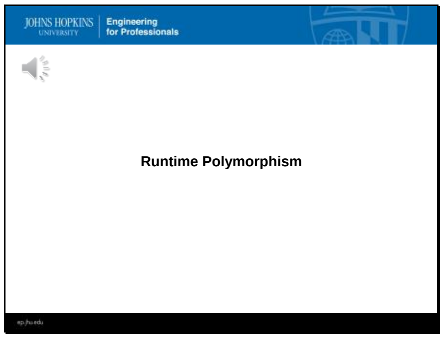
1



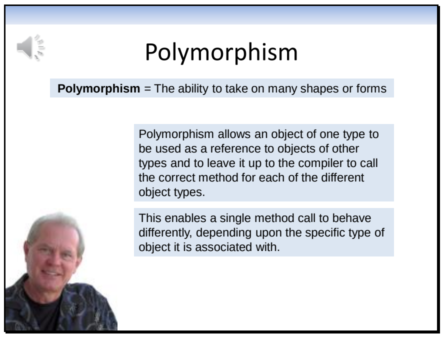
JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

Runtime Polymorphism

ep.jhu.edu

In this lecture you will learn about runtime polymorphism.

2



## Polymorphism

**Polymorphism** = The ability to take on many shapes or forms

Polymorphism allows an object of one type to be used as a reference to objects of other types and to leave it up to the compiler to call the correct method for each of the different object types.

This enables a single method call to behave differently, depending upon the specific type of object it is associated with.

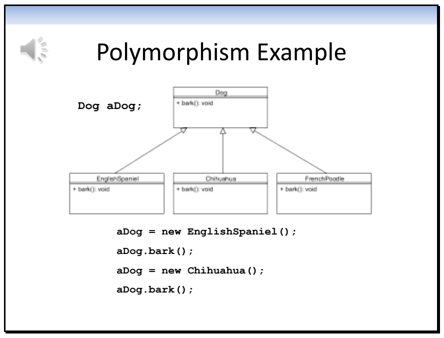
Let's start out with the definition of polymorphism. The dictionary definition of polymorphism is "The ability to take on many shapes or forms".

From a programming standpoint polymorphism is a very powerful mechanism. It allows an object of one type to be used as a reference to different types, and then leaves it up to the compiler to call the correct method for each different type of object.

Essentially, this enables a method call to behave differently, depending upon the specific type of object it's associated with.

Now, this may sound complicated, but it's pretty straightforward. So…let's take a look at an example.

3



## Polymorphism Example

```
Dog aDog;
```

```
aDog = new EnglishSpaniel();
aDog.bark();
aDog = new Chihuahua();
aDog.bark();
```

This inheritance structure shows a Dog class that has several subclasses.

Each subclass represents a different type of dog. Each of these types differs from one another in several different ways.

One characteristic that is certainly different between them is the sound of their bark.

This diagram shows that each subclass has overridden an inherited bark() method. If we were to represent dogs in a computer program, each of the bark methods would contain different code…maybe just a different sound clip…but different nevertheless.

The mechanism of polymorphism, or runtime polymorphism to be more precise, works like this.

We can declare a reference to the base class…which is the Dog class.  Let's call this reference aDog.

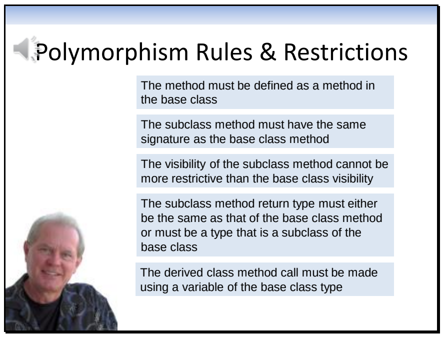
In Java, a base class object can reference any base class object or derived class object. This means that we let the variable aDog reference an EnglishSpaniel object, a Chihuahua object, or a FrenchPoodle object.

We could reference an EnglishSpaniel like this .

And, if we invoke the bark() method, like this , Java will execute the bark() method for an EnglishSpaniel.

We could also reference a Chihuahua, like this , and execute the Chihuahua's bark() method …and so on.

4



## Polymorphism Rules & Restrictions

- The method must be defined as a method in the base class
- The subclass method must have the same signature as the base class method
- The visibility of the subclass method cannot be more restrictive than the base class visibility
- The subclass method return type must either be the same as that of the base class method or must be a type that is a subclass of the base class
- The derived class method call must be made using a variable of the base class type

There are some rules and restrictions, however, when it comes to polymorphism.

First, for a method to be invoked polymorphically, it must be defined as a method in the base class.

Second, the subclass method must have the same signature of the method defined in the base class.

