CODING

Modern Java toys that boost productivity, from type inference to text blocks

Developers using older versions of the Java platform are missing out.

by Angie Jones

October 23, 2020

Download a PDF of this article

Although Java is one of the industry's most widely used programming languages, it has gotten an undeserved bad reputation over the years as being verbose and stagnant. Yes, sometimes you have to write a lot of code to do the most basic things. And yes, the releases of Java 7, 8, and 9 were each three years apart—and that's an eternity in software development.

Fortunately, the powers that be have heard us loud and clear: Java has received a much-needed makeover, and now new versions of the language are being released every 6 months, with the most recent version being Java 15, which was released in September 2020.

With so many features, it may be hard to keep up, especially when it comes to identifying the parts of the platform that can make applications faster and easier to write. In this article, I'll demonstrate several of the newer features of Java that I find most useful.

Local variable type inference

The following example follows Java conventions and uses good names for both the class and the object:

```
AccountsOverviewPage accountsOverviewPage =
page.login(username, password);
```

However, many times, as you see here, these two names are the same, which is redundant and makes for a lot of typing.

In version 10, Java introduced local variable type inference. What this means is that instead of explicitly declaring an object or a variable's type, you can instead use the keyword `var`, and Java will infer what the type is based on what is being assigned to it, for example:

```
var accountsOverviewPage = page.login(username,
password);
```

This feature saves developers some keystrokes and addresses some of the verbosity of the language.

Java is still a statically typed language. The use of type inference does not make Java a dynamically typed language such as JavaScript or Python. The type is still there; it's just inferred from the right-hand side of the statement, which means you can use `var` only if you're actually initializing the variable. Otherwise, Java will not be able to infer what the type is, as in the following example:

```
var accountsOverviewPage; //gives compilation
error
```

Type inference cannot infer type on global variables. As the name of the feature implies, it works only for local variables. You can use `var` inside of methods, loops, and decision structures; however, you cannot use `var` for global variables, even if you are initializing them. The following code produces an error:

```
public class MyClass {
    var accountsOverviewPage = page. login(us
}
```

Type inference is not allowed in headers. While local variable type inference can be used within the body of local constructs, it cannot be used in the headers of methods or constructors, as shown in the following example. This is because the caller needs to know the data type of the arguments to send.

```
public class MyTests {

    public MyTests(var data) {} //gives compi
}
```

Type inference means that naming is even more important now. Given the following variable name and the following method name, I have no idea what the inferred data type of `x` would be:

```
var x = getX();
```

Java will know because it can infer the type based on what's returned from `getX()`. However, as someone reading the code, I can't easily tell what it is. That makes it difficult to work with this variable.

You should always use good variable names, but it's even more important if you're going to use `var` because some of the context is removed.

Not everything needs to be a var. Once you start using type inference, you're going to love it! However, please don't overdo it.

For example, using `var` as shown below doesn't do you any favors, because it removes context for no good reason:

```
var numberOfAccounts = 5;
```

Be careful with cases that lead to potential ambiguity. In the following declaration, what would you guess the inferred type would be?

```
var expectedAccountIdsList = new ArrayList();
```

If you guessed an `ArrayList` of `Objects`, you're correct!

While this may be OK in many cases, if you want to use the dot operator (`.`) on any of the elements in this collection, you'll be limited to the methods available in the `Object` class.

For more specific inference, use as much information as you can on the right-hand side of the assignment. For example, using the diamond operator to specify the type as `String` ensures that `expectedAccountIdsList` is defined as an `ArrayList` of `Strings`.

```
var expectedAccountIdsList = new
ArrayList<String>();
```

New operations can improve stream efficiency

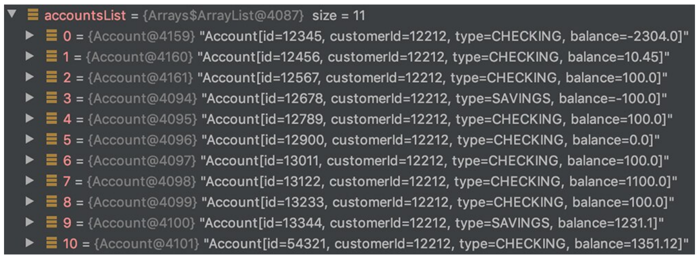
Java 9 introduced two new operations in the Stream API: takeWhile and dropWhile.

