Java 8 idioms: Java knows your type

Learn how to use type inference in lambda expressions, and get tips for improving parameter naming

Venkat Subramaniam

October 11, 2017

The Java compiler is more than capable of inferring type, so why not let it? Learn how to use type inference in lambda expressions, and get tips for improving parameter naming.

About this series
Java 8 is the most significant update to the Java language since its inception—packed so full of new features that you might wonder where to start. In this series, author and educator Venkat Subramaniam offers an idiomatic approach to Java 8: short explorations that invite you to rethink the Java conventions you've come to take for granted, while gradually integrating new techniques and syntax into your programs.

Java™ 8 is the first version of Java to support type inference, and it does so only for lambda expressions. Using type inference in lambdas is powerful, and it will set you up for future versions of Java, where type inference will be added for variables and possibly more. The trick is to name your parameters well, and trust the Java compiler to infer the rest.

Most of the time, the compiler is more than capable of inferring type. And when it can't, it will complain.

Learn how type inference works in lambda expressions, and see at least one example where it fails. Even then, there is a workaround.

Explicit types and redundancy

Suppose you ask someone, "What's your name?" and they say "My name is John." This happens often enough, but it would be more efficient to simply say, "John." All you need is a name, so the rest of that sentence is noise.

Unfortunately, we do this sort of thing all the time in code. Here's how a Java developer might use `forEach` to iterate and print double of each value in a range:

```
IntStream.rangeClosed(1, 5)
  .forEach((int number) -> System.out.println(number * 2));
```

The `rangeClosed` method produces a stream of `int` values from 1 to 5. The lambda expression, in its full glory, receives an `int` parameter named `number` and uses the `println` method of `PrintStream` to print out double of that value. Syntactically there's nothing wrong with the lambda expression, but the type detail is redundant.

Type inference in Java 8

When you extract a value from a range of numbers, the compiler knows that value's type is `int`. There is no need to state the value explicitly in your code, although that has been the convention until now.

In Java 8, we can drop the type from a lambda expression, as shown here:

```
IntStream.rangeClosed(1, 5)
  .forEach((number) -> System.out.println(number * 2));
```

Being statically typed, Java needs to know the types of all objects and variables at compile time. Omitting the type in the parameter list of a lambda expression does not bring Java closer to being a dynamically typed language. Adding a healthy dose of type inference does, however, bring Java closer to other statically typed languages, like Scala or Haskell.

Trust the compiler

If you omit the type in a parameter to a lambda expression, Java will require contextual details to infer that type.

Going back to the previous example, when we call `forEach` on `IntStream`, the compiler looks up that method to determine the parameter(s) it takes. The `forEach` method of `IntStream` expects the functional interface `IntConsumer`, whose abstract method `accept` takes a parameter of type `int` and returns `void`.

If you specify the type in the parameter list, the compiler will confirm that the type is as expected.

If you omit the type, then the compiler will infer the expected type—`int` in this case.

Whether you provide the type or the compiler infers it, Java knows the type of its lambda expression parameters at compile time. You can test this out by making an error within a lambda expression, while omitting the type for the parameter:

```
IntStream.rangeClosed(1, 5)
  .forEach((number) -> System.out.println(number.length() * 2));
```

When you compile this code, the Java compiler will return the following error:

```
Sample.java:7: error: int cannot be dereferenced
      .forEach((number) -> System.out.println(number.length() * 2));
                                                    ^
1 error
```

The compiler knows the type of the parameter called `number`. It complained because it isn't possible to de-reference a variable of type `int` using the dot operator. You can do that for objects, but not for `int` variables.

Benefits of type inference

There are two major benefits to omitting type in lambda expressions:

- Less typing. There is no need to key in the type information given the compiler can easily determine that for itself.
- Less code noise—`(number)` is much simpler than `(int number)`.

Furthermore, as a rule, if we have only one parameter, omitting type means we can also leave out the `()`, as shown:

```
IntStream.rangeClosed(1, 5)
  .forEach(number -> System.out.println(number * 2));
```

Note that you *will* need to add the parenthesis for lambda expressions taking more than one parameter.

Type inference and readability

Type inference in lambda expressions is a departure from the normal practice in Java, where we specify the type of every variable and parameter. While some developers argue that Java's convention of specifying type makes it more readable and easier to understand, I believe that preference may reflect habit more than necessity.

Take an example of a function pipeline with a series of transformations:

```
List<String> result =
  cars.stream()
    .map((Car c) -> c.getRegistration())
    .map((String s) -> DMVRecords.getOwner(s))
    .map((Person o) -> o.getName())
    .map((String s) -> s.toUpperCase())
    .collect(toList());
```

Here we start with a collection of `Car` instances and associated registration information. We obtain the owner of each car and the owner's name, which we convert to uppercase. Finally, we put the results into a list.

Every lambda expression in this code has a type specified for its parameter, but we've used single-letter variable names for the parameters. This is very common in Java. It is also unfortunate, because it leaves out domain-specific context.

We can do better than this. Let's see what happens when we rewrite the code with stronger parameter names:

```
List<String> result =
  cars.stream()
    .map((Car car) -> car.getRegistration())
    .map((String registration) -> DMVRecords.getOwner(registration))
    .map((Person owner) -> owner.getName())
    .map((String name) -> name.toUpperCase())
    .collect(toList());
```

These parameter names carry domain-specific information. Rather than using `s` to represent a `String`, we've specified domain-specific details like `registration` and `name`. Likewise, instead of `p` or `o`, we use `owner` to show that `Person` is not just a person but is the owner of the car.

