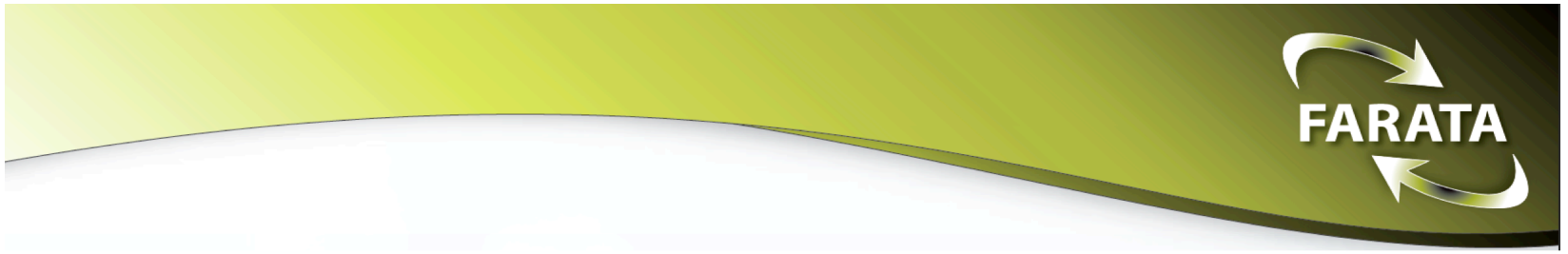

Java Programming
Unit 4

Abstract Classes, Interfaces, Polymorphism

By Yakov Fain (a.k.a. Budam)

Casting

All Java classes form an inheritance tree with the class Object. While declaring non-primitive variables you are allowed to use either the exact data type of this variable or one of its ancestor data types. For example, if the class NJTax extends Tax each of these lines is correct.

```
NJTax myTax1   = new NJTax();
Tax myTax2     = new NJTax();  // upcasting
Object myTax3  = new NJTax();  // upcasting
```

If Employee and Contractor extend class Person, you can declare array of type Person, but populate it with employees and contractors:

```
Person workers[] = new Person [100];

workers[0] = new Employee("Yakov", "Fain");
workers[1] = new Employee("Mary", "Lou");
workers[2] = new Contractor("Bill", "Shaw");
```

Casting (cont.)

While processing a collection of different objects you may use the instanceof operator to check the actual data type of an object. Placing a data type in parenthesis in front of another object means that you want to *cast* this object to specified type.

```
Person workers[] = new Person [20];

// Populate the array workers here….
for (int i=0; i<20; i++){
    Employee currentEmployee;
    Contractor currentContractor;

    if (workers[i] instanceof Employee){                    // type check

        currentEmployee = (Employee) workers[i];            // downcasting
        // do some employee-specific processing here

    } else if  (workers[i] instanceof Contractor){

        currentContractor = (Contractor) workers[i];        // downcasting
        // do some contractor-specific  processing here
    }
}
```

Abstract Classes

A class is called abstract if it has at least one method that's not implemented.
It must have the keyword abstract in the declaration line. You can not instantiate an abstract class.

```
abstract public class Person {

      public void changeAddress(String address){
                System.out.println("New address is" + address);
       }
 ...
  // an abstract method to be implemented in subclasses
   public abstract boolean increasePay(int percent);
}
```

The increasePay() method will be implemented in the subclasses of Person, which may implement it differently, but the name and the number of arguments of increasePay() will be the same. Guaranteed.

Promoting Workers. Take 1.

A company has employees and contractors. Design the classes without using interfaces to represent the people who work for this company.

The classes should have the following methods:

changeAddress()
promote()
giveDayOff()
increasePay()

Promotion means giving one day off and raising the amount in the pay check.

