Unit 23: Generics

After taking this unit, students should:

- know how to define and instantiate a generic type and a generic method
- be familiar with the term parameterized types, type arguments, type parameters
- appreciate how generics can reduce duplication of code and improve type safety

The `Pair` class

Sometimes it is useful to have a lightweight class to bundle a pair of variables together. One could, for instance, write a method that returns two values. The example defines a class `IntPair` that bundles two `int` variables together. This is a utility class with no semantics nor methods associated with it and so, we did not attempt to hide the implementation details.

```
1  class IntPair {
2    private int first;
3    private int second;
4
5    public IntPair(int first, int second) {
6        this.first = first;
7        this.second = second;
8    }
9
10   int getFirst() {
11       return this.first;
12   }
13
14   int getSecond() {
15       return this.second;
16   }
17 }
```

This class can be used, for instance, in a function that returns two `int` values.

```
1  IntPair findMinMax(int[] array) {
2      int min = Integer.MAX_VALUE;  // stores the min
3      int max = Integer.MIN.VALUE; // stores the max
4      for (int i : array) {
5          if (i < min)  {
6              min = i;
7          }
8          if (i > max) {
```

```
            max = i;
        }
    }
    return new IntPair(min, max);
}
```

We could similarly define a pair class for two doubles (`DoublePair`), two booleans (`BooleanPair`), etc. In other situations, it is useful to define a pair class that bundles two variables of two different types, say, a `Customer` and a `ServiceCounter`; a `String` and an `int`; etc.

We should not, however, create one class for each possible combination of types. A better idea is to define a class that stores two `Object` references:

```
class Pair {
  private Object first;
  private Object second;

  public Pair(Object first, Object second) {
      this.first = first;
      this.second = second;
  }

  Object getFirst() {
      return this.first;
  }

  Object getSecond() {
      return this.second;
  }
}
```

At the cost of using a wrapper class in place of primitive types, we get a single class that can be used to store any type of values.

You might recall that we used a similar approach for our `contains` method to implement a general *method* that works for any type of object. Here, we are using this approach for a general *class* that encapsulates any type of object.

Unfortunately, the issues we faced with narrowing type conversion and potential run-time errors apply to the `Pair` class as well. Suppose that a function returns a `Pair` containing a `String` and an `Integer`, and we accidentally treat this as an `Integer` and a `String` instead, the compiler will not be able to detect the type mismatch and stop the program from crashing during run-time.

```
Pair foo() {
  return new Pair("hello", 4);
}

```

```
Pair p = foo();
Integer i = (Integer) p.getFirst(); // run-time ClassCastException
```

To reduce the risk of human error, what we need is a way to specify the following: suppose the type of `first` is S and type of `second` is T, then we want the return type of `getFirst` to be S and of `getSecond` to be T.

Generic Types

In Java and many other programming languages, the mechanism to do this is called generics or templates. Java allows us to define a *generic type* that takes other types as *type parameters*, just like how we can write methods that take in variables as parameters.

Declaring a Generic Type

Let's see how we can do this for `Pair`:

```
class Pair<S,T> {
  private S first;
  private T second;

  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  S getFirst() {
      return this.first;
  }

  T getSecond() {
      return this.second;
  }
}
```

We declare a generic type by specifying its type parameters between `<` and `>` when we declare the type. By convention, we use a single capital letter to name each type parameter. These type parameters are scoped within the definition of the type. In the example above, we have a generic class `Pair<S,T>` (read "pair of S and T") with `S` and `T` as type parameters. We use `S` and `T` as the type of the fields `first` and `second`. We ensure that `getFirst()` returns type `S` and `getSecond()` returns type `T`, so that the compiler will give an error if we mix up the types.

Note that the constructor is still declared as `Pair` (without the type parameters).

Using/Instanting a Generic Type

To use a generic type, we have to pass in *type arguments*, which itself can be a non-generic type, a generic type, or another type parameter that has been declared. Once a generic type is instantiated, it is called a *parameterized type*.

To avoid potential human errors leading to `ClassCastException` in the example above, we can use the generic version of `Pair` as follows, taking in two non-generic types:

```
Pair<String,Integer> foo() {
  return new Pair<String,Integer>("hello", 4);
}

Pair<String,Integer> p = foo();
Integer i = (Integer) p.getFirst(); // compile-time error
```

With the parameterized type `Pair<String,Integer>`, the return type of `getFirst` is bound to `String`, and the compiler now have enough type information to check and give us an error since we try to cast a `String` to an `Integer`.

Note that we use `Integer` instead of `int`, since *only reference types* can be used as type arguments.

Just like you can pass a parameter of a method to another method, we can pass the type parameter of a generic type to another:

```
class DictEntry<T> extends Pair<String,T> {
      :
}
```

We define a generic class called `DictEntry<T>` with a single type parameter `T` that extends from `Pair<String,T>`, where `String` is the first type argument (in place of `S`), while the type parameter `T` from `DictEntry<T>` is passed as the type argument for `T` of `Pair<String,T>`.

