`Polymorphism`

Allowing methods and objects to take on different forms

Alwin Tareen

Polymorphism: Method Overloading

Method overloading

- This is where more than one method in the same class has the same name, but different parameter lists. For example:

```
public double calcArea(doulbe length, double width)
public double calcArea(double radius)
public double calcArea(double base, double height)
```

Polymorphism: Superclass References

Consider the following class, Pet:

```
public class Pet
{
    private String name;

    public Pet(String n)
    {
        name = n;
    }
}
```

Polymorphism: Superclass References

Consider Dog and Cat, which are subclasses of Pet:

```
public class Dog extends Pet
{
    public Dog(String n)
    {
        super(n);
    }
}
```

```
public class Cat extends Pet
{
    public Cat(String n)
    {
        super(n);
    }
}
```

Polymorphism: Superclass References

Now, consider the following client class that uses Dog, Cat, and Pet:

```
public class PetTest
{
    public static void main(String[] args)
    {
        Pet animal;
        animal = new Dog("Fido");
        animal = new Cat("Fluffy");
    }
}
```

Polymorphism: Superclass References

- The previously indicated code is an example of polymorphism. First, I declared a reference of type `Pet`, called `animal`. Then, I can assign both a `Dog` object and a `Cat` object to the reference `animal`.
- In general, `animal` can reference any object of a class that is a subclass of `Pet`.

Downcasting

Consider the following Pet class, with an instance variable, a constructor, and a method:

```
public class Pet
{
    private String name;

    public Pet(String n)
    {
        name = n;
    }

    public String getName()
    {
        return name;
    }
}
```

Downcasting

Now consider the following Dog class, which is a subclass of Pet:

```
public class Dog extends Pet
{
    public Dog(String n)
    {
        super(n);
    }

    public String bark()
    {
        return "ruff, ruff";
    }
}
```

Downcasting

Consider the following client class, where the `bark()` method is being run on a Dog object.

```
public class DogTest
{
    public static void main(String[] args)
    {
        Pet animal = new Dog("Fido");
        System.out.println(animal.getName());

        String result = animal.bark(); // ERROR!
        System.out.println(result);
    }
}
```

Downcasting

`animal` can only access `Pet` methods

- Even though the reference called `animal` has a `Dog` object created within it, it is a `Pet` data type, and therefore can only access `Pet` methods.

The `bark()` method requires a downcast

- Therefore, the call to `getName()` works fine, but the call to the method `bark()` requires that the `animal` reference be **cast** to a `Dog` data type, before the call is made. This is called a **downcast**.
- Subclass methods can only be “seen” by superclass references with a downcast.

Downcasting

The following is the corrected version of DogTest. Note that we need an extra set of brackets to force the downcast to be evaluated first, because the dot operator has a higher level of precedence.

```
public class DogTest
{
    public static void main(String[] args)
    {
        Pet animal = new Dog("Fido");
        System.out.println(animal.getName());

        String result = ( (Dog) animal).bark(); // downcast
        System.out.println(result);
    }
}
```

`Polymorphism`: End of Notes