Point`s to any type of object.

Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph) : "having many forms"

- Enables you to “program in the general” rather than “program in the specific.”

- Polymorphism enables you to write programs that process objects that share the same superclass as if they’re all objects of the superclass; this can simplify programming.
Polymorphism

- A **polymorphic reference** is a variable that can refer to different types of objects at different points in time.

- All object references in Java are potentially polymorphic and can refer to an object of any type compatible with its defined type.

- Compatibility of class types can be based on either Inheritance or Interfaces (which we will see later).

A Polymorphic Example

```java
Dog myDog;
myDog = new Dog();

Animal myAnimal;
myAnimal = myDog;
```

An Example Class Hierarchy

Everything is an Object!

- When we say:
  ```java
  myDog = new Dog();
  ```
- the Dog constructor gets called.
- It, in turn, must call the Animal constructor.
- When you don’t extend anything, by default you extend Object.
- Thus the Animal constructor calls the Object constructor.
- Looking at an object in memory it will look like something like this:
Polymorphism Explained

- The rule is very simple
- A reference can refer to an object which is either
  - The same type as the reference
  - Has a superclass of the same type as the reference
- So all of the following are legal
  - Dog d = new Dog();
  - Animal a = new Animal();
  - Object o = new Object();

An Illegal Example

- We are able to assign an object of a sub-class into an object of a super-class as in:
  Animal MyAnimal = new Dog();
- But the reverse is not true. We can’t assign a superclass object into a sub-class object.
  Dog MyDog = new Animal(); // illegal

Polymorphism Examples

- Example: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes Fish, Frog and Bird represent the three types of animals under investigation.
  - Each class extends superclass Animal, which contains a method move and maintains an animal’s current location as x-y coordinates. Each subclass implements method move.
  - A program maintains an Animal array containing references to objects of the various Animal subclasses. To simulate the animals’ movements, the program sends each object the same message once per second—namely, move.
Polymorphism Examples

- Each specific type of Animal responds to a move message in a unique way:
  - a Fish might swim three meters
  - a Frog might jump five meters
  - a Bird might fly ten meters.

- The program issues the same message (i.e., move) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to “do the right thing” in response to the same method call is the key concept of polymorphism.
- The same message sent to a variety of objects has “many forms” of results—hence the term polymorphism.

Polymorphism Examples (Cont.)

- A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system’s code.
- To add new objects to our video game:
  - Build a class that extends SpaceObject and provides its own draw method implementation.
  - When objects of that class appear in the SpaceObject collection, the screen manager code invokes method draw, exactly as it does for every other object in the collection, regardless of its type.
  - So the new objects simply “plug right in” without any modification of the screen manager code by the programmer.

Polymorphism Examples (Cont.)

- Example: Space Objects in a Video Game
  - A video game manipulates objects of classes Martian, Venusian, Plutonian, SpaceShip and LaserBeam. Each inherits from SpaceObject and overrides its draw method.
  - A screen manager maintains a collection of references to objects of the various classes and periodically sends each object the same message—namely, draw.
  - Each object responds in a unique way.
    - A Martian object might draw itself in red with green eyes and the appropriate number of antennae.
    - A SpaceShip object might draw itself as a bright silver flying saucer.
    - A LaserBeam object might draw itself as a bright red beam across the screen.

The same message (in this case, draw) sent to a variety of objects has “many forms” of results.

Demonstrating Polymorphic Behavior

- A superclass object cannot be treated as a subclass object, because a superclass object is not an object of any of its subclasses.
- The is-a relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler does allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.
Demonstrating Polymorphic Behavior (Cont.)

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.
  - The Java compiler allows this “crossover” because an object of a subclass is an object of its superclass (but not vice versa).
- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable’s class type.
  - If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use.
  - This process is called dynamic binding.

Method Calls and Polymorphism

Assume the Dog class extends the Animal class, redefining the “makeNoise” method.

Consider the following:

```java
Animal myAnimal = new Dog();
myAnimal.makeNoise();
```

Note: The Animal reference is referring to a Dog object. And it is the Dog’s makeNoise method that gets invoked!

Dynamic Binding

- Very simple rule.
  - No matter what the reference type is, Java will search the object and execute the lowest occurrence of a method it finds.
- class Object has a toString method
- Assume that both Animal and Dog have overridden the toString method

```
Object o
Animal a
Dog d

o.toString();
a.toString();
d.toString();
```

Polymorphism

- With polymorphism, we can design and implement systems that are easily extensible
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy.
  - The only parts of a program that must be altered for new classes are those that require direct knowledge of the new classes.
Polymorphism

- A variable of a type T can legally refer to an object of any subclass of T.

```java
Employee person = new Lawyer();
System.out.println(person.getSalary()); // 50000.0
System.out.println(person.getVacationForm()); // pink
```

- You can call any methods from Employee on the variable person, but not any methods specific to Lawyer (such as Sue).

- Once a method is called on the object, it behaves in its normal way (as a Lawyer, not as a normal Employee).

Polymorphism + parameters

- You can declare methods to accept superclass types as parameters, then pass a parameter of any subtype.

```java
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer lisa = new Lawyer();
        Secretary steve = new Secretary();
        printInfo(lisa);
        printInfo(steve);
    }
    public static void printInfo(Employee empl) {
        System.out.println("salary = " + empl.getSalary());
        System.out.println("days = " + empl.getVacationDays());
        System.out.println("form = " + empl.getVacationForm());
        System.out.println();
    }
}

OUTPUT:
salary = 50000.0
vacation days = 21
vacation form = pink
salary = 50000.0
vacation days = 10
vacation form = yellow
```

Polymorphism vs. Inheritance

- Inheritance is required in order to achieve polymorphism (we must have class hierarchies).
  - Re-using class definitions via extension and redefinition

- Polymorphism is not required in order to achieve inheritance.
  - An object of class A acts as an object of class B (an ancestor to A).
References and Inheritance

- Assigning a child object to a parent reference is considered to be a *widening conversion*, and can be performed by simple assignment
  - The widening conversion is the most useful

- Assigning a parent object to a child reference can be done, but it is considered a *narrowing conversion* and two rules/guidelines apply:
  - A narrowing conversion must be done with a cast
  - A narrowing conversion should only be used to restore an object back to its original class (back to what it was “born as” with the new operator)

Example

- Given:
  
  ```java
  Person[] people = new Person[4];
  people[0] = new Student("DeBanque, Robin", 8812);
  ```

- When invoking:
  ```java
  people[0].writeOutput();
  ```

- Which `writeOutput()` is invoked, the one defined for `Student` or the one defined for `Person`?

- Answer: The one defined for `Student`

Polymorphism Example

- Consider an array of `Person`
  ```java
  Person[] people = new Person[4];
  ```

- Since `Student` and `Undergraduate` are types of `Person`, we can assign them to `Person` variables
  ```java
  people[0] = new Student("DeBanque, Robin", 8812);
  ```

  ```java
  people[1] = new Undergraduate("Cotty, Manny", 8812, 1);
  ```

Example

```java
public class PolymorphismDemo {
  public static void main(String[] args) {
    Person[] people = new Person[4];
    people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
    people[1] = new Undergraduate("Kick, Anita", 9931, 2);
    people[2] = new Student("DeBanque, Robin", 8812);
    people[3] = new Undergraduate("Bugg, June", 9901, 4);
    for (Person p : people) {
      p.writeOutput();
      System.out.println();
    }
  }
}
```
A polymorphism problem

- Assume that the following four classes have been declared:

```java
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }
    public void method2() {
        System.out.println("foo 2");
    }
    public String toString() {
        return "foo";
    }
}
```

```java
public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

```java
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }
    public String toString() {
        return "baz";
    }
}
```

```java
public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

A polymorphism problem (cont’d)

- What would be the output of the following client code?

```java
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Finding output with diagrams

- One way to determine the output is to diagram each class and its methods, including their output:
  - Add the classes from top (superclass) to bottom (subclass).
  - Include any inherited methods and their output.
Finding output with tables

Another possible technique for solving these problems is to make a table of the classes and methods, writing the output in each square.

<table>
<thead>
<tr>
<th>method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td>foo 1</td>
<td>baz 1</td>
<td>baz 1</td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td>foo 2</td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td>foo</td>
<td>baz</td>
<td>baz</td>
</tr>
</tbody>
</table>

Polymorphism answer

```java
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Output:

baz
baz baz 1
foo 2
foo foo 1
bar 2
baz baz 1
mumble 2
foo foo 1
foo 2

Another problem

Assume that the following classes have been declared:
- The order of classes is changed, as well as the client.
- The methods now sometimes call other methods.

```java
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}
```

What would be the output of the following client code?

```java
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b   ");
    }
}
public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a   ");
        super.a();
    }
    public String toString() {
        return "Yam";
    }
}
```

```java
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println(); // to end the line of output
    food[i].b();
    System.out.println(); // to end the line of output
    System.out.println();
}
```
The class diagram

- The following diagram depicts the class hierarchy:

```
  Ham
   a
   b
   toString

  Lamb
   a
   b
   toString

  Yam
   a
   b
   toString

  Spam
   a
   b
   toString
```

- Notice that Ham's a method calls b. Lamb overrides b.
  - This means that calling a on a Lamb will also have a new result.

```java
public class Ham {
    public void a() {
        System.out.print("Ham a ");
        b();
    }
    public void b() {
        System.out.print("Ham b ");
    }
    public String toString() {
        return "Ham";
    }
}
```

```java
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b ");
    }
}
```

```
public class Ham {
    public void a() {
        System.out.print("Ham a ");
        b();
    }
    public void b() {
        System.out.print("Ham b ");
    }
    public String toString() {
        return "Ham";
    }
}
```

- Lamb's a output: Ham a Lamb b

The table

- Fill out the following table with each class's behavior:

<table>
<thead>
<tr>
<th>method</th>
<th>Ham</th>
<th>Lamb</th>
<th>Yam</th>
<th>Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toString</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
public class Ham {
    public void a() {
        System.out.print("Ham a ");
        b();
    }
    public void b() {
        System.out.print("Ham b ");
    }
    public String toString() {
        return "Ham";
    }
}
```

```
public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b ");
    }
}
```

```
public class Ham {
    public void a() {
        System.out.print("Ham a ");
        b();
    }
    public void b() {
        System.out.print("Ham b ");
    }
    public String toString() {
        return "Ham";
    }
}
```

The answer

- Fill out the following table with each class's behavior:

```
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

```
Output:
Yam
Yam a Ham a Spam b
Yam b
Spam b
Spam
Yam a Ham a Lamb b
Lamb b
Yam
Yam a Ham a Ham b
Ham b
Ham
Ham a Lamb b
Lamb b
```

Polymorphism at work!
Acknowledgments

- The course material used to prepare this presentation is mostly taken/adopted from the list below:
  - Java - How to Program, Paul Deitel and Harvey Deitel, Prentice Hall, 2012
  - Java - An Introduction to Problem Solving and Programming, Walter Savitch, Pearson, 2012
  - Mike Scott, CS314 Course notes, University of Texas Austin