Generic Methods

Methods can be parameterized with a type parameter as well. Consider the `contains` method, which we now put within a class for clarity.

```
class A {
    // version 0.1 (with polymorphism)
    public static boolean contains(Object[] array, Object obj) {
      for (Object curr : array) {
        if (curr.equals(obj)) {
          return true;
        }
      }
      return false;
```

```
    }
}
```

While using this method does not involve narrowing type conversion and type casting, it is a little to general -- it allows us to call `contains` in a nonsensical way, like this:

```
String[] strArray = new String[] { "hello", "world" };
A.contains(strArray, 123);
```

Searching for an integer within an array of strings is a futile attempt! Let's constrain the type of the object to search for to be the same as the type of the array. We can make this type the parameter to this method:

```
class A {
    // version 0.4 (with generics)
    public static <T> boolean contains(T[] array, T obj) {
      for (T curr : array) {
        if (curr.equals(obj)) {
          return true;
        }
      }
      return false;
    }
}
```

The above shows an example of a *generic method*. The type parameter `T` is declared within `<` and `>` and is added before the return type of the method. This parameter `T` is then scoped within the whole method.

To call a generic method, we need to pass in the type argument placed before the name of the method[1]. For instance,

```
String[] strArray = new String[] { "hello", "world" };
A.<String>contains(strArray, 123); // type mismatch error
```

The code above won't compile since the compiler expects the second argument to also be a `String`.

Bounded Type Parameters

Let's now try to apply our newly acquired trick to fix the issue with `findLargest`. Recall that we have the following `findLargest` method (which we now put into an ad hoc class just for clarity), which requires us to perform a narrowing type conversion to cast from `GetAreable` and possibly leading to a run-time error.

```
class A {
    // version 0.4
    public static GetAreable findLargest(GetAreable[] array) {
      double maxArea = 0;
      GetAreable maxObj = null;
      for (GetAreable curr : array) {
        double area = curr.getArea();
        if (area > maxArea) {
          maxArea = area;
          maxObj = curr;
        }
      }
      return maxObj;
    }
}
```

Let's try to make this method generic, by forcing the return type to be the same as the type of the elements in the input array,

```
class A {
    // version 0.4
    public static <T> T findLargest(T[] array) {
        double maxArea = 0;
        T maxObj = null;
        for (T curr : array) {
            double area = curr.getArea();
            if (area > maxArea) {
                maxArea = area;
                maxObj = curr;
            }
        }
        return maxObj;
    }
}
```

The code above won't compile, since the compiler cannot be sure that it can find the method `getArea()` in type `T`. In contrast, when we run `contains`, we had no issue since we are invoking the method `equals`, which exists in any reference type in Java.

Since we intend to use `findLargest` only in classes that implement the `GetAreable` interface and supports the `getArea()` method, we can put a constraint on `T`. We can say that `T` must be a subtype of `GetAreable` when we specify the type parameter:

```
class A {
    // version 0.5
    public static <T extends GetAreable> T findLargest(T[] array) {
        double maxArea = 0;
        T maxObj = null;
        for (T curr : array) {
            double area = curr.getArea();
            if (area > maxArea) {
```

```
                maxArea = area;
                maxObj = curr;
            }
        }
        return maxObj;
    }
}
```

We use the keyword `extends` here to indicate that `T` must be a subtype of `GetAreable`. It is unfortunate that Java decides to use the term `extends` for any type of subtyping when declaring a bounded type parameter, even if the supertype (such as `GetAreable`) is an interface.

We can use bounded type parameters for declaring generic classes as well. For instance, Java has a generic interface `Comparable<T>`, which dictates the implementation of the following `int compareTo(T t)` for any concrete class that implements the interface. Any class that implements the `Comparable<T>` interface can be compared with an instance of type `T` to establish an ordering. Such ordering can be useful for sorting objects, for instance.

Suppose we want to compare two `Pair` instances, by comparing the first element in the pair, we could do the following:

```
class Pair<S extends Comparable<S>,T> implements Comparable<Pair<S,T>> {
  private S first;
  private T second;

  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  S getFirst() {
      return this.first;
  }

  T getSecond() {
      return this.second;
  }

  @Override
  public int compareTo(Pair<S,T> s1) {
      return this.first.compareTo(s1.first);
  }

  @Override
  public String toString() {
    return this.first + " " + this.second;
  }
}
```

Let's look at what it means:

- We declared `Pair` to be a generic type of two type parameters: the first one `S` is bounded and must be a subtype of `Comparable<S>`. This bound is self-referential, but it is intuitive -- we say that `S` must be comparable to itself, which is common in many use cases.
- Since we want to compare two `Pair` instances, we make `Pair` implements the `Comparable` interface too, passing in `Pair<S,T>` as the type argument to `Comparable`.

Let's see this in action with `Arrays::sort` method, which sorts an array based on the ordering defined by `compareTo`.

```
1   Object[] array = new Object[] {
2     new Pair<String,Integer>("Alice", 1),
3     new Pair<String,Integer>("Carol", 2),
4     new Pair<String,Integer>("Bob", 3),
5     new Pair<String,Integer>("Dave", 4),
6   };
7
8   java.util.Arrays.sort(array);
9
10  for (Object o : array) {
11    System.out.println(o);
12  }
```

You will see the pairs are sorted by the first element.

---

1. Java actually can infer the type using the *type inference* mechanism and allows us to skip the type argument, but for clarity, we insist on specifying the type explicitly until students get used to the generic types and reasoning about types. ↵