Documentation

The Java™ Tutorials

Trail: Learning the Java Language
Lesson: Generics (Updated)
Section: Wildcards

Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as `List<?>` but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as *wildcard capture*.

For the most part, you don't need to worry about wildcard capture, except when you see an error message that contains the phrase "capture of".

The `WildcardError` example produces a capture error when compiled:

```
import java.util.List;

public class WildcardError {

    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}
```

In this example, the compiler processes the `i` input parameter as being of type `Object`. When the `foo` method invokes List.set(int, E), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason — to enforce type safety at compile time.

The `WildcardError` example generates the following error when compiled by Oracle's JDK 7 `javac` implementation:

```
WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;
    i.set(0, i.get(0));
     ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a *private helper method* which captures the wildcard. In this case, you can work around the problem by creating the private helper method, `fooHelper`, as shown in `WildcardFixed`:

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }


    // Helper method created so that the wildcard can be captured
    // through type inference.
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }
}
```

Thanks to the helper method, the compiler uses inference to determine that `T` is `CAP#1`, the capture variable, in the invocation. The example now compiles successfully.

By convention, helper methods are generally named *`originalMethodName`*`Helper`.

Now consider a more complex example, `WildcardErrorBad`:

```
import java.util.List;

public class WildcardErrorBad {

    void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {
      Number temp = l1.get(0);
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
                            // got a CAP#2 extends Number;
                            // same bound, but different types
      l2.set(0, temp);      // expected a CAP#1 extends Number,
                            // got a Number
    }
}
```

In this example, the code is attempting an unsafe operation. For example, consider the following invocation of the `swapFirst` method:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double>  ld = Arrays.asList(10.10, 20.20, 30.30);
swapFirst(li, ld);
```

While `List<Integer>` and `List<Double>` both fulfill the criteria of `List<? extends Number>`, it is clearly incorrect to take an item from a list of `Integer` values and attempt to place it into a list of `Double` values.

Compiling the code with Oracle's JDK `javac` compiler produces the following error:

```
WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given types;
      l2.set(0, temp);      // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given types;
        i.set(0, i.get(0));
         ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
3 errors
```

There is no helper method to work around the problem, because the code is fundamentally wrong.

Previous page: Wildcards and Subtyping
Next page: Guidelines for Wildcard Use