Hands-On: Generics

This activity is designed to reinforce the concepts of type variables, parameterized types, generic types, type safety, and related concepts. In working through this activity, you may find it useful to configure your IDE to provide more detailed messages concerning type safety warnings. You can configure jGRASP to provide this information be doing the following:

1. Click on *Settings* > *Compiler Settings* > *Workspace* in the jGRASP menu bar.
2. Click on the *Flags/Args* tab.
3. Under *FLAGS or ARGS* (the first column), click on the selector box for *Compile*.
4. In the *Compile* text entry box, enter: `-Xlint:unchecked`
5. Click *Apply* and then *OK*.

Generic methods

A method is *generic* if it declares one or more *type variables*. In this portion of the activity, you will make two different methods generic.

Making the `search` method generic

1. Open `GenericsA.java` in jGRASP then compile and run it.
2. Make the `search` method generic. Specifically:
   - Add a type variable named `T` to the method.
   - Declare the parameters to be of this type.
   - Change the inequality comparison from `!=` to negation of the `equals` method return value.

Your code should appear as follows when finished:

```
public static <T> int search(T[] a, T target) {
    int i = 0;
    while ((i < a.length) && (!a[i].equals(target))) {
       i++;
    }
    if (i < a.length) {
       return i;
    }
    else {
       return -1;
```

```
    }
}
```

3. Compile `GenericsA` and note the error that now results.

```
GenericsA.java:32: error: method search in class GenericsA cannot be applied to give
        int i = GenericsA.search(a1, 8);
                         ^
  required: T[],int
  found:    int[],int
  reason: inference variable T has incompatible bounds
    equality constraints: int
    lower bounds: Object
  where T is a type-variable:
    T extends Object declared in method <T>search(T[],int)
1 error
```

   - This error message is long, confusing, and difficult to sort out. There is more than one way to proceed from here, but we're going to proceed in a way that best lends itself to the learning goals of this activity.

4. Bind a value to the type variable when the `search` method is called. We can't use primitive types for generic type variables, so we'll use the `Integer` type instead.

   - Change the method call to the following:

```
int i = GenericsA.<Integer>search(a1, 8);
```

5. Compile `GenericsA` and note the error that results.

   Now the error message is easier to understand. It's clearly a typing error that involves the `a1` parameter that we're passing to the `search` method. It turns out that autoboxing does not apply to arrays. That is, `int` is autoboxed to `Integer`, but `int[]` is **not** autoboxed to `Integer[]`.

6. Eliminate the incompatible types error.

   - Change the declaration of `a1` to:

```
Integer[] a1 = {4, 10, 2, 8, 6};
```

7. Compile and run `GenericsA`. The `search` method is now generic and it is being called in a type-safe manner.

Making the `min` method generic

1. Open `GenericsB.java` in jGRASP then compile it.

2. Read the type safety warning that the compiler reports, and examine the source of the warning in the code. Make sure you can answer the following questions.

    - What does the compiler mean by an "**unchecked call**"?
    - What is a "**raw type**"?
    - Why does the compiler refer to `Comparable` as a raw type?

3. Since there was only a warning and no error reported by the compiler, bytecode was generated and you can execute `GenericsB` . Run this program and observe the results. Make sure you can answer the following question.

    - If the compiler warned that the program was not type-safe, why were there no runtime errors?

4. Make sure you understand why we used `Comparable` for the parameter's type in the first place.

    - To find the minimum, we have to be able to compare the elements in the array on the basis of "less than or equal to." That is, we have to compare the elements in the array based on a defined *total order*. If value *x* precedes *y* in a total order, the value *x* is *less than* the value *y*. It doesn't matter what type *x* and *y* are, this definition of *less than* is true for all types with a total order. The `Comparable` interface is designed to serve as a marker or contract for any reference type that defines a total order on its values. So whatever type is actually in the array, it must implement the `Comparable` interface.

