Java Generics

Lab Setup

- Create a new Java project called "JavaGenerics"
- Add a main class to the project

Generic programming

Generic programming is a paradigm in which algorithms or data structures are defined generically. This means that they can work with any type.

Generic programming allows for code reuse, and in Java, stronger type checking at compile time.

Generics

We can define our own generic classes, methods, and interfaces.

After the name of the class, we add a type parameter surrounded by angled brackets.

```
public class Box<T> {
    public T item;

    public Box(T item) {
        this.item = item;
    }
};
```

Figure 1 Generic class Box which can hold any type T.

The `T` here stands for Type. The scope of the type parameter is the entire class and the parameter can have any name.

Naming conventions for type parameters

By convention, type parameters are single, uppercase letters. The most common names and uses are shown in Figure 2.

E
Element (as in element in a collection)

K
Key

N
Number

T
Type

V
Value

S,U,V etc.
2nd, 3rd, 4th types

Figure 2 Naming conventions for type parameters.

Invoking and instantiating

Generic type invocation is replacing the type parameter with a concrete value. For example, in Figure 3 we instantiate a `Box` by passing `String` as the type argument.

```
Box<String> secret = new Box<String>("secret");
```

Figure 3 Instantiation of Box for type String

The compiler may be able to infer the type of the type argument allowing statements like the one shown in Figure 4.

The second use of the type argument is omitted and `secret` is still a parameterized type.

```
Box<String> secret = new Box<>("secret");
```

Figure 4 Instantiation of Box for type String with type inference.

Interfaces

Interfaces are defined to be generic the same way classes are. In the following examples we will define the interface Pair and an implementing class OrderedPair.

Graded Submission Task 1

1. Define an interface, `Pair`, with two type parameters. `Pair` can also be used for dictionaries, so we will use `K` and `V`.

2. `Pair` has two methods, `getKey()` and `getValue()` the return types of these methods are `K` and `V` respectively

3. Add a class called `OrderedPair` which implements `Pair`. This class is also generic, and we use the same names for the type parameters as we did for `Pair`

4. Instantiate two `OrderedPairs` in the main method. Use a `Pair` variable for one and a `OrderedPair` variable for the other as shown in Figure 5.

5. Print each Pair to the console.

```
Pair<Integer, Integer> p0 = new OrderedPair<Integer,Integer>(0,0);
OrderedPair<Integer, String> p1 = new OrderedPair<Integer, String>(0, "Hello");
```

Figure 5 OrderedPair instantiated from Pair and OrderedPair variables.

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 1. You will submit this copy at the end of this lab.

Methods

Syntax for generic methods has the type parameters before the method's return type.

```
public static <K, V> boolean compare(Pair<K,V> p1, Pair<K,V> p2) {
    return
(p1.getValue().equals(p2.getValue()) && p1.getKey().equals(p2.getKey()));
}
```

Figure 6 Generic method for comparing Pairs with the same two types.

```
Pair<Integer, String> p1 = new OrderedPair<>(0, "Hi");
Pair<Integer, String> p2 = new OrderedPair<>(1, "There");
```

Figure 7 Instantiation of Pairs p1 and p2

When invoking the method, we provide type arguments between before the function name as shown in Figure 8.

```
boolean sames = <Integer, String>compare(p1, p2);
```

Figure 8 Invocation of generic method compare.

We can also take advantage of type inference to omit the type arguments as shown in Figure 9.

```
sames = compare(p1,p2);
```

Figure 9 Same as Figure 8 using type inference

Type Erasure

During compile time, the Java compiler applies type erasure. It will replace type parameters with their bounds (discussed in later lab), or with Object if they are unbound. Figure 14a shows the effect of type erasure for the Box class. The type parameter was unbound so it would be replaced by Object at compile time. Figure 14e shows type erasure of a generic method. Like Box, the type parameters were not bounded, so they are replaced by Object. Figures 14c and 14d show the effect of type erasure on a bounded type parameter. The type parameter for NumberBox is upper-bounded by Number so at compile time it is replaced with Number. In Figure 14b, type erasure causes a compilation error because both filter methods will have the same method signature.

```java
public class Box<Object> {
    public Object item;

    public Box(Object item) {
        this.item = item;
    }
}
```

Figure 14a Type erasure of Box generic class

```java
public void filter(List<NumberBox> list) {
    //...
}

public void filter(List<Box> list) {
    //...
}
```

Figure 14b Effect of type erasure on method overloading

```java
public class NumberBox<T extends Number> {
    public T number;

    public NumberBox(T number) {
        this.number = number;
    }
}
```

Figure 14c NumberBox bounded generic class

```java
public class NumberBox {
    public Number number;

    public NumberBox(Number number) {
        this.number = number;
    }
}
```

Figure 14d Type erasure for NumberBox bounded generic class

```java
public static boolean compare(Pair<Object, Object> p1, Pair<Object, Object> p2)
```

Figure 14e Type erasure of compare generic method

Lab wrap-up

In this lab we discussed generic programming and generics in Java. Generic programming is a paradigm in which algorithms and data structures are defined so that they work with any type, which allows for code reuse. We have already seen generic types in the Collections Framework.

A generic class, method, or interface is defined using angled brackets. In the angled brackets we include a type parameter. Typical naming conventions for type parameters are shown in Figure 2. Invoking and instantiating a generic type requires providing a type argument. In Figures 3 and 4 we see two different ways of instanting a Box<String>. Figure 4 uses type inference to omit the second instance of the type argument.

Java also implements type erasure. During compile time, the compiler will replace type parameters with their bounds, or with Object if they are unbound. Figures 10-14 show examples of type erasure for different generic types.

Submission notes for graded tasks:

- Submit a zipped folder containing the Eclipse projects for each graded task.
- For each project, ensure you have included the src folder containing the .java file, the .classpath file, and the .project file.
- Ensure that each project is named appropriately to correspond with each task.
- Zip the projects together and name the zipped folder “Lab17_firstName_lastName”.
- Submit the zipped file to the respective Canvas link.

Rubrics
Task 1 (points allotted) - Compilation (2) - Pair interface defined (2) - OrderedPair implemented (2) - Driver demonstrates implementation (4)