Overview

° Problem: Can we delay decisions regarding which method to use until run time?

° Polymorphism
  • Different methods with the same name can be executed for the same reference at different times

° Polymorphism via interfaces
  • References of interfaces can be assigned addresses of child objects

° Interfaces
  • The role of interfaces now becomes more apparent
Example class hierarchy
Polymorphism

- Normally we have this when we create an object:
  
  ```java
  Dog dog = new Dog();
  ```

- Polymorphism allows us to also do this:
  
  ```java
  Animal pet = new Dog();
  ```

  - The object reference variable can be a super class of the actual object type!
Where Polymorphism is Helpful

- **Arrays**
- **Passing parameters**
- **Returning values from a method**
Polymorphic Array Example

Animal[] myPets = new Animal[5];
myPets[0] = new Cat();
myPets[1] = new Cat();
myPets[3] = new Dog();

for (int i = 0; i < myPets.length; i++) {
    myPets.feed();
}

You can put any subclass of Animal in the Animal array!
Side Effects of Polymorphism

ArrayList pets = new ArrayList();
Dog dog = new Dog();
pets.add(dog);
int index = pets.indexOf(dog);
Dog dog1 = pets.get(index); // won't work
Object dog2 = pets.get(index);
dog2.bark(); // won't work
((Dog)dog2).bark(); // works because of casting

if (dog2 instanceof Dog) { // being careful
    ((Dog)dog2).bark();
}

Dog dog3 = (Dog) pets.get(index); // works because of casting
Recap: Interfaces

- **Interface** is collection of constants and abstract methods
  - different meaning than set of public methods that are documented, as in API
  - to implement interface must provide definitions for all its methods

- **Abstract methods have no implementation or body**
  - method header followed by semicolon
  - specifies how to communicate with method, not what it does
Interfaces

public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}

public class Dog extends Canine implements Pet {
    public void beFriendly() {
        wagTail();
    }

    public void play() {
        chaseBall();
    }

    . . . all the other Dog methods . . .
}

Explicitly typing in public and abstract is not necessary since they MUST be public and abstract

Must implement these methods since they are in Pet
Interfaces vs. Subclasses

° Make a subclass only when you want to make a more specific version of a class.

° Use an interface when you want to define a role that other classes can play
  • Doesn’t matter where those classes are in the inheritance tree.
Polymorphism via Interfaces

° An interface reference variable can be used to refer to any object of any class that implements that interface.

° This works the same with superclasses.

```java
Pet myPet = new Dog();
```

° The same side effects of polymorphism occur with interfaces as with inheritance.
Polymorphism via Interfaces

- An interface name can be used as the type of an ‘object reference variable’
- (Remember, an ‘interface’ is not a class…)
  Speaker current;
- Reference can be used to point to any object class Speaker or any class that implements the Speaker interface
- The version of speak that the following line invokes depends on the type of object that current is referencing
  - Which class that ‘implements’ Speaker…
    current.speak();
Suppose two classes, \texttt{Philosopher} and \texttt{Dog}.

Both implement the \texttt{Speaker} interface, providing distinct versions of the \texttt{speak} method.

In the following code, the first call to \texttt{speak} invokes one version and the second invokes another:

```java
Speaker guest; (guest is a reference to \texttt{Speaker})
guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```
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()
    {
    ...
    }
    ...
}
Bunny Example

```java
public interface Bunnies {
    public void moveBunny(int direction);
}

public class BigBunny implements Bunnies {
    public void moveBunny(int direction) {
        if (direction == 12) {
            y = y + 3;
            carrots = carrots - 2;
        }
    }
}

public class LittleBunny implements Bunnies {
    public void moveBunny(int direction) {
        if (direction == 12) {
            y = y + 1;
            carrots = carrots - 1;
        }
    }
}
```
### Summary of Polymorphism

**Object instantiated as:**

<table>
<thead>
<tr>
<th>Parent or Earlier Ancestor Class</th>
<th>Can/Cannot be assigned:</th>
</tr>
</thead>
<tbody>
<tr>
<td>To Child or Later Descendent</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Any Class</th>
<th>Can/Cannot be assigned:</th>
</tr>
</thead>
<tbody>
<tr>
<td>To any Interface it implements (and back to its original class)</td>
<td></td>
</tr>
<tr>
<td>To any “incompatible class”</td>
<td></td>
</tr>
</tbody>
</table>

- **Can/Cannot be assigned:**
  - To Parent or Earlier Ancestor (and back to its original class)
  - To any Interface it implements (and back to its original class)
  - To any “incompatible class”
Summary

- References of an interface can point to an object which implements the interface

- The specific method which is accessed depends on the type of the child

- This is very flexible since the method used depends on the flow of the program

- Often many different implementations of an interface may exist