The `takeWhile()` operation processes the items of a collection and keeps each one while a given condition (known as a *predicate*) is true. The `dropWhile()` operator does the opposite: It disregards the items of a collection while the predicate is true.

In the example below, I get a list of accounts, and then I use `takeWhile()` to keep all the accounts that have a type of `CHECKING` but only until the code gets to an account that does not have this type:

```
var accountsList = APIUtil.getAccounts(custom
var checkingAccountsList = accountsList
        .stream()
        .takeWhile(account -> account.type().
        .collect(Collectors.toList());
```

Given the list of accounts shown in **Figure 1**, calling `takeWhile()` with a predicate of type `equals CHECKING` would lead to the first three entries being kept. Although there are additional entries here that match the predicate, the stream ends when the predicate is not met. Since the fourth element is of type `SAVINGS`, the stream is stopped when this element is reached.



```
▼ accountsList = {Arrays$ArrayList@4087}  size = 11
  ▶ 0 = {Account@4159} "Account[id=12345, customerId=12212, type=CHECKING, balance=-2304.0]"
  ▶ 1 = {Account@4160} "Account[id=12456, customerId=12212, type=CHECKING, balance=10.45]"
  ▶ 2 = {Account@4161} "Account[id=12567, customerId=12212, type=CHECKING, balance=100.0]"
  ▶ 3 = {Account@4094} "Account[id=12678, customerId=12212, type=SAVINGS, balance=-100.0]"
  ▶ 4 = {Account@4095} "Account[id=12789, customerId=12212, type=CHECKING, balance=100.0]"
  ▶ 5 = {Account@4096} "Account[id=12900, customerId=12212, type=CHECKING, balance=0.0]"
  ▶ 6 = {Account@4097} "Account[id=13011, customerId=12212, type=CHECKING, balance=100.0]"
  ▶ 7 = {Account@4098} "Account[id=13122, customerId=12212, type=CHECKING, balance=1100.0]"
  ▶ 8 = {Account@4099} "Account[id=13233, customerId=12212, type=CHECKING, balance=100.0]"
  ▶ 9 = {Account@4100} "Account[id=13344, customerId=12212, type=SAVINGS, balance=1231.1]"
  ▶ 10 = {Account@4101} "Account[id=54321, customerId=12212, type=CHECKING, balance=1351.12]"
```

Figure 1. A list of bank accounts

Similarly (yet the opposite situation), if you invoked `dropWhile()` on this stream, the elements at index 3–10 will be kept: `dropWhile()` drops the first three entries because they matched the predicate, and once it reaches type `SAVINGS` on the fourth element, the stream ends.

```
var accountsList = APIUtil.getAccounts(custom
var checkingAccountsList = accountsList
        .stream()
        .dropWhile(account -> account.type().
        .collect(Collectors.toList());
```

Sort collections for deterministic results. If you're interested in collecting or dropping *all* the elements that match the predicate, be sure to sort the stream before calling `takeWhile()` or `dropWhile()`, for example:

```
var accountsList = APIUtil.getAccounts(custom
var checkingAccountsList = accountsList
        .stream()
        .sorted(Comparator.comparing(Account:
        .takeWhile(account -> account.type().
        .collect(Collectors.toList());
```

Sorting the collection, as seen on line 4 above, guarantees that all elements that match the predicate are accepted or dropped as expected.

The difference between takeWhile and filter. A common question is "What's the difference between `takeWhile` and `filter`?" Both use a predicate to narrow a stream.

The difference is that the `filter()` operation looks through the entire collection and gathers all elements that match the predicate, whereas `takeWhile()` short-circuits this process by stopping the operation once it encounters an element that does not match the predicate, which makes `takeWhile()` faster.

