Introducing Generics → Type Inference

Introducing Generics

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.

Generic Types

A Simple Box Class

A *generic* type is a generic class or interface that is parameterized over types. The following `Box` class will be modified to demonstrate the concept.

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since its methods accept or return an `Object`, you are free to pass in whatever you want, provided that it is not one of the primitive types. There is no way to verify, at compile time, how the class is used. One part of the code may place an `Integer` in the box and expect to get objects of type `Integer` out of it, while another part of the code may mistakenly pass in a `String`, resulting in a runtime error.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (`<>`), follows the class name. It specifies the type parameters (also called type variables) `T1`, `T2`, ..., and `Tn`.

To update the `Box` class to use generics, you create a generic type declaration by changing the code "`public class Box`" to "`public class Box<T>`". This introduces the type variable, `T`, that can be used anywhere inside the class.

With this change, the `Box` class becomes:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of `Object` are replaced by `T`. A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S, U, V etc. - 2nd, 3rd, 4th types
- You will see these names used throughout the Java SE API and the rest of this section.

Invoking and Instantiating a Generic Type

To reference the generic `Box` class from within your code, you must perform a generic type invocation, which replaces `T` with some concrete value, such as `Integer`:

```
Box<Integer> integerBox;
```

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument — `Integer` in this case — to the `Box` class itself.

> *Type Parameter and Type Argument Terminology: Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the `T` in `Foo<T>` is a type parameter and the `String` in `Foo<String> f` is a type argument. This section observes this definition when using these terms.*

Like any other variable declaration, this code does not actually create a new `Box` object. It simply declares that `integerBox` will hold a reference to a "Box of Integer", which is how `Box<Integer>` is read.

An invocation of a generic type is generally known as a parameterized type.

To instantiate this class, use the `new` keyword, as usual, but place `<Integer>` between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond. For example, you can create an instance of `Box<Integer>` with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

For more information on diamond notation and type inference, see the Type Inference section of this tutorial.

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic `OrderedPair` class, which implements the generic `Pair` interface:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
    }

    public K getKey()    { return key; }
    public V getValue() { return value; }
}
```

The following statements create two instantiations of the `OrderedPair` class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

The code, `new OrderedPair<String, Integer>()`, instantiates `K` as a `String` and `V` as an `Integer`. Therefore, the parameter types of `OrderedPair`'s constructor are `String` and `Integer`, respectively. Due to autoboxing, it is valid to pass a `String` and an `int` to the class.

As mentioned in The Diamond section, because a Java compiler can infer the `K` and `V` types from the declaration `OrderedPair<String, Integer>`, these statements can be shortened using diamond notation:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types

You can also substitute a type parameter (that is, `K` or `V`) with a parameterized type, that is, `List<String>`. For example, using the `OrderedPair<K, V>` example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic `Box` class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of `Box<T>`, you supply an actual type argument for the formal type parameter `T`:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of `Box<T>`:

```
Box rawBox = new Box();
```

Therefore, `Box` is the raw type of the generic type `Box<T>`. However, a non-generic class or interface type is not a raw type.

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;               // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();           // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

The Type Erasure section has more information on how the Java compiler uses raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

```
Note: Example.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

This can happen when using an older API that operates on raw types, as shown in the following example:

```
public class WarningDemo {
    public static void main(String[] args){
        Box<Integer> bi;
        bi = createBox();
    }

    static Box createBox(){
        return new Box();
    }
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with `-Xlint:unchecked`.

Recompiling the previous example with `-Xlint:unchecked` reveals the following additional information:

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
        bi = createBox();
                     ^
1 warning
```

To completely disable unchecked warnings, use the `-Xlint:-unchecked` flag. The `@SuppressWarnings("unchecked")` annotation suppresses unchecked warnings. If you are unfamiliar with the `@SuppressWarnings` syntax, see the section Annotations.

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The `Util` class includes a generic method, compare, which compares two `Pair` objects:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

This feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section,

Type Inference.

Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the `extends` keyword, followed by its upper bound, which in this example is `Number`. Note that, in this context, `extends` is used in a general sense to mean either "`extends`" (as in classes) or "`implements`" (as in interfaces).

```
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a `String`:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

The `isEven()` method invokes the `intValue()` method defined in the Integer class through `n`.

Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

```
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If bound `A` is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Generic Methods and Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array `T[]` that are greater than a specified element `elem`.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
```

```
    return count;
}
```

The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as `short`, `int`, `double`, `long`, `float`, `byte`, and `char`. You cannot use the > operator to compare objects. To fix the problem, use a type parameter bounded by the `Comparable<T>` interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

The resulting code will be:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Generics, Inheritance, and Subtypes

As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an `Integer` to an `Object`, since `Object` is one of `Integer`'s supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

In object-oriented terminology, this is called an "is a" relationship. Since an `Integer` is a kind of Object, the assignment is allowed. But `Integer` is also a kind of `Number`, so the following code is valid as well:

```
public void someMethod(Number n) { /* ... */ }

someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

The same is also true with generics. You can perform a generic type invocation, passing `Number` as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with `Number`:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
```

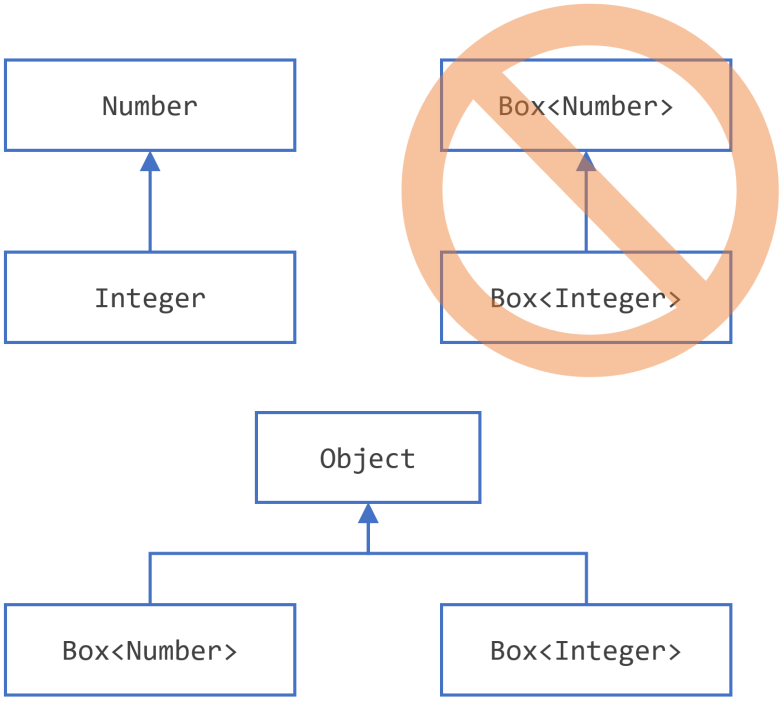
```
box.add(new Double(10.1));  // OK
```

Now consider the following method:

```
public void boxTest(Box<Number> n) { /* ... */ }
```

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect? The answer is "no", because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn. `Box<Integer>` is not a subtype of `Box<Number>` even though `Integer` is a subtype of `Number`.



Subtyping parameterized types.

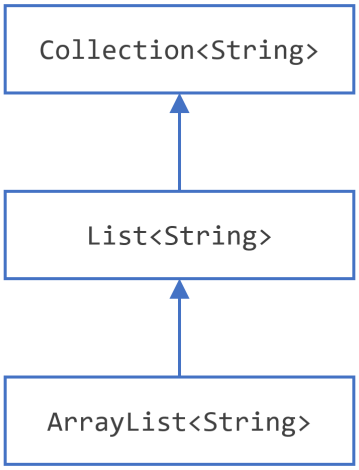
> *Note: Given two concrete types `A` and `B`, for example, `Number` and `Integer`, `MyClass<A>` has no relationship to `MyClass<B>`, regardless of whether or not `A` and `B` are related. The common parent of `MyClass<A>` and `MyClass<B>` is `Object`.*

For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see the section Wildcards and Subtyping.

Generic Classes and Subtyping

You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.

Using the Collections classes as an example, `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`. So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.



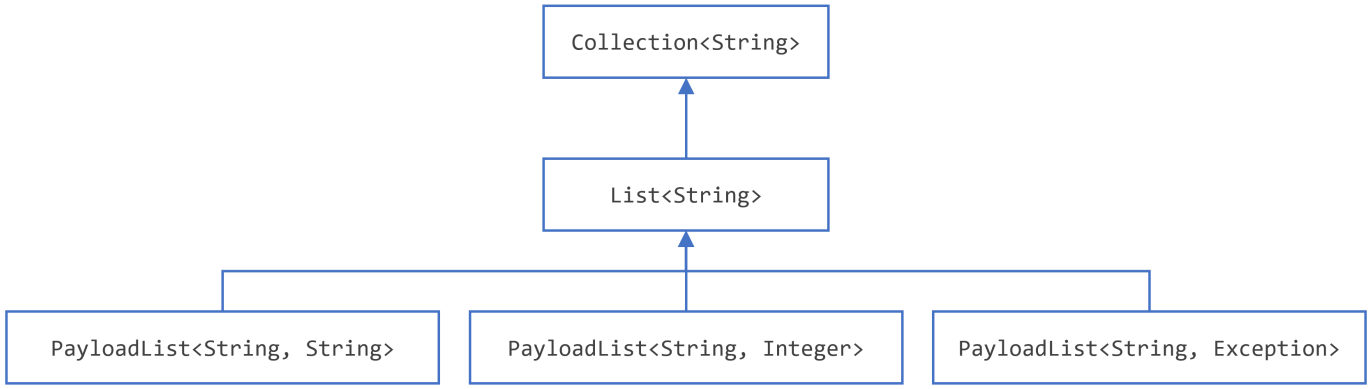
A sample Collection hierarchy.

Now imagine we want to define our own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. Its declaration might look like:

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

The following parameterizations of `PayloadList` are subtypes of `List<String>`:

- `PayloadList<String,String>`
- `PayloadList<String,Integer>`
- `PayloadList<String,Exception>`



A sample Payload hierarchy.

Last update: *September 14, 2021*

Introducing Generics

Type Inference