Unit 24: Type Erasure

After taking this unit, students are expected to:

- understand that generics are implemented with type erasure in Java
- understand that type information is not fully available during run-time when generics are used, and problems that this could cause
- be aware that arrays and generics don't mix well in Java
- know the terms reifiable type and heap pollution.

Implementing Generics

There are several ways one could implement generics in a programming language.

For instance, in C#, every instantiation of a generic type causes new code to be generated for that instantiated type. For instance, instantiating `Pair<S,T>` into `Pair<String,Integer>` causes a new type to be generated during run-time. In C++ and in Rust, instantiating `Pair<String,Integer>` causes new code to be generated during compile-time. This approach is sometimes called *code specialization*.

Java takes a *code sharing* approach, instead of creating a new type for every instantiation, it chooses to *erase* the type parameters and type arguments during compilation (after type checking, of course). Thus, there is only one representation of the generic type in the generated code, representing all the instantiated generic types, regardless of the type arguments.

Part of the reason to do this is for compatibility with the older version of Java. Java introduces generics only from version 5 onwards. Prior to version 5, one has to use `Object` to implement classes that are general enough to works on multiple types, similar to what we did with `Pair` here:

```
1  class Pair {
2    private Object first;
3    private Object second;
4
5    public Pair(Object first, Object second) {
6        this.first = first;
7        this.second = second;
8    }
9
```

```
    Object getFirst() {
        return this.first;
    }

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

The Java type erasure process transforms:

```
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;
    }
}
```

to the version above. Note that each type parameter `S` and `T` are replaced with `Object`. If the type parameter is bounded, it is replaced by the bounds instead (e.g., If `T extends GetAreable`, then `T` is replaced with `GetAreable`).

Where a generic type is instantiated and used, the code

```
Integer i = new Pair<String,Integer>("hello", 4).getSecond();
```

is transformed into

```
Integer i = (Integer) new Pair("hello", 4).getSecond();
```

The generated code is similar to what we would write earlier, but this is generated by the compiler after type checking, it ensures that the casting will not lead to `ClassCastException` during run-time.

Type erasures have several important implications. We will explore some of them below, and a few others during recitation.

Generics and Arrays Can't Mix

Let's consider the hypothetical code below:

```
// create a new array of pairs
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];

// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair<Double,Boolean>(3.14, true);
```

This is similar to what we have in Unit 21, where we showed we could get an `ArrayStoreException` due to Java arrays being covariant. We would not, however, get an exception when we try to put a pair of double and boolean, into an array meant to store a pair of string and integer! This type checking is done during run-time, and due to type erasure, the run-time has no information about what is the type arguments to `Pair`. The run-time sees:

```
// create a new array of pairs
Pair[] pairArray = new Pair[2];

// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair(3.14, true);
```

It checks that we have an array of pairs and we are putting another pair inside. Everything checks out. This would have caused a *heap pollution*, a term that refers to the situation where a variable of a parameterized type refers to an object that is not of that parameterized type.

Heap pollution is dangerous, as now, we will get a `ClassCastException` when we do:

```
// getting back a string?  -- now we get ClassCastException
String str = pairArray[0].getFirst();
```

The example above shows why generics and arrays don't mix well together. An array is what is called *reifiable* type -- a type where full type information is available during run-time. It is because Java array is reifiable that the Java run-time can check what we store into the array matches the type of the array and throw an `ArrayStoreException` at us if there is a mismatch. Java generics, however, is not reifiable due to type erasure. Java designers have decided not to mix the two.

The hypothetical code above actually is not a valid Java syntax. We can't compile this line:

```
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];
```

The following is illegal as well:

```
new Pair<S,T>[2];
new T[2];
```