Java SE Documentation

# Generics

When you take an element out of a `Collection`, you must cast it to the type of element that is stored in the collection. Besides being inconvenient, this is unsafe. The compiler does not check that your cast is the same as the collection's type, so the cast can fail at run time.

Generics provides a way for you to communicate the type of a collection to the compiler, so that it can be checked. Once the compiler knows the element type of the collection, the compiler can check that you have used the collection consistently and can insert the correct casts on values being taken out of the collection.

Here is a simple example taken from the existing Collections tutorial:

```
// Removes 4-letter words from c. Elements must be strings
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
      if (((String) i.next()).length() == 4)
        i.remove();
}
```

Here is the same example modified to use generics:

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
      if (i.next().length() == 4)
        i.remove();
}
```

When you see the code `<Type>`, read it as "of `Type`"; the declaration above reads as "`Collection` of `String c`." The code using generics is clearer and safer. We have eliminated an unsafe cast and a number of extra parentheses. More importantly, we have moved part of the specification of the method from a comment to its signature, so the compiler can verify at compile time that the type constraints are not violated at run time. Because the program compiles without warnings, we can state with certainty that it will not throw a `ClassCastException` at run time. The net effect of using generics, especially in large programs, is improved readability and robustness.

To paraphrase Generics Specification Lead Gilad Bracha, when we declare `c` to be of type `Collection<String>`, this tells us something about the variable `c` that holds true wherever and whenever it is used, and the compiler guarantees it (assuming the program compiles without warnings). A cast, on the other hand, tells us something the programmer thinks is true at a single point in the code, and the VM checks whether the programmer is right only at run time.

While the primary use of generics is collections, there are many other uses. "Holder classes," such as `WeakReference` and `ThreadLocal`, have all been *generified*, that is, they have been retrofitted to make use of generics. More surprisingly, class `Class` has been generified. Class literals now function as *type tokens*, providing both run-time and compile-time type information. This enables a style of static factories exemplified by the `getAnnotation` method in the new `AnnotatedElement` interface:

```
<T extends Annotation> T getAnnotation(Class<T> annotationType);
```

This is a *generic method*. It infers the value of its *type parameter* `T` from its argument, and returns an appropriate instance of `T`, as illustrated by the following snippet:

```
Author a = Othello.class.getAnnotation(Author.class);
```

Prior to generics, you would have had to cast the result to `Author`. Also you would have had no way to make the compiler check that the actual parameter represented a subclass of `Annotation`.

Generics are implemented by *type erasure*: generic type information is present only at compile time, after which it is *erased* by the compiler. The main advantage of this approach is that it provides total interoperability between generic code and legacy code that uses non-parameterized types (which are technically known as *raw* types). The main disadvantages are that parameter type information is not available at run time, and that automatically generated casts may fail when interoperating with ill-behaved legacy code. There is, however, a way to achieve guaranteed run-time type safety for generic collections even when interoperating with ill-behaved legacy code.

The `java.util.Collections` class has been outfitted with wrapper classes that provide guaranteed run-time type safety. They are similar in structure to the synchronized and unmodifiable wrappers. These "checked collection wrappers" are very useful for debugging. Suppose you have a set of strings, `s`, into which some legacy code is mysteriously inserting an integer. Without the wrapper, you will not find out about the problem until you read the problem element from the set, and an automatically generated cast to `String` fails. At this point, it is too late to determine the source of the problem. If, however, you replace the declaration:

```
Set<String> s = new HashSet<String>();
```

with this declaration:

```
Set<String> s = Collections.checkedSet(new HashSet<String>(), String.class);
```

the collection will throw a `ClassCastException` at the point where the legacy code attempts to insert the integer. The resulting stack trace will allow you to diagnose and repair the problem.

You should use generics everywhere you can. The extra effort in generifying code is well worth the gains in clarity and type safety. It is straightforward to use a generic library, but it requires some expertise to write a generic library, or to generify an existing library. There is one caveat: You may not use generics (or any other Tiger features) if you intend to deploy the compiled code on a pre-5.0 virtual machine.

If you are familiar with C++'s *template* mechanism, you might think that generics are similar, but the similarity is superficial. Generics do not generate a new class for each specialization, nor do they permit "template metaprogramming."

There is much more to learn about generics. See the Generics lesson in the Java Tutorials.

Contact Us