Using takeWhile or dropWhile on parallel streams. Performance suffers if `takeWhile()` or `dropWhile()` are used on parallel streams, even when the streams are ordered.

It's recommended that you use these operations on standalone streams for optimal performance.

Switch expressions

Java 12 introduced switch expressions, which enable you to use `switch` to directly assign a value to a variable. In the following example, notice I am using `switch` on the right side of a statement to initialize the variable `id`.

```
String id = switch(name) {
        case "john" -> "12212";
        case "mary" -> "4847474";
        case "tom" -> "293743";
        default -> "";
};
```

This code is saying if the name is `john`, then assign `12212` to the variable `id`.

The `case` statements don't need a colon in `switch` expressions, but instead they use an arrow.

Fall-through in switch expressions. You don't need a `break` statement in `switch` expressions because there is no fall-through with `switch` expressions. This is one of the benefits of using `switch` expressions, because a common error is to forget a `break` statement in `switch` statements, which results in unexpected behavior. This error can be avoided with `switch` expressions.

However, there are times where you may want to address multiple cases with a single block. You can do so in `switch` expressions by specifying each case in a comma-delimited list, as shown below:

```
return switch(name) {
        case "john", "demo" -> "12212";
        case "mary" -> "4847474";
        case "tom" -> "293743";
        default -> "";
    };
```

Notice that in the first case, if the name is `john` or `demo`, then `12212` will be returned.

Executing additional logic in switch expressions. While the primary purpose of `switch` expressions is to assign a value, additional logic may be required to determine that value.

To accomplish this, you may implement a block of code within the `case` statements of `switch` expressions by enclosing the statements inside a set of curly braces.

However, the final statement of the `switch` expression must be the `yield` method, which provides a value for the assignment, as seen in the `case` statement for `john` below:

```
return switch(name) {
    case "john" -> {
        System.out.println("Hi John");
        yield "12212";
    }
    case "mary" -> "4847474";
    case "tom" -> "293743";
    default -> "";
};
```

Throwing exceptions from switch expressions. You can use any of the `case` statements to throw an exception.

```
return switch(name){
    case "john" -> "12212";
    case "mary" -> "4847474";
    case "tom" -> "293743";
    default -> throw new InvalidNameException
};
```

Of course, in the default case, no value is being returned because the entire flow is interrupted by the exception.

Throwing exceptions is not limited to the default case. An exception can be thrown from any of the `case` statements, as shown below:

```
return switch(name){
    case "john" -> "12212";
    case "mary" -> throw new AccountClosedExc
    case "tom" -> "293743";
    default -> throw new InvalidNameException
};
```

When to use switch expressions. Switch *expressions* are not a replacement for switch *statements*; they are an addition to the language. You certainly can still use `switch` statements, and in some cases, that may be the more favorable option.

As a rule of thumb, use `switch` expressions when you are using this construct to assign a value; and use `switch` statements when you're not assigning a value, but you just need to conditionally invoke statements.

Records

Records are a new type of class introduced in Java 14 as a preview feature. Records are great for simple classes that only need to contain fields and access to those fields. Here is a record that can serve as a model for an `Account`:

```
public record Account(
       int id,
       int customerId,
       String type,
       double balance) {}
```

Notice that instead of the word *class*, I used *record*. Also, the fields are defined in the class declaration within a set of parentheses, followed by a set of curly braces.

That's it! This simple declaration creates a record with these fields. You don't need to create any getters or setters. You don't need to override the inherited methods of `equals()`, `hashCode()`, or `toString()`. All that is done for you.

