Java Training

Day 9: Generics & Interfaces

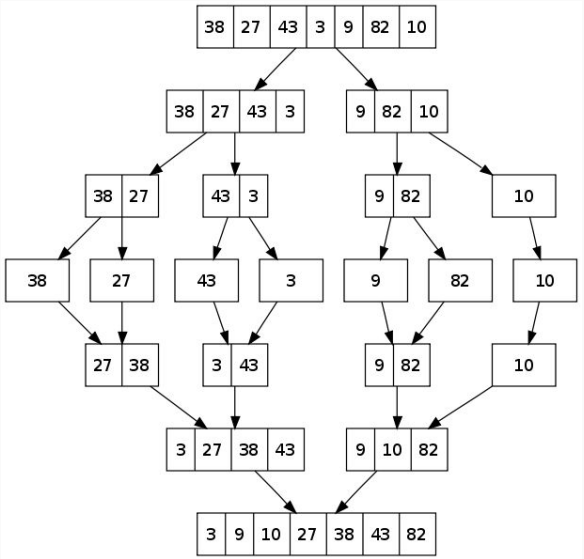
Download today’s slides:
go/java+espresso-training/day9

The idea behind Generic Types

Many algorithms are logically the same irrespective of the type of data they are being applied to. By using **generics** you can define an algorithm once and apply it to any data type.

The classic example is **sorting**. As long as you can define an **ordering** relationship between elements of a given type (numbers, letters, objects), you can apply any sorting algorithm.

In theory, you should then be able to use a single sorting algorithm implementation to sort any set of orderable elements



Idea of Generic Types (part 2)

With the fixed types of Java, a generally applicable sorting algorithm is impossible to implement

```
void sort(Integers[] nums){...}
```
- works for `Integers`

But if you need to sort an array of `Pumpkin` objects by weight, you'd need

```
void sort(Pumpkins[] pumps){...}
```

What is needed is a way to create a `sort()` method that takes an array of type `T` (where `T` can be any object), which, internally, would use whatever `compare()` method is appropriate for type `T`

With generics you can do that with:

```
<T> void sort(T[] arr){...}
```

Generics with `Bob`

```
public class Generics1 {
        public static void main(String[] args) {
                Bob bob1 = new Bob();
                bob1.age = 34.4;
                System.out.println("bob1 is "+ bob1.age);
        }
}

class Bob<T> {
        public T age;
}
```

```
> java Generics1
bob1 is 34.4
```

`javac` correctly infers that `T` is `Double`

More Generics with `Bob`

```
public class Generics1 {
        public static void main(String[] args) {
                Bob bob1 = new Bob();
                bob1.age = 34.4;
                System.out.println("bob1 is "+ bob1.age);

                Bob bob2 = new Bob();
                bob2.age = "really old";
                System.out.println("bob2 is "+ bob2.age);
        }
}

class Bob<T> {
        public T age;
}
```

```
> java Generics1
bob1 is 34.4
bob2 is really old
```


In `bob1, age` is inferred to be `Double`, but for `bob2 age` is a `String`

Yet more, Generics with `Bob`

```
public class Generics1 {
        public static void main(String[] args) {
                Bob bob1 = new Bob();
                bob1.age = 34.4;
                System.out.println("bob1 is "+ bob1.age);
        }
}

class Bob<T> {
        public T age;
}
```

If you tried compiling, you may have noticed:

```
Note: Generics1.java uses
unchecked or unsafe
operations.
Note: Recompile with
-Xlint:unchecked for details.
```

```
> javac -Xlint Generics1.java

Generics1.java:3: warning: [rawtypes] found raw type: Bob
         Bob bob1 = new Bob();
         ^
  missing type arguments for generic class Bob<T>
  where T is a type-variable:
    T extends Object declared in class Bob
...
```

Return of, Generics with `Bob`

```
public class Generics1 {
        public static void main(String[] args) {
                Bob<Double> bob1 = new Bob<Double>();
                bob1.age = 34.4;
                System.out.println("bob1 is "+ bob1.age);
        }
}

class Bob<T> {
        public T age;
}
```

`javac` likes to have the type for generics declared when the instance is created

