Polymorphism

Downcasting and upcasting

IT TALENTS
Training Camp

Polymorphism

- Polymorphism is one of the four concepts in OOP
- Polymorphism is the characteristic of being able to assign a different meaning or usage to something in different contexts
- In other word, a variable with a given name may be allowed to have different forms and the program can determine which form of the variable to use at the time of execution
- In java polymorphism is achieved by overriding methods in the subclass
- Polymorphism is a generic term that means 'many shapes'

IT TALENTS
Training Camp

Demonstrating polymorphism by example

Let's override method walk() in classes

Dog, Cat and Bird

This way, each animal will walk in different way, typical for the respective animal

Demonstrating polymorphism by example

```
public class Cat extends Animal{
    ...

    @Override
    public void walk() {
        System.out.println("Walking like a cat");
    }
}
```

```
public class Dog extends Animal{
    ...

    @Override
    public void walk() {
        System.out.println("Walking like a dog");
    }
}
```

Demonstrating polymorphism by example

What happens when in the main method in class ZooDemo try to invoke method walk for the variable cat?

```
public class ZooDemo {
    public static void main(String[] args) {
        ...

        Animal cat = new Cat();

        ...
        cat.walk();
    }
}
```

Demonstrating polymorphism by example

The ouput in the console is:

```
Walking like a cat
```

The method walk() from class Cat has been invoked instead of walk() from class Animal

The programmer (and the program) does not always have to know the exact type of the object in advance, and so the exact behavior is determined at run-time (this is called late binding or dynamic binding)

IT TALENTS
Training Camp

Demonstrating polymorphism by example

- You always deal with reference, but the method which will be called depends on the type of the instance, not the type of the reference
- In some other OOP languages there is a term *virtual method* (or function)
- In java all methods are virtual, so everytime you invoke a method, the decision which method to be called is taken at runtime and it depends on the instance

Demonstrating polymorphism by example

Let's complete our example and call methods walk and makeSomeNoise for all the animals in the cage.

Demonstrating polymorphism by example

```java
public static void main(String[] args) {
        Zoo zoo = new Zoo(10);
        Animal cat = new Cat();
        Dog dog = new Dog();
        Bird bird = new Bird();

        zoo.addAnimal(cat);
        zoo.addAnimal(dog);
        zoo.addAnimal(bird);

        Animal[] animalsInTheZoo = zoo.getAnimals();

        for (int i = 0; i < animalsInTheZoo.length; i++) {
            if(animalsInTheZoo[i] != null) {
                animalsInTheZoo[i].walk();
                animalsInTheZoo[i].makeSomeNoise();
            }
        }
    }
```

More about references and instances

Lets add method sing() in the class Bird

What will happen when try to do this?

```java
public class Bird extends Animal {
    ...

    public void sing() {
        System.out.println("Singing...");
    }
}
```

```java
Animal bird = new Bird();
bird.sing();
```

IT TALENTS
Training Camp

More about references and instances

This will result in a compilation error, because there is not such method declared in class Animal, although the instance is of type Bird.

Remember:

- Which methods can be called depends on the reference type
- But which body will be executed depends on the instance type

What's the solution?
Downcasting

Downcasting (or just casting) is used to explicitly say to the compiler that reference of a base class refers to an instance of its subclass

```
public static void main(String[] args) {
    Animal bird = new Bird();
    ((Bird)bird).sing();
}
```

Downcasting

(or just casting)

Downcasting

Downcasting is unsave operation and is good idea to check if the reference refer to an instance of the right class using *instanceof* operator

```
for (int i = 0; i < animalsInTheZoo.length; i++) {
    if(animalsInTheZoo[i] != null) {
        animalsInTheZoo[i].walk();
        animalsInTheZoo[i].makeSomeNoise();

        if(animalsInTheZoo[i] instanceof Bird) {
            Bird birdInZoo = (Bird) animalsInTheZoo[i];
            birdInZoo.sing();
        }
    }
}
```

Using instanceof operator before casting

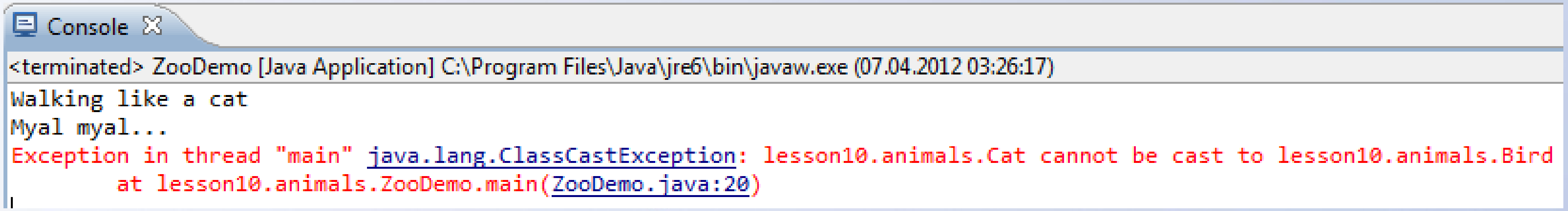
IT TALENTS
Training Camp

Downcasting

If we cast to wrong class we'll get exception of type ClassCastException

```
zoo.addAnimal(cat);
zoo.addAnimal(dog);
zoo.addAnimal(bird);
Animal[] animalsInTheZoo = zoo.getAnimals();

for (int i = 0; i < animalsInTheZoo.length; i++) {
    if(animalsInTheZoo[i] != null) {
        animalsInTheZoo[i].walk();
        animalsInTheZoo[i].makeSomeNoise();
        Bird birdInZoo = (Bird) animalsInTheZoo[i];
        birdInZoo.sing();
    }
}
```



Console

<terminated> ZooDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (07.04.2012 03:26:17)

```
Walking like a cat
Myal myal...
Exception in thread "main" java.lang.ClassCastException: lesson10.animals.Cat cannot be cast to lesson10.animals.Bird
	at lesson10.animals.ZooDemo.main(ZooDemo.java:20)
```

Upcasting

- Java permits an object of a subclass type to be treated as an object of any superclass type. This is called upcasting.
- Upcasting is save operation
- Upcasting is very rarely used

Upcasting

In most situations, the upcast is entirely unnecessary and has no effect. However, there are situations where the presence of the upcast changes the meaning of the statement(or expression):

Suppose that we have overloaded methods:

```
public void doIt(Object o)...
public void doIt(String s)...
```

If we have a String and we want to call the first overload rather than the second, we have to do this:

```
String arg = ...

doIt((Object) arg);
```

Final methods and classes

- *final* keyword can be applied to methods and classes
- Final class means that it cannot be extended by other class
- Final method means that this method cannot be overrided in the subclasses. This way its body is never changed

```
public final void walk() {
    System.out.println("Walking like a dog");
}
```

Summary

- What is polymorphism and how to achieve it?
- Upcasting and downcasting
- What the meaning of *final* for methods and classes

IT TALENTS
Training Camp