Ripe for the picking: 11 essential Java features to help modernize your code

Use these Java language improvements to make your code easier to write, read, and maintain.

by Ian Darwin

Since many developers learned Java years ago, Java Magazine allowed me a few minutes to show some of the features, introduced over the past several years, that can help you modernize and improve your code. Think of these Java improvements as being tasty fruit that’s within your reach: If you aren’t using these 11 features in your software, you should be. At the very least, you should taste-test them.

Let me emphasize that these aren’t all new features. In fact, in this article I’m focusing on some older features that you and your developer team may not be using.

The main takeaway is that far-reaching changes have always been a fact of life for Java developers (as in many programming languages), and the exercise of keeping abreast of these changes can help keep your code supple. If you aren’t using these 11 language improvements, perhaps it’s time to start.

Date/Time package (introduced with Java 8)

One of my favorite goodies, included with Java 8, is the java.time package, originally known as Joda-Time. (Prior to Java 8, developers could explicitly use the Joda-Time library.)

Many early adopters struggled with JDK 1.0’s Date class, with its peculiar years-from-1970 bias and limited computation capabilities. The Calendar (added in Java 1.1) helped a bit but not enough. The Java 8
package's JSR 310 improvements added a lot of date and time classes and can seem overwhelming.

The main classes most developers work with are LocalDate, LocalDateTime, ZonedDateTime, and ZonedDateTime. As the names imply, two of these are for day-only representation, and two are for date and time display; two have no time zone associated and two do. They all have sensible defaults and simple constructor methods, and they all have an awesome degree of calculation methods built in, such as plusDays() and between().

For example, to use today's date, simply reference LocalDate.now(). To use a fixed date, such as Christmas Day 2025, use LocalDate.of(2025, 12, 25). The order of fields is unambiguous and follows the ISO specification for dates: year, month, and day.

```java
LocalDate d = LocalDate.now();
System.out.println("Today is " + d);
Today is 2027-03-12
```

You can print dates and date-times in any format you like using a DateTimeFormatter (from the java.time.format subpackage).

```java
DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy/MM/dd");
System.out.println(df.format(LocalDate.now()));
2027/03/12
```

Because not all code could be converted to the new Date/Time API overnight, you can convert between the old API and the new using methods built into both. Date offers toInstance(), which returns an Instant (a point in time) without any time zone (true to the original Date class). To get a LocalDateTime instance, pass the Instant object along with a time zone into which to convert it, commonly the system default, as follows:

```java
Date oldDate = new Date();
System.out.println(oldDate.toInstant());
LocalDateTime newDate = LocalDateTime.ofInstant(oldDate.toInstant(), ZoneId.systemDefault());
System.out.println(newDate);
2027-05-12T19:23:19.695Z
2027-05-12T15:23:19.695
```

Doing that prints the following:

```bash
2027-05-12T19:23:19.695Z
2027-05-12T15:23:19.695
```

More interesting is the calculation functionality built into the new API. For example, in running a small business, you need to know when to pay people. Assume there's a mix of monthly salaried employees and hourly employees paid weekly. When will workers of each type get paid? The TemporalAdjusters (in java.time.temporal) will help.
Aided by the great range of methods to create a modified date and
time, all objects in the new API are immutable. Immutability is useful in
writing thread-safe code. There's a trend towards immutable objects, as
will be seen in the later section on records.

There's more to this API. These examples (with imports) are in my
javasrc on GitHub in the folder main/src/main/java/datetime.

**Lambdas (introduced with Java 8)**

Lambdas are anonymous functions and gave Java the functional
programming tenet of "code as data." In one sense, that capability had
always been there: You could define a variable of a given object type
and pass that into a method. With lambdas, however, the process is
much easier and clearer.

Lambdas are not new. The term was first used by Alonzo Church (of
Church and Turing fame) in mathematics in the 1930s. To see how
lambdas are implemented by Java, check Ben Evans' article "Behind
the scenes: How do lambda expressions really work?"

How do you use lambdas? Consider the once-common case of adding
an `ActionListener` to a `JButton`.

```java
JButton qb = new JButton("Quit");
class QuitListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
};
ActionListener al = new QuitListener();
qb.addActionListener(al);
```

The code above creates a class that is used only a single time. You can
shorten the code using an anonymous inner class, as follows:

```java
JButton qb = new JButton("Quit");
 ActionListener al = new ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
};
```
This code lets the compiler pick a name for the class, but it still creates a single instance that is used only once. You could shorten the code a bit more by not getting a reference to the anonymous inner class.

```java
qb.addActionListener(new ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
```

But the code is still repeating a lot of information that is unambiguously inferable by the compiler. Lambdas take the fullest advantage of compiler inference.

```java
qb.addActionListener(e -> System.exit(0));
```

Boom! So much redundant typing saved, so much less code to review, and so much less code to read and reread when maintaining the code.

The syntax of lambdas is `param-list -> statement`.