Return of the Son of, Generics with `Bob`

```
public class Generics1 {
        public static void main(String[] args) {
                Bob<Double,Integer> bob1 = new Bob<Double,Integer>();
                bob1.age = 34.4;
                bob1.height = 22;
                System.out.println("bob1 is "+ bob1.age);
                System.out.println("and height: "+bob1.height);
        }
}

class Bob<T,S> {
        public T age;
        public S height;
}


> java Generics1
bob1 is 34.4
and height: 22
```

You can use any capital for a generic, but the convention is

- E - Element
- K - Key
- T - Type
- V - Value
- S,U,V - 2nd, 3rd, 4th types

Code exercise #1

Modify `bob1` in `main()` so that `age` is a `String` and `height` is a `Double`

```
public class Generics1 {
        public static void main(String[] args) {
                Bob<Double,Integer> bob1 = new Bob<Double,Integer>();
                bob1.age = 34.4;
                bob1.height = 22;
                System.out.println("bob1 is "+ bob1.age);
                System.out.println("and height: "+bob1.height);
        }
}

class Bob<T,U> {
        public T age;
        public U height;
}
```

Solution

```
public class Generics1 {
        public static void main(String[] args) {
                Bob<String,Double> bob1 = new Bob<String,Double>();
                bob1.age = "damn old";
                bob1.height = 55.556;
                System.out.println("bob1 is "+ bob1.age);
                System.out.println("and height: "+bob1.height);
        }
}

class Bob<T,U> {
        public T age;
        public U height;
}
```

```
$ java Generics1
bob1 is damn old
and height: 55.556
```

Code exercise #2

Modify the `bob` class so that it has a third generic member variable `weight` and use it in `bob1`

```
public class Generics1 {
        public static void main(String[] args) {
                Bob<Double,Integer> bob1 = new Bob<Double,Integer>();
                bob1.age = 34.4;
                bob1.height = 22;
                System.out.println("bob1 is "+ bob1.age);
                System.out.println("and height: "+bob1.height);
        }
}

class Bob<T,U> {
        public T age;
        public U height;
}
```

Solution

```
public class Generics1 {
        public static void main(String[] args) {
                Bob<String,Double,Integer> bob1 = new Bob<String,Double,Integer>();
                bob1.age = "damn old";
                bob1.height = 55.556;
                bob1.weight = 250;
                System.out.println("bob1 is "+ bob1.age);
                System.out.println("and height: "+bob1.height);
                System.out.println("and weight: "+bob1.weight);
        }
}

class Bob<T,U,V> {
        public T age;
        public U height;
        public V weight;
}
```

Java Interfaces

A **Java interface** is a bit like a class, except a **Java interface** can only contain method signatures and fields. An **Java interface** cannot contain an implementation of the methods, only the signature (name, parameters and exceptions) of the method. You can use **interfaces** in **Java** as a way to achieve polymorphism.

```
interface Animal {
   public void eat();
   public void makeNoise();
   public void move();
}
```

Interface Example, and exercise #3
create a new creature class that implements `Animal`

```
public class Zoo {
        public static void main(String[] args){
                Snake sam = new Snake();
                sam.makeNoise();
        }
}
interface Animal {
        public void eat();
        public void makeNoise();
        public void move();
}

class Snake implements Animal {
        public void eat(){ System.out.println("eats it whole"); }
        public void makeNoise(){ System.out.println("HisSSSSssssss"); }
        public void move(){ System.out.println("slithering"); }
}
```

A class can implement multiple **Interfaces**

```
public class Zoo {
        public static void main(String[] args){
                Snake sam = new Snake();
                sam.makeNoise();
                sam.looksLike();
        }
}
interface Appearance {
        public void looksLike();
}
interface Animal {
        public void eat();
        public void makeNoise();
        public void move();
}
class Snake implements Animal, Appearance {
        public void eat(){ System.out.println("eats it whole"); }
        public void makeNoise(){ System.out.println("HisSSSSssssss"); }
        public void move(){ System.out.println("slithering"); }
        public void looksLike(){ System.out.println("Long thin thing"); }
}
```