Each lambda expression in this example is a notch better than the one it replaces. When we read the lambda—for example, `(Person owner) -> owner.getName()`—we know that we're getting the name of the owner and not just some arbitrary person.

Naming parameters

Some languages like Scala and TypeScript place more importance on parameter names than their types. In Scala, we define parameters before type, for instance by writing:

```
def getOwner(registration: String)
```

instead of:

```
def getOwner(String registration)
```

Both type and parameter names are useful, but in Scala parameter names are more important. We can also take this idea to heart when writing lambda expressions in Java. Note what happens when we drop the type details and parenthesis from our car registration example in Java:

```
List<String> result =
  cars.stream()
    .map(car -> car.getRegistration())
    .map(registration -> DMVRecords.getOwner(registration))
    .map(owner -> owner.getName())
    .map(name -> name.toUpperCase())
    .collect(toList());
```

Because we added descriptive parameter names, we did not lose much context, and the explicit type, now redundant, has quietly disappeared. The result is cleaner, quieter code.

Limits of type inference

While using type inference has benefits for efficiency and readability, it is not a technique for all occasions. In some cases, it is simply not possible to use type inference. Fortunately, you can count on the Java compiler to let you know when that happens.

We'll first look at an example where the compiler is tested but succeeds, then one where it fails. What's most important is that in both cases, you can count on the compiler to work as it is supposed to.

Expanding type inference

For our first example, suppose we want to create a `Comparator` to compare `Car` instances. The first thing we need is a `Car` class:

```
class Car {
  public String getRegistration() { return null; }
}
```

Next, we create a `Comparator` that compares `Car` instances based on their registration information:

```
public static Comparator<Car> createComparator() {
  return comparing((Car car) -> car.getRegistration());
}
```

The lambda expression used as argument to the `comparing` method carries type information in its parameter list. We know the Java compiler is pretty shrewd about type inference, so let's see what happens if we omit the type of the parameter, like so:

```
public static Comparator<Car> createComparator() {
  return comparing(car -> car.getRegistration());
}
```

The `comparing` method takes one argument. It expects `Function<? super T, ? extends U>` and returns `Comparator<T>`. Since `comparing` is a static method on `Comparator<T>`, the compiler so far has no clue to what `T` or `U` might be.

To resolve this, it expands its inference a little further, beyond the argument passed to the `comparing` method. It looks for what we're doing with the result of the call to `comparing`. From this, the compiler determines that we merely return the result. Next, it sees that the `Comparator<T`, which is returned by `comparing`, is further returned as `Comparator<Car>` by `createComparator`.

Ta daaa! Now the compiler is catching on to us: it infers that `T` should be bound to `Car`. From this, it knows that the type of the parameter `car` in the lambda expression should be `Car`.

The compiler had to do some extra work to infer the type in this case, but it succeeded. Next let's see what happens when we level up the challenge, and reach the limits of what the compiler can do.

Limits of inference

To start, we'll add a new call following the previous one to `comparing`. In this case, we also reintroduce the explicit type for the lambda expression's parameter:

```
public static Comparator<Car> createComparator() {
  return comparing((Car car) -> car.getRegistration()).reversed();
}
```

This code has no problem compiling with the explicit type, but now let's leave out the type information and see what happens:

```
public static Comparator<Car> createComparator() {
  return comparing(car -> car.getRegistration()).reversed();
}
```

As you can see below, it doesn't go well. The Java compiler complains with errors:

```
Sample.java:21: error: cannot find symbol
    return comparing(car -> car.getRegistration()).reversed();
                               ^
  symbol:   method getRegistration()
  location: variable car of type Object
Sample.java:21: error: incompatible types: Comparator<Object> cannot be converted to Comparator<Car>
    return comparing(car -> car.getRegistration()).reversed();
                                                           ^
2 errors
```

Just like the previous scenario, before we included `.reversed()`, the compiler asks what we're doing with the result of the call to `comparing(car -> car.getRegistration())`. In the previous case, we returned the result as `Comparable<Car>`, and so the compiler was able to infer that `T`'s type would be `Car`.

In this modified version, however, we've passed the result of `comparable` as a target to call `reversed()`. The `comparable` returns `Comparable<T>`, and `reversed()` doesn't reveal anything more about what `T` might be. From this, the compiler infers that `T`'s type must be `Object`. Sadly, that's not sufficient for this code, because `Object` lacks the `getRegistration()` method that we're calling within our lambda expression.

Type inference fails at this point. In this case, the compiler actually needed information. Type inference looks at the arguments and the return or assignment elements to determine type, but if the context offers insufficient details, the compiler reaches its limits.

Method references to the rescue?

Before we give up on this particular situation, let's try one more thing: instead of a lambda expression, let's try using a method reference:

```
public static Comparator<Car> createComparator() {
  return comparing(Car::getRegistration).reversed();
}
```

The compiler is quite happy with this solution. It uses the `Car::` in the method reference to infer the type.

Conclusion

Java 8 introduced limited type inference for parameters to lambda expressions, and type inference will be expanded to local variables in a future version of Java. Learning to omit type details and trust the compiler now will help set you up for what's coming ahead.

Relying on type inference and well-named parameters will help you write code that is concise, more expressive, and less noisy. Use type inference whenever you believe the compiler will be able to infer type on its own. Provide type details only in situations where you are certain the compiler truly needs your help.

Related topics

- Java programming with lambda expressions
- Java 8 language changes
- Functional Programming in Java: The Pragmatic Bookshelf, 2014