However, if you want to override anything or add additional methods, you can do so within the curly braces, for example:

```
public record Account(
       int id,
       int customerId,
       String type,
       double balance
) {
    @Override
    public String toString(){
        return "I've overridden this!";
    }
}
```

Instantiating records. Records can be instantiated just like classes. In the following example, `Account` is the name of my record, and I use the `new` keyword and call the constructor passing in all of the values:

```
Account account = new Account(13344, 12212,
"CHECKING", 4033.93);
```

Records are immutable. The fields of a record are final, so there are no setter methods generated for records. Of course, you can add a setter within the curly braces of the record, but there’s no good reason to do that since the fields are final and cannot be modified.

```
Account account = new Account(13344, 12212, "
account.setType("SAVINGS"); //gives compilati
```

For the same reason, you cannot directly use records as `builder` classes. Attempting to modify the final fields of a record results in a compilation error, for example:

```
public record Account(
        int id,
        int customerId,
        String type,
        double balance)
{
    //gives compilation error
    public Account withId(int id){
        this.id = id;
    }
}
```

Accessor methods. With records, you do have accessor methods; however, they do not start with the word *get*. Instead, the accessor method name is the same as the field name. Notice below that `account.balance()` is called rather than `account.getBalance()`.

```
Account account = new Account(13344, 12212, "
double balance = account.balance();
```

Inheritance is not supported. Since records are final, they cannot inherit from other classes or records. Attempting to use the `extends` clause in a record declaration will result in a compilation error, as shown below:

```
public record CheckingAccount() extends
Accounts {} //gives compilation error
```

Records can implement interfaces. Records can, however, implement interfaces. Just like classes, records use the `implements` keyword in their declaration to specify their intent, and the methods can be implemented within the records’ curly braces, for example:

```
public interface AccountInterface {

    void someMethod();
}
```

```
public record Account(
        int id,
        int customerId,
        String type,
        double balance) implements AccountsIn
{
    public void someMethod(){

    }
}
```

Text blocks

Representing big blocks of complex text within a Java string can be very tedious. In the following example, notice how all of the quotation marks need to be escaped, new line characters are needed for each line break, and plus signs are needed to join each line:

```
String response =
"[\n" +
"  {\n" +
    "    \"id\": 13344,\n" +
    "    \"customerId\": 12212,\n" +
    "    \"type\": \"CHECKING\",\n" +
    "    \"balance\": 4022.93\n" +
    "  },\n" +
    "  {\n" +
    "    \"id\": 13455,\n" +
    "    \"customerId\": 12212,\n" +
    "    \"type\": \"CHECKING\",\n" +
    "    \"balance\": 1000\n" +
    "  }\n" +
    "]";
```

Text blocks, introduced in Java 13, allow you to use three quotation marks to open and close a big block of text, for example:

```
return """
        [
          {
            "id": 13344,
            "customerId": 12212,
            "type": "CHECKING",
            "balance": 3821.93
          },
          {
            "id": 13455,
            "customerId": 12212,
            "type": "LOAN",
            "balance": 989
          }
        ]
       """;
```

Notice that you don’t need to escape anything. The individual quotation marks are still there on the fields and the line breaks are respected.

Text cannot begin on the same line as the opening quote. You cannot include the entire text block on the same line. If you do, you’ll get a compilation error, as shown below:

```
return """ Hey y'all! """; //gives compilation
error
```

A new line must be after the opening quotes as shown in the next example. This is legal but it is not the preferred way.

```
return """
             Hey y'all!""";
```

The preferred way is to have both the opening and the closing quotes aligned on their own respective lines with the text block in between, for example:

```
return """
             Hey y'all!
            """;
```

Conclusion

These are just a few of my favorite new features from recent versions of Java. As you can see, the language has certainly improved in the areas of verbosity and adopting modern programming trends. Cheers to beloved Java!

Dig deeper

- var and Java 10’s expanded type inference
- Java 9 core library updates: Collections and streams
- Inside Java 13’s switch expressions and reimplemented Socket API
- Records come to Java
- Text blocks come to Java

Angie Jones

Angie Jones is a Java Champion who specializes in test automation strategies and techniques. She shares her wealth of knowledge by speaking and teaching at software conferences all over the world, and she leads the online learning platform Test Automation University. As a master inventor, Jones is known for her innovative and out-of-the-box thinking style, which has resulted in more than 25 patented inventions in the US and China. In her spare time, Jones volunteers with Black Girls Code to teach coding workshops to young girls in an effort to attract more women and minorities to tech.

Share this Page

Contact
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us
Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials
Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events
Acquisitions
Blogs
Events
Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices