# From Oracle Java Trail

# Lesson: Generics (Updated)

In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

Generics add stability to your code by making more of your bugs detectable at compile time. After completing this lesson, you may want to follow up with the [Generics](http://docs.oracle.com/javase/tutorial/extra/generics/index.html) tutorial by Gilad Bracha.

# Why Use Generics?

In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

* Stronger type checks at compile time.  
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* Elimination of casts.  
  The following code snippet without generics requires casting:
* List list = new ArrayList();
* list.add("hello");
* String s = **(String)** list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

* Enabling programmers to implement generic algorithms.  
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

**From Bracha……**

**Introduction**

JDK 5.0 introduces several new extensions to the Java programming language. One of these is the introduction of *generics*.

This trail is an introduction to generics. You may be familiar with similar constructs from other languages, most notably C++ templates. If so, you'll see that there are both similarities and important differences. If you are unfamiliar with look-a-alike constructs from elsewhere, all the better; you can start fresh, without having to unlearn any misconceptions.

Generics allow you to abstract over types. The most common examples are container types, such as those in the Collections hierarchy.

Here is a typical usage of that sort:

List myIntList = new LinkedList(); // 1

myIntList.add(new Integer(0)); // 2

Integer x = (Integer) myIntList.iterator().next(); // 3

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required.

Of course, the cast not only introduces clutter. It also introduces the possibility of a run time error, since the programmer may be mistaken. **(For example, there is nothing in the above code to prevent myIntList.add(“Hello”);**

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

List<Integer> myIntList = new LinkedList<Integer>(); // 1'

myIntList.add(new Integer(0)); // 2'

Integer x = myIntList.iterator().next(); // 3'

Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>. We say that List is a generic interface that takes a *type parameter*--in this case, Integer. We also specify a type parameter when creating the list object.

Note, too, that the cast on line 3' is gone.

Now, you might think that all we've accomplished is to move the clutter around. Instead of a cast to Integer on line 3, we have Integer as a type parameter on line 1'. However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that myIntList is declared with type List<Integer>, this tells us something about the variable myIntList, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code.

The net effect, especially in large programs, is improved readability and robustness.

<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

http://docs.oracle.com/javase/tutorial/extra/generics/index.html