The Basics of Java Generics

Last modified: January 13, 2023

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Java (https://www.baeldung.com/category/java) +

Core Java (https://www.baeldung.com/tag/core-java)

1. Overview

JDK 5.0 introduced Java Generics with the aim of reducing bugs and adding an extra layer of abstraction over types.

This tutorial is a quick intro to Generics in Java, the goal behind them and how they can improve the quality of our code.

Further reading:

Method References in Java (/java-method-references)

A quick and practical overview of method references in Java.

Read more (/java-method-references) →

Retrieve Fields from a Java Class Using Reflection (/java-reflection-class-fields)

Learn how to get the fields of a class using reflection, including inherited fields

Read more (/java-reflection-class-fields) →

2. The Need for Generics

Let's imagine a scenario where we want to create a list in Java to store *Integer*.

We might try to write the following:

```
List list = new LinkedList();
list.add(new Integer(1));
Integer i = list.iterator().next();
```

Surprisingly, the compiler will complain about the last line. It doesn't know what data type is returned.

The compiler will require an explicit casting:

```
Integer i = (Integer) list.iterator.next();
```

There is no contract that could guarantee that the return type of the list is an *Integer*. The defined list could hold any object. We only know that we are retrieving a list by inspecting the context. When looking at types, it can only guarantee that it is an *Object* and therefore requires an explicit cast to ensure that the type is safe.

This cast can be annoying — we know that the data type in this list is an *Integer*. The cast is also cluttering our code. It can cause type-related runtime errors if a programmer makes a mistake with the explicit casting.

It would be much easier if programmers could express their intention to use specific types and the compiler ensured the correctness of such types. This is the core idea behind generics.

Let's modify the first line of the previous code snippet:

```
List<Integer> list = new LinkedList<>();
```

By adding the diamond operator <> containing the type, we narrow the specialization of this list to only *Integer* type. In other words, we specify the type held inside the list. The compiler can enforce the type at compile time.

In small programs, this might seem like a trivial addition. But in larger programs, this can add significant robustness and makes the program easier to read.

3. Generic Methods

We write generic methods with a single method declaration, and we can call them with arguments of different types. The compiler will ensure the correctness of whichever type we use.

These are some properties of generic methods:

- Generic methods have a type parameter (the diamond operator enclosing the type) before the return type of the method declaration.
- Type parameters can be bounded (we explain bounds later in this article).
- Generic methods can have different type parameters separated by commas in the method signature.
- Method body for a generic method is just like a normal method.

Here's an example of defining a generic method to convert an array to a list:

```java
public <T> List<T> fromArrayToList(T[] a) {
    return Arrays.stream(a).collect(Collectors.toList());
}
```

The *<T>* in the method signature implies that the method will be dealing with generic type *T*. This is needed even if the method is returning void.

As mentioned, the method can deal with more than one generic type. Where this is the case, we must add all generic types to the method signature.

Here is how we would modify the above method to deal with type *T* and type *G*:

```java
public static <T, G> List<G> fromArrayToList(T[] a, Function<T, G>
mapperFunction) {
    return Arrays.stream(a)
      .map(mapperFunction)
      .collect(Collectors.toList());
}
```

We're passing a function that converts an array with the elements of type *T* to list with elements of type *G*.

An example would be to convert *Integer* to its *String* representation:

```java
@Test
public void givenArrayOfIntegers_thanListOfStringReturnedOK() {
    Integer[] intArray = {1, 2, 3, 4, 5};
    List<String> stringList
      = Generics.fromArrayToList(intArray, Object::toString);

    assertThat(stringList, hasItems("1", "2", "3", "4", "5"));
}
```

Note that Oracle recommendation is to use an uppercase letter to represent a generic type and to choose a more descriptive letter to represent formal types. In Java Collections, we use *T* for type, *K* for key and *V* for value.

3.1. Bounded Generics

Remember that type parameters can be bounded. Bounded means "restricted," and we can restrict the types that a method accepts.

For example, we can specify that a method accepts a type and all its subclasses (upper bound) or a type and all its superclasses (lower bound).

To declare an upper-bounded type, we use the keyword *extends* after the type, followed by the upper bound that we want to use:

```
public <T extends Number> List<T> fromArrayToList(T[] a) {
    ...
}
```

We use the keyword *extends* here to mean that the type *T* extends the upper bound in case of a class or implements an upper bound in case of an interface.