For employees, the method increasePay() should raise the yearly salary and, for contractors, it should increase their hourly rate.

```
abstract public class Person {

        private String name;
        int INCREASE_CAP = 20;  // cap on pay increase

         public Person(String name){
                this.name=name;
         }

         public String getName(){
                return "Person's name is " + name;
         }

         public void changeAddress(String address){
                  System.out.println("New address is" + address);
         }

         private void giveDayOff(){
                  System.out.println("Giving a day off to " + name);
         }

      public void promote(int percent){
                  System.out.println(" Promoting a worker...");
                  giveDayOff();

                  //calling an abstract method
                  increasePay(percent);
          }
          // an abstract method to be implemented in subclasses
          public abstract boolean increasePay(int percent);
}
```

Listing 7.1 shows an abstract ancestor

```
public class Employee extends Person{

        public Employee(String name){
                super(name);
        }
        public boolean increasePay(int percent) {
                System.out.println("Increasing salary by " +
               percent + "%. "+ getName());
                return true;
        }
}
```

Descendants implement increasePay() differently

```
public class Contractor extends Person {

        public Contractor(String name){
                super(name);
        }
        public boolean increasePay(int percent) {
                if(percent < INCREASE_CAP){
                  System.out.println("Increasing hourly rate by " +
                        percent + "%. "+ getName());
                  return true;
                } else {
                  System.out.println("Sorry, can't increase hourly rate by more
               than " + INCREASE_CAP + "%. "+ getName());
                  return false;
                }
        }
}
```

```
public class TestPayIncrease2 {

        public static void main(String[] args) {

      Person workers[] = new Person[3];

                workers[0] = new Employee("John");
                workers[1] = new Contractor("Mary");
                workers[2] = new Employee("Steve");

                for (Person p: workers){
                        p.promote(30);
                }
        }
}
```

The array workers has a mix of employees and contractors, but
the class TestPayIncrease2 includes the code that promotes people in a generic way.

The proper method will be invoked based on the actual type of the worker.

This is an illustration of a polymorphic behavior

Walkthrough

- Import the sample code for Lesson 7
- Run TestPayIncrease2
- Review the output shown below.
  I know, it's hard to understand. Ask questions.

```
Promoting a worker...
Giving a day off to John
Increasing salary by 30%. Person's name is John
 Promoting a worker...
Giving a day off to Mary
Sorry, can't increase hourly rate by more than 20%. Person's name is Mary
 Promoting a worker...
Giving a day off to Steve
Increasing salary by 30%. Person's name is Steve
```

Interfaces

- Interfaces are special entities that can contain only declarations of methods and final variables.

```
public interface Payable {
    boolean increasePay(int percent);
}
```

- A class can implement one or more interfaces

```
class Employee implements Payable, Promotionable {…}

class Contractor implements Payable{…}
```

- If a class declaration has the `implements` keyword it MUST implement every functions that's declared in every interface.

Polymorphic solution with interfaces. Take 1

```
public class TestPayIncreasePoly {

      public static void main(String[] args) {

            Person workers[] = new Person[3];

             workers[0] = new Employee("John");
             workers[1] = new Contractor("Mary");
             workers[2] = new Employee("Steve");

             for (Person p: workers){

               ((Payable)p).increasePay(30);

             }
      }
}
```

Assumption:
both Employee and Contractor extend Person and implement Payable (see Listing 6-2 and 6-3)

Polymorphic solution with interfaces. Take 2

```
public class TestPayIncreasePoly {

   public static void main(String[] args) {

       Payable workers[] = new Payable[3];

              workers[0] = new Employee("John");
              workers[1] = new Contractor("Mary");
              workers[2] = new Employee("Steve");

              for (Payable p: workers){

                 p.increasePay(30);

              }
   }
}
```

Assumption:
both Employee and Contractor
implement Payable (see Listing 6-2 and 6-3)

Homework

1. Study the materials from Lesson 7 from the textbook and do the assignment from the Try It sections of these lessons.

2. Do additional reading about the Java interfaces: http://download.oracle.com/javase/tutorial/java/concepts/interface.html

3. Invent and program any sample application that can be implemented with interfaces illustrating polymorphism