Polymorphism

Lecture 11

Object-Oriented Programming

Agenda

- Classes and Interfaces
- The Object Class
- Object References
- Primitive Assignment
- Reference Assignment
- Relationship Between Objects and Object References
- References and Inheritance
- Single vs. Multiple Inheritance
- Polymorphism
- Key points about Polymorphism
- Polymorphism via Interfaces

Classes and Interfaces

Interface	Class
Models a *role*; defines a set of responsibilities	Models an *object* with properties and capabilities
Factors out common capabilities of *dissimilar* objects	Factors out common properties and capabilities of *similar* objects
Declares, but does not define methods	Declares methods and may define some or all of them
A class can implement *multiple* interfaces	A class can extend *only one* superclass

The `Object` Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The `Object` Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been overriding an existing definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class together along with some other information

The `Object` Class

- All objects are guaranteed to have a `toString` method via inheritance
- Thus the `println` method can call `toString` for any object that is passed to it

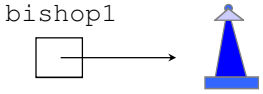
The `Object` Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- The `String` class defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version

Object References

- All interaction with an object occurs through object reference variables
- An object reference variable holds the reference (address, the location) of an object

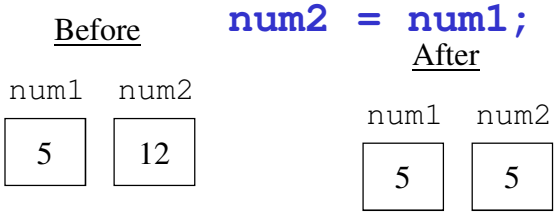
```
ChessPiece bishop1 = new ChessPiece();
```



Primitive Assignment

- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:

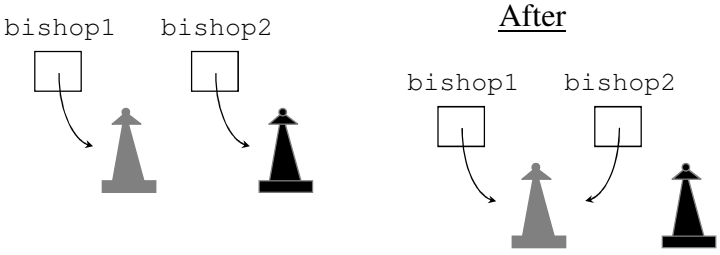
```
num2 = num1;
```



Reference Assignment

- For object references, the reference is copied:

```
bishop2 = bishop1;
```

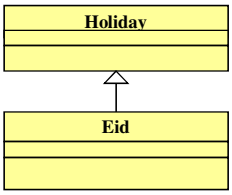


Relationship Between Objects and Object References

- Two or more references can refer to the same object; these references are called *aliases* of each other
- One object (and its data) can be accessed using different references

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class derived from it by inheritance
- For example, if the **`Holiday`** class is used to derive a child class called **`Eid`**, then a **`Holiday`** reference could actually be used to point to a **`Eid`** object




```
Holiday day;
day = new Holiday();
…
day = new Eid();
```

References and Inheritance

- Assigning an object to an ancestor reference is considered to be a widening conversion, and can be performed by simple assignment

```
Holiday day = new Eid();
```

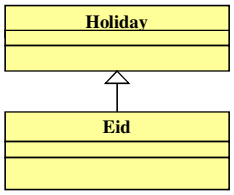
- Assigning an ancestor object to a reference can also be done, but it is considered to be a narrowing conversion and must be done with a cast

```
Eid c1 = new Eid();
Holiday day = c1;
Eid c2 = (Eid) day;
```

- The widening conversion is the most useful
  - for implementing polymorphism

References and Inheritance

- An object reference variable can refer to any object instantiated from
  - its own class, or
  - any class derived from it by inheritance
- For example,




```
Holiday day;
day = new Holiday();
…
day = new Eid();
```

The assignment of an object of a derived class to a reference variable of the base class can be considered as a widening conversion

References and Inheritance

- Through a given type of reference variable, we can invoke only the methods defined in that type

```
Holiday day;
day = new Eid();
```

```
class Holiday
{
   public void celebrate()
   {…}
}
class Eid extends Holiday
{
   public void celebrate()
   {…}
   public void goToPrayers()
   {…}
}
```

Can we do the following statements:

```
day.celebrate();
day.goToPrayers();
```

References and Inheritance

- We can "promote" an object back to its original type through an explicit narrowing cast:

```
Holiday day = new Eid();
day.celebrate();
…

Eid e = (Eid) day;
e.goToPrayers();
```

Question: which celebrate() will be invoked by the line:
day.celebrate();

What is Polymorphism?

- A polymorphic reference can refer to different types of objects at different times
  - In java every reference can be polymorphic except of references to base types and final classes.
- It is the type of the object being referenced, not the reference type, that determines which method is invoked
  - Polymorphic references are therefore resolved at run-time, not during compilation; this is called *dynamic binding*
- Careful use of polymorphic references can lead to elegant, robust software designs

Polymorphism

- Polymorphism: A polymorphic reference v is declared as class C, but unless C is final or base type, v can refer to an object of class C or to an object of *any class derived* from C.
- A method call v.<method_name>(<args>) invokes a method of the class of an object referred to by v (not necessarily C):

```
Ex1:
Holiday day =
       new Eid();
day.celebrate();
…
```

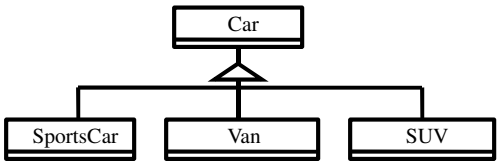
```
Ex2:
void process(Holiday day)
 { …
   day.celebrate();
   …  }
Eid day = ...;
process(day)
```

- A very common usage of polymorphism: If classes C1, C2, ..., Cn are all derived from C, define an array A of elements of C.
  The entries A[i] can then refer to objects of classes C1, ..., Cn.

Single vs. Multiple Inheritance

- Some object-oriented languages allow *multiple inheritance,* which allows a class to be derived from two or more classes, inheriting the members of all parents
- The price: collisions, such as the same variable name, same method name in two parents, have to be resolved
- Java decision: *single inheritance*, meaning that a derived class can have only one parent class

Polymorphism



- `SportsCar`, `Van`, and `SUV` are all *subclasses* of the `Car` class
- `Car` has an abstract `move()` method that each of its subclasses define
- however, each of its subclasses defines this method differently

Polymorphism

- What happens when we:
  - tell **`SportsCar`** to **`move()`**?
    (moves fast)
  - tell **`Van`** to **`move()`**?
    (moves at a moderate speed)
  - tell **`SUV`** to **`move()`**?
    (sometimes moves, sometimes just stops!
- Polymorphism is a fancy word for "multiple forms,"
  - e.g., multiple forms of response to the same message

Key points about Polymorphism

- A subclass inherits methods from its superclass; it can respond to all the same messages
- We can refer to an instance of a subclass as if it were an instance of its superclass – relaxes "strict type matching"

```
// Declaration:  Car is the "declared type" of myCar
Car myCar;

// Instantiation:  Van is the "actual type" of myCar
myCar = new Van();
```

Key points about Polymorphism

- Object will respond according to implementation defined in subclass!
- Assignment is NOT creating an instance of `Car`, which can only be done by `new`. It "is a" `Car`, but not an instance of `Car`!

```
myCar.move(); // myCar will move like a
Van, // though we refer to it as a Car!
```

- Can only call methods of the declared type!
- Thus can only call `Car` methods on `myCar` – can't take advantage of any of the methods of the actual type

Code Example

```
/**
 * A simple example of Polymorphism - doesn't show the
   full power...
 */
public class RaceTrack {

   private Car _car1, _car2, _car3;
   // note they're all declared as Cars
   public RaceTrack() {

      _car1 = new SportsCar();
      _car2 = new Van();
      _car3 = new SUV();
      // but actual types are subtypes of Car
   }
```

Code Example (Cont'd)

```
public void startRace() {
    // tell Car instances to move
    _car1.move();
    _car2.move();
    _car3.move();
  /** Note:
  startRace coded polymorphically in terms
  of declared   type Car, but methods will
  be called on instances of actual subtypes */
  }
} // end of class RaceTrack
```

Class Knowledge

- When sending a message to an instance, we do not need to know its *exact* class...
  - as long as it extends some superclass, we can send it any message we could send to the superclass
  - but the message may be handled differently by the subclass than it would be by the superclass
- Classic example: shapes (similar to simple **`RaceTrack`** example)
  - each shape subclass knows how to draw itself, but all do it differently
    - simply say **`_shape.draw()`**

Another Code Example

```
public class ShapeApp extends wheels.users.Frame {
    private Shape _shape1,_shape2;

    public ShapeApp() {
        super();
      _shape1 = new Triangle();
      _shape2 = new Square();
    }

    public void drawShapes() {
        _shape1.draw();
       _shape2.draw();
    }

    public static void main(String[] argv) {
        ShapeApp app = new ShapeApp();
    }
}
```

Polymorphism via Interfaces

- Define a polymorphism reference through interface
  - declare a reference variable of an interface type

    `Doable obj;`

  - the **`obj`** reference can be used to point to any object of any class that implements the `Doable` interface
  - the version of **`doThis`** depends on the type of object that **`obj`** is referring to:

    `obj.doThis();`

More Examples

```
Speaker guest;

guest = new Philosopher();
guest.speak();

guest = Dog();
guest.speak();
```

```
Speaker special;
special = new Philosopher();

special.pontificate();  // compiler error
```

```
Speaker special;
special = new Philosopher();

((Philosopher)special).pontificate();
```

```
public interface Speaker
{
   public void speak();
}

class Philosopher extends Human
    implements Speaker
{
   //
   public void speak()
    {…}
   public void pontificate()
   {…}
}

class Dog extends Animal
    implements Speaker
{
   //
   public void speak()
    {
   …
    }
}
```

Design Problem

- You are designing a program to sort a list of double values. However, you cannot use a predetermined sorting algorithm. Your choice of sorting algorithm must be made at runtime.
- What will you do to solve this problem?

Interface and Polymorphism Example

```
public interface SortInterface {
    public void sort(double[] list);
    }
```

```
public class QuickSort implements
    SortInterface {
    public void sort(double[] a) {
    // quick sort code here
    }
}
```

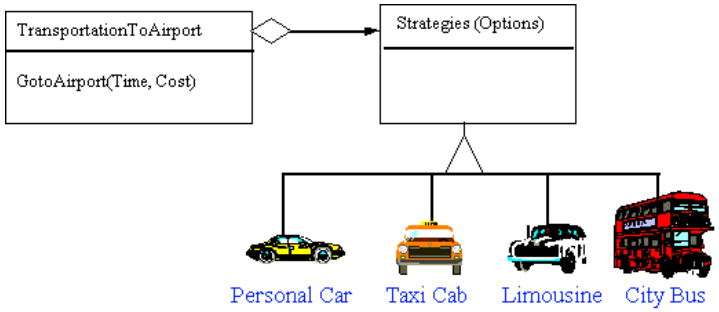
```
public class BubbleSort implements
    SortInterface {
    public void sort(double[] list)
    {
    // bubble sort code here
    }
}
```

```
public class SortingContext {
    private SortInterface sorter = null;

    public void sortDouble(double[] list) {
    sorter.sort(list);
    }

    public SortInterface getSorter() {
    return sorter;
    }

    public void setSorter(SortInterface
    sorter) {
    this.sorter = sorter;
    }
    }
```

```
public class SortingClient {
    public static void main(String[] args) {
    double[] list =
    {1,2.4,7.9,3.2,1.2,0.2,10.2,22.5,19.6};

    SortingContext context = new
    SortingContext();

    context.setSorter(new BubbleSort());

    context.sortDouble(list);
    for(int i =0; i< list.length; i++) {
    System.out.println(list[i]);
    }
}
```

Strategy Pattern

- The code we just saw is called a Strategy Pattern.
- It uses the polymorphic behavior through interfaces.



Readings

- **Book Name:** Head First JAVA
  Author Name: Kathy Sierra & Bert Bates
  Content: Chapter # 7 & 8