CMSC131

Introduction to “Generics” Concept

Example Data Structure: Pair

In Java, a method can return more than one value.

50% A. True

50% B. False

Response Counter

A Simple Problem

When we say that a method can only return one value, we are being very literal. It can only return a single primitive or reference.

What if I asked you to write a program that had various methods that ***needed*** to return a pair of values, in this case we'll say of the same data type as each other?

General Purpose Data Structures

Imagine building a `Pair` class with the ability to store references to two objects of the same data type.

The constructor could be sent references to the two objects and store them. The getters could return those references. A single object could "hold" two things.

- References use the same amount of space in memory, regardless of the size of the object to which they refer.
- Java makes use of everything we declare having a specific type at compile time for checking for valid commands.

Polymorphism and Generics

One of the powerful paradigms used in **object oriented programming** is being able to have a common structure that can be used with a wide variety of data types.

This is one form of what is known as polymorphism, specifically *parametric polymorphism*.

- In Java, this can be supported by something called **Generics** to effectively build a template than can be instantiated in many ways.

`Pair<Type>`

`Pair` is an example class that I've created for demonstration purposes, though it could be useful. It can hold two references to any type of object that you specify.

- It is a **Generic** structure, and you can specify what `<Type>` of object references a specific instance of `Pair` will be allowed to hold.

Why not totally generic?

Having a template where the programmer can state the data type being used in a particular `Pair` can be better than simply allowing two references to *any* object type to be stored.

- Specifically, this approach will allow for compile-time type checking and make it easier to invoke methods using the objects that were stored in the list.
- We will see a way to make this totally generic later, and some of the related coding challenges.

`Pair` behaviors/properties

When created, a `Pair` object's constructor must be sent two references (to objects of the proper type).

Once a `Pair` object exists, in this class example, we have getters `getRefToFirst()` and `getRefToSecond()` that can be invoked using a reference to the `Pair` object.

Getting around the return limitation…

We now have a way to return more than one value from a method. Utilizing this `Pair` object, we can return a pair of object references with the return value of a method, circumventing the fact that Java only supports returning one value – we return a reference to a single object that contains references to two others!

This is another example of the power of objects.

Generics and Primitives

Recall that every primitive type (int, double, etc.) as an object-based wrapper version in Java (`Integer`, `Double`, etc) and that Java will do a variety of automatic boxing and unboxing of these.

We could make use of this aspect of Java by (for example) being able to create a `Pair` of `Double`.

Declaring and creating a `Pair`

```
//Declare a reference.
Pair<Student> twoStudents;

//Create some Student objects.
Student s1 = new Student("Pat");
Student s2 = new Student("Sam");

//Create a new Pair object to hold
//  references to them and store the
//  reference to that new object.
twoStudents = new Pair<Student>(s1,s2);
```

Using the `Pair`

Recall, that **`Pair`** has instance methods that allow you to access the items.

```
System.out.println("First was " +
  twoStudents.getRefToFirst());

System.out.println("Second was " +
  twoStudents.getRefToSecond());
```

Note, in my implementation the constructor and getters just copy the references. These are what we have been calling shallow copies.

Which do you think `Pair` should have?

- 0% A. Deep copies of the two values.
- 0% B. Setters like `setRefForFirst`.
- 0% C. The ability for a heterogeneous pair.