5. Declare a type variable for the `min` method, and change the parameter and local variables to be of that type. The method should appear as follows:

```
public static <T extends Comparable> T min(T[] a) {
    T min = a[0];
    for (T val : a) {
       if (val.compareTo(min) < 0) {
          min = val;
       }
    }
    return min;
}
```

6. Bind a value to the type variable when the `min` method is called. To ensure that we're not using raw types anywhere:

    - Change the first method call to the following:

```
Comparable min1 = GenericsB.<Integer>min(a2);
```

- Change the second method call to the following:

```
Comparable min2 = GenericsB.<String>min(a3);
```

7. Compile `GenericsB` and notice that the type safety warning is still there.
   - Make sure you understand *why* the compiler still warns us that this code isn't type-safe. Specifically, reflect on the meaning of a *raw type* and why the compiler is telling us that we're still using one.
8. Place an *upper bound* on this type variable to ensure that it implements the `Comparable` interface for its type. Then replace the use of `Comparable` with the generic type variable. The method signature should appear as follows:

```
public static <T extends Comparable<T>> T min(T[] a)
```

9. Compile `GenericsB` and notice that there are now no warnings; the code is type-safe.
   - Again, make sure you understand why this last step was necessary to ensure type-safety.

Generic types

A class or interface that declares one or more generic variables is called a *generic type*. In this portion of the activity, you will make a class generic.

1. Open `GenericsC.java` in jGRASP then compile it.
   - At this point you should be familiar with the two type-safety warnings given by the compiler. You should be able to understand the source of the error: the use of the raw types `List` and `Collection`.
   - Since the `List` being declared (`al`) is a field of the `GenericsC` class, we will want to make the class itself generic in order to achieve the generality that we want.
2. Declare a type variable for the class.
   - The class "signature" should appear as follows:

```
public class GenericsC<T>
```

   - Notice how the declaration of the type variable (i.e., `<T>`) is placed differently for classes and interfaces than for methods. For classes and interfaces, the type variable declaration immediately follows the class name.
3. Use this type variable as the *value* for the type parameter of the `List` field `al`.

- The declaration should appear as follows:

```
private List<T> al;
```

4. Compile `GenericsC.java` and notice the results.
   - One warning is the same as before (unchecked call to `add` for the raw type `Collection`), but the other warning has changed. Worse, though, we now have a type **error**, which means our code isn't just unsafe: **It's incorrect.**
   - Let's try to correct the new warning first (unchecked conversion of `ArrayList`). Since `ArrayList` is a generic type, we can try to provide a type value for its type variable.
5. Use the type variable that we declared for this class as the value of the type parameter for `ArrayList1`. Your code should appear as follows:

```
al = new ArrayList<T>();
```

6. Compile `GenericsC.java` and notice the results.
   - This did indeed eliminate the type warning regarding `ArrayList`, but the other warning and the error remain. Let's tackle the error now.
7. Read the error message carefully and try to understand **why** `al.add(o)` is a problem.

```
javac -Xlint:unchecked GenericsC.java
GenericsC.java:32: error: incompatible types: Object cannot be converted to T
            al.add(o);
                   ^
  where T is a type-variable:
    T extends Object declared in class GenericsC
GenericsC.java:51: warning: [unchecked] unchecked call to add(E) as a member of the
            c.add(i);
                 ^
  where E is a type-variable:
    E extends Object declared in interface Collection
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
1 error
1 warning
```

   - This is a typing error because `al` has been declared as a `List` with elements of type `T` (i.e., `List<T>`). Therefore, the only type of elements than can be added to `al` are *elements of the actual type value used as the T parameter when the class is instantiated.* Therefore, since `o` is typed as `Object` it can't be added to `al`. **Make sure you understand this.**

8. Fix this error by (1) using the parameterized type `Collection<T>` instead of the raw type `Collection` and (2) declaring `o` to be of type `T` instead of `Object`. Your code should appear as follows.

```
public void addAll(Collection<T> c) {
    for (T o : c) {
       al.add(o);
    }
}
```

9. Compile `GenericsC.java` and confirm that this change eliminated the error.
    - Now we're left with two warnings (Note that one is new. *Why?*) Both can be eliminated by **using appropriate parameterized types instead of raw types** in the `main` method.
10. Change the declaration of `c` to be of an appropriate parameterized type. Your code should appear as follows:

```
Collection<Integer> c = new ArrayList<Integer>();
```

    - Recompile and note that one warning has been eliminated.
11. Change the declaration of `lab` to be of an appropriate parameterized type. Your code should appear as follows:

```
GenericsC<Integer> lab = new GenericsC<Integer>();
```

    - Recompile and note that the final warning has been eliminated and the code is now type-safe.

Summary

Generic typing is the accepted way to write general, type-safe code in Java. Since you will use generics from this point forward, it's very important that you not only grasp the high-level idea but also the low-level details. If you need to work through this activity again to make sure you understand everything that's here, please do so.

Submission

The submission page for this activity asks you to submit each of the three parts of this activity for a grade.