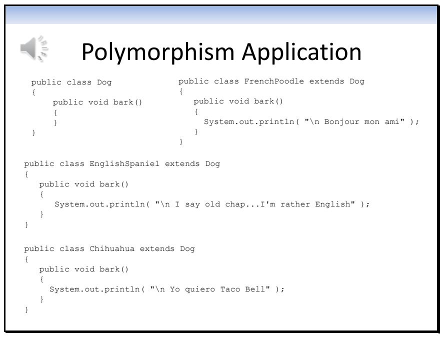
Third, the subclass method visibility cannot be more restrictive than the base class method visibility.

Fourth, the return type of the subclass method must either be the same as that of the base class method or must be a type that is a subclass of the base class.

And fifth, the derived class method call must be made using a variable of the base class type

Let’s look at an example.

5



## Polymorphism Application

```
public class Dog
{
    public void bark()
    {
    }
}
```

```
public class FrenchPoodle extends Dog
{
   public void bark()
   {
     System.out.println( "\n Bonjour mon ami" );
   }
}
```

```
public class EnglishSpaniel extends Dog
{
   public void bark()
   {
      System.out.println( "\n I say old chap...I'm rather English" );
   }
}

public class Chihuahua extends Dog
{
   public void bark()
   {
     System.out.println( "\n Yo quiero Taco Bell" );
   }
}
```

Let’s revisit the Dog example that was introduced earlier to see an actual application of polymorphism in action.

Here’s some code for the Dog class. We’re only going to be focusing on methods and not on descriptive attributes in this example.

Note that the bark() method in the Dog class doesn’t do anything. When a method doesn’t do anything it’s sometimes called a no-op method or a null method.

There’s a better way to accomplish this without using a no-op method…and we’ll see this in another lecture. For now, this will get the job done.

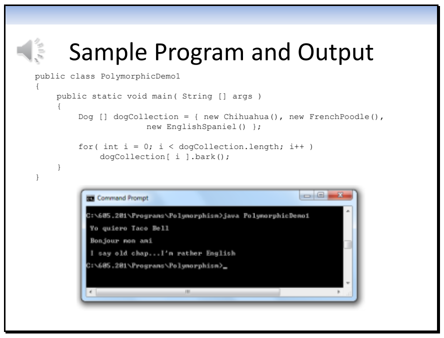
Here’s some code for the EnglishSpaniel bark() method. To keep things simple we wont’ use a sound clip.

And here’s some code for the Chihuahua and

FrenchPoodle bark() methods as well.

Now, let’s write a program that uses polymorphic behavior.

6



Sample Program and Output

```
public class PolymorphicDemo1
{
    public static void main( String [] args )
    {
        Dog [] dogCollection = { new Chihuahua(), new FrenchPoodle(),
                     new EnglishSpaniel() };

        for( int i = 0; i < dogCollection.length; i++ )
            dogCollection[ i ].bark();
    }
}
```

```
C:\605.201\Programs\Polymorphism>java PolymorphicDemo1
 Yo quiero Taco Bell
 Bonjour mon ami
 I say old chap...I'm rather English
C:\605.201\Programs\Polymorphism>_
```

Here’s our program.

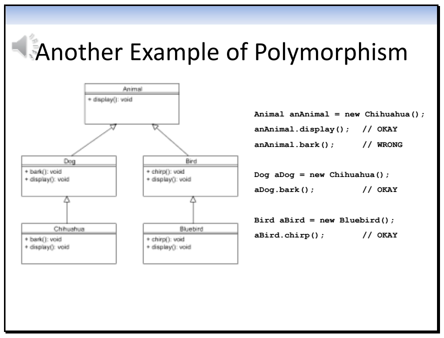
The program declares an array of type Dog, and creates a Chihuahua member, a French Poodle member, and an EnglishSpaniel member.

The array is really an array of references to Dog objects.

Then, the program simply loops through the array and invokes the bark() method for each Dog in the array.

And,  here’s the program output.

7



Another Example of Polymorphism

```
Animal anAnimal = new Chihuahua();
anAnimal.display();  // OKAY
anAnimal.bark();     // WRONG

Dog aDog = new Chihuahua();
aDog.bark();         // OKAY

Bird aBird = new Bluebird();
aBird.chirp();       // OKAY
```

Here’s one more example of polymorphism.

In this example, Animal is the base class and defines a display() method.

Polymorphic behavior can be obtained for the display() method by using a reference to Animal with any of the classes derived from Animal.

 For example, a Chihuahua’s display() method can be invoked polymorphically like this.

An Animal reference can’t be used to invoke a Chihuahua’s bark() method…like this , because the bark() method is not defined in the Animal class.

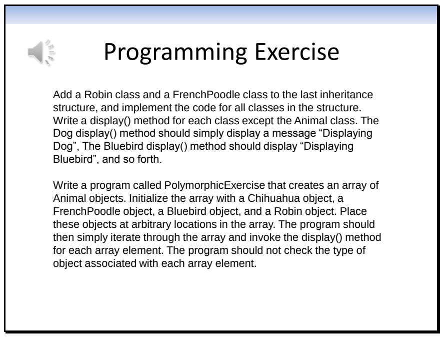
Similarly, an Animal reference can't be used to invoke a Bird's chirp() method for exactly the same reason.

A Dog reference  could be used to invoke polymorphic behavior for the bark() method.

And, a Bird reference  could be used to invoke polymorphic behavior for the chirp() method.

If this is not clear…then please revisit the polymorphism rules and restrictions at the beginning of the lecture.

8



## Programming Exercise

Add a Robin class and a FrenchPoodle class to the last inheritance structure, and implement the code for all classes in the structure. Write a display() method for each class except the Animal class. The Dog display() method should simply display a message "Displaying Dog", The Bluebird display() method should display "Displaying Bluebird", and so forth.

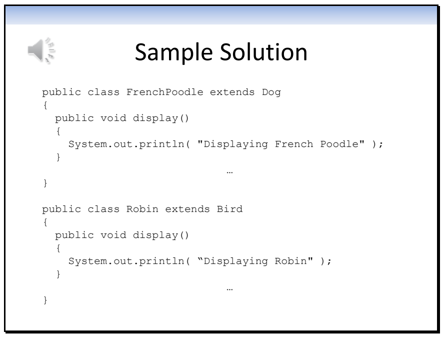
Write a program called PolymorphicExercise that creates an array of Animal objects. Initialize the array with a Chihuahua object, a FrenchPoodle object, a Bluebird object, and a Robin object. Place these objects at arbitrary locations in the array. The program should then simply iterate through the array and invoke the display() method for each array element. The program should not check the type of object associated with each array element.

At this point I'd like you to do a programming exercise.

Please pause this lecture now and read the exercise instructions.

You may resume the lecture once you have your program working.

9



## Sample Solution

```
public class FrenchPoodle extends Dog
{
  public void display()
  {
    System.out.println( "Displaying French Poodle" );
  }
                         …
}

public class Robin extends Bird
{
  public void display()
  {
    System.out.println( "Displaying Robin" );
  }
                         …
}
```

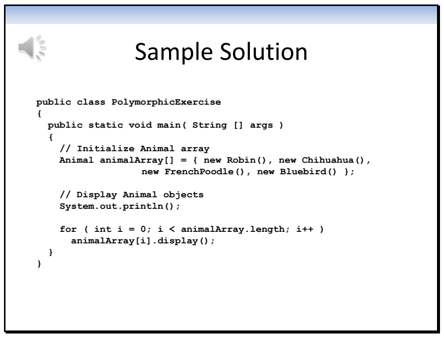
Here's a partial solution to the exercise.

The Dog and Bird classes are implemented by extending the Animal class and providing the appropriate implementation for the display() method.

The French Poodle class extends the Dog class and overrides the inherited display() method. The Chihuahua class is implemented in a similar manner.

The Robin class extends the Bird class and overrides its inherited display() method as well as illustrated here. The Bluebird class is implemented in a similar manner.

10



## Sample Solution

```
public class PolymorphicExercise
{
  public static void main( String [] args )
  {
    // Initialize Animal array
    Animal animalArray[] = { new Robin(), new Chihuahua(),
                   new FrenchPoodle(), new Bluebird() };

    // Display Animal objects
    System.out.println();

    for ( int i = 0; i < animalArray.length; i++ )
      animalArray[i].display();
  }
}
```

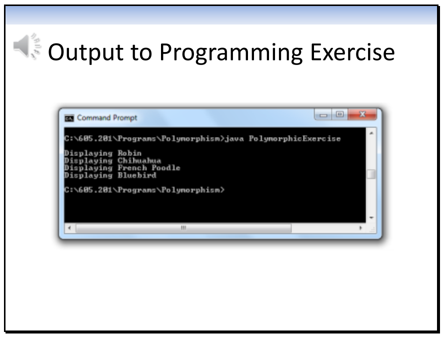
Here's a program that tests things out.

The program simply initializes an array of Animal objects with a number of different Animal subclasses, then loops through the array and invokes the display() method for each element in the array.

Polymorphic behavior causes the correct display() method to be invoked for each subclass object.

Let's take a look at the output.

11



## Output to Programming Exercise

Command Prompt

```
C:\605.201\Programs\Polymorphism>java PolymorphicExercise

Displaying Robin
Displaying Chihuahua
Displaying French Poodle
Displaying Bluebird

C:\605.201\Programs\Polymorphism>
```

And…here’s the program output.