Generics

Contents

Overview

Generics were introduced with the aim of reducing bugs and adding an extra layer of abstraction over types.
Generics enforce type correctness at compile time and enable implementing generic algorithms without causing any extra overhead to our applications as we do not need to define a type, this will not however work with primitive data types.

Tutorial

Generics add stability to our code by making more bugs detectable at compile time.
If we have a method that takes an object, it will accept everything except primitive data types.
If we cast the object to an Integer at the end and then pass in a String, we will end up getting a runtime exception.

```
public class Cage<T> {

    private T object;

    public void add(T object) {
        this.object = object;
    }

    public T get() {
        return object;
    }
}
```

In the above example we are defining a class that uses generics, which is shown by the `<T>` in the class definition.
We also declare a variable called object of the generic type T, and have two methods add, which sets the class variable of object to the generic sent in, and get which returns the generic object in the class.

NOTE: It is known that Object is the supertype of all Java classes, however a collection of Object is not the supertype of any collection.
For example a `List<Object>` is not the supertype of `List<String>`.

Using generics for this class allows it to be used by various objects, so if we had a class of Elephant and a class of Lion, both could use the methods within the Cage class due to the fact that it is using generics.
However if the add method took a parameter of type Elephant, then the Lion class would no longer be able to use the add method.

Bounded Generics

Generics can be bounded, which means they can be restricted.
For example we can specify that a method accepts a type and all its subclasses (upper bound) or a type and its superclasses (lower bound).

Upper Bound

To declare an upper bound generic we can do the following.

```
import java.util.List;

public class Cage<T extends Animal> {

    private List<T> list;

    public void add(List<T> list) {
        this.list = list;
    }

    public List<T> get() {
        return list;
    }
}
```

The generics used within the class above will all be restricted to using types that are either the type Animal or a subclass of Animal.
Note that in this context, the extends keyword is used in a general sense to mean either extends (classes) or implements (interfaces).

We can also have multiple upper bounds, to do this simply use a "&" followed by the next upper bound.

```
import java.util.List;

public class Cage<T extends Mammal & Reptile> {

    private List<T> list;

    public void add(List<T> list) {
        this.list = list;
    }

    public List<T> get() {
        return list;
    }
}
```

In this case we will only allow the types Mammal and Reptile as well as their subclasses.

Lower Bound

Lower bound can be defined as shown below.

```
import java.util.List;

public class Cage {

    private List<? super Elephant> list;

    public void add(List<? super Elephant> list) {
        this.list = list;
    }

    public List<? super Elephant> get() {
        return list;
    }
}
```

With lower bound we don't define the generic in the class definition, instead wherever the generic would be defined we also need to define the lower bound.
The lower bound of `<? super Elephant>` means that we restrict the generic to an unknown class that is a superclass of Elephant.
The "?" within the generic specifies a wildcard and simply refers to an unknown type.

Exercises

There are no exercises for this module.