Type Inference

Type Inference and Generic Methods

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the most specific type that works with all of the arguments.

To illustrate this last point, in the following example, inference determines that the second argument being passed to the pick method is of type `Serializable`:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Generic Methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets. Consider the following example, `BoxDemo`, which requires the `Box` class:

```
public class BoxDemo {

  public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }

  public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
    int counter = 0;
    for (Box<U> box: boxes) {
      U boxContents = box.get();
      System.out.println("Box #" + counter + " contains [" +
             boxContents.toString() + "]");
      counter++;
```

```
    }
  }

  public static void main(String[] args) {
    java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
      new java.util.ArrayList<>();
    BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
    BoxDemo.outputBoxes(listOfIntegerBoxes);
  }
}
```

The following is the output from this example:

```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```

The generic method `addBox()` defines one type parameter named `U`. Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them. For example, to invoke the generic method `addBox()`, you can specify the type parameter with a type witness as follows:

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

Alternatively, if you omit the type witness,a Java compiler automatically infers (from the method's arguments) that the type parameter is `Integer`:

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

Type Inference and Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (`<>`) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (`<>`):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Note that to take advantage of type inference during generic class instantiation, you must use the diamond. In the following example, the compiler generates an unchecked conversion warning because the `HashMap()` constructor refers to the `HashMap` raw type, not the `Map<String, List<String>>` type:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Type Inference and Generic Constructors of Generic and Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes. Consider the following example:

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

Consider the following instantiation of the class `MyClass`:

```
new MyClass<Integer>("")
```

This statement creates an instance of the parameterized type `MyClass<Integer>;` the statement explicitly specifies the type `Integer` for the formal type parameter, `X`, of the generic `class MyClass<X>`. Note that the constructor for this generic class contains a formal type parameter, `T`. The compiler infers the type `String` for the formal type parameter, `T`, of the constructor of this generic class (because the actual parameter of this constructor is a `String` object).

Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods. However, compilers in Java SE 7 and later can infer the actual type parameters of the generic class being instantiated if you use the diamond (`<>`). Consider the following example:

```
MyClass<Integer> myObject = new MyClass<>("");
```

In this example, the compiler infers the type `Integer` for the formal type parameter, `X`, of the generic class `MyClass<X>`. It infers the type `String` for the formal type parameter, `T`, of the constructor of this generic class.

> *Note: It is important to note that the inference algorithm uses only invocation arguments, target types, and possibly an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.*

Target Types

The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears. Consider the method `Collections.emptyList()`, which is declared as follows:

```
static <T> List<T> emptyList();
```

Consider the following assignment statement:

```
List<String> listOne = Collections.emptyList();
```

This statement is expecting an instance of `List<String>` this data type is the target type. Because the method `emptyList()` returns a value of type `List<T>`, the compiler infers that the type argument `T` must be the value `String`. This works in both Java SE 7 and 8. Alternatively, you could use a type witness and specify the value of `T` as follows:

```
List<String> listOne = Collections.<String>emptyList();
```

However, this is not necessary in this context. It was necessary in other contexts, though. Consider the following method:

```
void processStringList(List<String> stringList) {
    // process stringList
}
```

Suppose you want to invoke the method `processStringList()` with an empty list. In Java SE 7, the following statement does not compile:

```
processStringList(Collections.emptyList());
```

The Java SE 7 compiler generates an error message similar to the following:

```
List<Object> cannot be converted to List<String>
```

The compiler requires a value for the type argument `T` so it starts with the value `Object`. Consequently, the invocation of `Collections.emptyList()` returns a value of type `List<Object>`,

which is incompatible with the method `processStringList()`. Thus, in Java SE 7, you must specify the value of the type argument as follows:

```
processStringList(Collections.<String>emptyList());
```

This is no longer necessary in Java SE 8. The notion of what is a target type has been expanded to include method arguments, such as the argument to the method `processStringList()`. In this case, `processStringList()` requires an argument of type `List<String>`. The method `Collections.emptyList()` returns a value of `List<T>`, so using the target type of `List<String>`, the compiler infers that the type argument `T` has a value of `String`. Thus, in Java SE 8, the following statement compiles:

```
processStringList(Collections.emptyList());
```

Target Typing in Lambda Expressions

Suppose you have the following methods:

```
public static void printPersons(List<Person> roster, CheckPerson tester)
```

and

```
public void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester)
```

You then write the following code to call these methods:

```
printPersons(
        people,
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25);
```

and

```
printPersonsWithPredicate(
        people,
        p -> p.getGender() == Person.Sex.MALE
             && p.getAge() >= 18
             && p.getAge() <= 25);)
```

How do you determine the type of the lambda expression in these cases?

When the Java runtime invokes the method `printPersons()`, it is expecting a data type of `CheckPerson`, so the lambda expression is of this type. However, when the Java runtime invokes the method `printPersonsWithPredicate()`, it is expecting a data type of `Predicate<Person>`, so the lambda expression is of this type. The data type that these methods expect is called the target type. To determine the type of a lambda expression, the Java compiler uses the target type of the context or situation in which the lambda expression was found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:

- Variable declarations
- Assignments
- Return statements
- Array initializers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions, `?:`
- Cast expressions

Target Types and Method Arguments

For method arguments, the Java compiler determines the target type with two other language features: overload resolution and type argument inference.

Consider the following two functional interfaces (`java.lang.Runnable` and `java.util.concurrent.Callable<V>`:

```
public interface Runnable {
    void run();
}

public interface Callable<V> {
    V call();
}
```

The method `Runnable.run()` does not return a value, whereas `Callable<V>.call()` does.

Suppose that you have overloaded the method invoke as follows (see the section Defining Methods for more information about overloading methods):

```
void invoke(Runnable r) {
    r.run();
}
```

```
4
5  <T> T invoke(Callable<T> c) {
6      return c.call();
7  }
```

Which method will be invoked in the following statement?

```
1  String s = invoke(() -> "done");
```

The method `invoke(Callable<T>)` will be invoked because that method returns a value; the method `invoke(Runnable)` does not. In this case, the type of the lambda expression `() -> "done"` is `Callable<T>`.

Last update: *September 14, 2021*

Introducing Generics ← **Type Inference** → Wildcards

Home > Tutorials > Generics > Type Inference