3.2. Multiple Bounds

A type can also have multiple upper bounds:

```
<T extends Number & Comparable>
```

If one of the types that are extended by *T* is a class (e.g. *Number*), we have to put it first in the list of bounds. Otherwise, it will cause a compile-time error.

4. Using Wildcards With Generics

Wildcards are represented by the question mark *?* in Java, and we use them to refer to an unknown type. Wildcards are particularly useful with generics and can be used as a parameter type.

But first, there is an important note to consider. **We know that *Object* is the supertype of all Java classes.** However, a collection of *Object* is not the supertype of any collection.

For example, a *List<Object>* is not the supertype of *List<String>*, and assigning a variable of type *List<Object>* to a variable of type *List<String>* will cause a compiler error. This is to prevent possible conflicts that can happen if we add heterogeneous types to the same collection.

The same rule applies to any collection of a type and its subtypes.

Consider this example:

```
public static void paintAllBuildings(List<Building> buildings) {
    buildings.forEach(Building::paint);
}
```

If we imagine a subtype of *Building*, such as a *House*, we can't use this method with a list of *House*, even though *House* is a subtype of *Building*.

If we need to use this method with type *Building* and all its subtypes, the bounded wildcard can do the magic:

```
public static void paintAllBuildings(List<? extends Building>
buildings) {
    ...
}
```

Now this method will work with type *Building* and all its subtypes. This is called an upper-bounded wildcard, where type *Building* is the upper bound.

We can also specify wildcards with a lower bound, where the unknown type has to be a supertype of the specified type. Lower bounds can be specified using the *super* keyword followed by the specific type. For example, *<? super T>* means unknown type that is a superclass of *T* (= T and all its parents).

5. Type Erasure

Generics were added to Java to ensure type safety. And to ensure that generics won't cause overhead at runtime, the compiler applies a process called *type erasure* on generics at compile time.

Type erasure removes all type parameters and replaces them with their bounds or with *Object* if the type parameter is unbounded. This way, the bytecode after compilation contains only normal classes, interfaces and methods, ensuring that no new types are produced. Proper casting is applied as well to the *Object* type at compile time.

This is an example of type erasure:

```java
public <T> List<T> genericMethod(List<T> list) {
    return list.stream().collect(Collectors.toList());
}
```

With type erasure, the unbounded type *T* is replaced with *Object*:

```java
// for illustration
public List<Object> withErasure(List<Object> list) {
    return list.stream().collect(Collectors.toList());
}

// which in practice results in
public List withErasure(List list) {
    return list.stream().collect(Collectors.toList());
}
```

If the type is bounded, the type will be replaced by the bound at compile time:

```java
public <T extends Building> void genericMethod(T t) {
    ...
}
```

and would change after compilation:

```java
public void genericMethod(Building t) {
    ...
}
```

6. Generics and Primitive Data Types

One restriction of generics in Java is that the type parameter cannot be a primitive type.

For example, the following doesn't compile:

```java
List<int> list = new ArrayList<>();
list.add(17);
```

To understand why primitive data types don't work, let's remember that **generics are a compile-time feature**, meaning the type parameter is erased and all generic types are implemented as type *Object*.

Let's look at the *add* method of a list:

```java
List<Integer> list = new ArrayList<>();
list.add(17);
```

The signature of the *add* method is:

```java
boolean add(E e);
```

and will be compiled to:

```java
boolean add(Object e);
```

Therefore, type parameters must be convertible to *Object*. **Since primitive types don't extend *Object*, we can't use them as type parameters.**

However, Java provides boxed types for primitives, along with autoboxing and unboxing to unwrap them:

```java
Integer a = 17;
int b = a;
```

So, if we want to create a list that can hold integers, we can use this wrapper:

```java
List<Integer> list = new ArrayList<>();
list.add(17);
int first = list.get(0);
```

The compiled code will be the equivalent of the following:

```
List list = new ArrayList<>();
list.add(Integer.valueOf(17));
int first = ((Integer) list.get(0)).intValue();
```

Future versions of Java might allow primitive data types for generics. Project Valhalla (http://openjdk.java.net/projects/valhalla/) aims at improving the way generics are handled. The idea is to implement generics specialization as described in JEP 218 (http://openjdk.java.net/jeps/218).

7. Conclusion

Java Generics is a powerful addition to the Java language because it makes the programmer's job easier and less error-prone. Generics enforce type correctness at compile time and, most importantly, enable implementing generic algorithms without causing any extra overhead to our applications.

The source code that accompanies the article is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-lang-syntax).

Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

ABOUT