The `param-list` can be a single parameter by itself or a possibly empty list of parameters in parentheses, as with any other method call. A lambda has to implement a functional interface, that is, a Java interface with one and only one abstract method. `ActionListener` is one such method, but the related `WindowListener` method is not; `WindowListener` has seven methods to implement. Extensive compiler inference is used to infer all the other information needed.

To save you from needing to create your own functional interfaces, you can find a series of predefined ones in Java 8's `java.util.function` subpackage. The simplest is `Consumer`, which has one abstract method, `accept(T)`. The `T` is a type parameter, introduced in Java 5, and hopefully it’s familiar to you.

```
$ javap java.util.function.Consumer
Compiled from "Consumer.java"
public interface java.util.function.Consumer<T> {
    public abstract void accept(T);
    public default java.util.function.Consumer<T> andThen(java.util.function.Consumer<T> another)
}
$
```

There’s an example of using `Consumer` in the next section.

**Default methods and List.forEach (introduced with Java 8)**

Java 8 added the notion of default methods to interfaces with, as the name implies, default method bodies defined in the interface. This allowed for backwards-compatible changes. My favorite example of that is `List.forEach()`. It's actually `Iterable.forEach()`, but that's
the super-super-interface to `List`, and `List` is where people tend to use it. The `Iterable` interface now looks like the following:

```java
public interface java.lang.Iterable<T> {
    public abstract java.util.Iterator<T> iterator();
    public default void forEach(java.util.function.Consumer<? super T> action);
    public default java.util.Spliterator<T> spliterator();
}
```

What about that `Consumer`? It's from `java.util.function`, discussed above in the lambdas section.

While old Java code didn't have the `forEach` method, that method's addition does not interfere with existing code; if you have your own `List` implementation, for example, you don't have to do anything. The default method will simply work.

```java
List<String> names = List.of("Robin", "Toni", "JJ");
names.forEach(System.out::println);
```

The last piece in the above code, `System.out::println`, is a method expression for the `println` method. The `forEach()` method requires a `Consumer`, which is a functional interface whose abstract `accept()` method (a method accepting one parameter and returning `void`) can be applied to `System.out.println(String s)`, so you can use it here.

The `::` syntax generates a method reference, also new in Java 8, that allows creating a lambda reference to an existing function. How? By using a static (class) or instance definition, the `::` operator, and the method to be referenced.

### Streams (introduced with Java 8)

A stream is an object (defined in `java.util.stream`) that provides a flow of data objects. Streams can be finite or infinite, single-threaded or parallel. Streams can come from a `List` or other collection or from a file, be generated dynamically, and so on. A typical streams process contains one stream generator, zero or more stream operations, and one terminal operation.

Consider the problem of counting the number of unique lines in a file. Without streams, you might read the lines into an array, sort the array, and walk through it comparing each line to the one before it. Other approaches are to use a hashing table or tree. Streams are simpler, and the code is more readable. Below, the `Files.lines()` method (in `java.nio.file` since Java 7) provides a finite stream of all the lines of text from a file. The code sorts the stream, removes duplicates, and counts the number of distinct lines.

```java
long numberLines = Files.lines(Path.of("lines.txt"))
    .sorted()
    .distinct()
    .count();
System.out.printf("lines.txt contains " + numberLines + " lines.");
```
Here’s another example. A useful method, `Random.doubles()`, returns an infinite stream of random numbers generated on the fly. The infinite stream can be stopped by calling the stream method `limit(int num);` all the elements in the stream up to `num` will be passed on, in this case, to the `average()` function. For a large value of `num`, the result should be about 0.5—for example, a large number of coin tosses of an unmodified coin should be 0.5 heads and 0.5 tails.

The `average()` function returns an `OptionalDouble`. Call the `getAsDouble()` method (in `OptionalDouble`) to retrieve the value (which can’t be the value `empty`, given what you know about the input). For more about `Optional`, see “12 recipes for using the Optional class as it’s meant to be used” by Mohamed Taman.

```
System.out.println(new Random().doubles().limit(10).average());
0.5384002737224926
System.out.println(new Random().doubles().limit(10).average());
0.633140919801152
System.out.println(new Random().doubles().limit(10).average());
0.514099594406035
System.out.println(new Random().doubles().limit(10).average());
0.48159324772449064
System.out.println(new Random().doubles().limit(10).average());
0.5059246586725906
```

As you can see, with larger values of `num`, the average does approach .5.

**Modules (introduced with Java 9)**

Long ago, the maintainers of the JDK realized that its source code had grown too large to manage. A plan was set in motion to modularize the JDK, both to make maintenance easier and to provide better isolation between parts of the JDK and between the JDK and applications—a process which continues even today, as Ben Evans explains in “A peek into Java 17: Continuing the drive to encapsulate the Java runtime internals.”

This initiative, originally called the Java Platform Module System, was successful at simplifying and segregating the code of the JDK. It was decided to make this same mechanism part of normal Java application development. While this caused quite a bit of churn as third-party tools and libraries moved more slowly than others to support modularity, it is now pretty solidly established.

Best of all, the module system is key to helping you build and maintain your own large applications.

The module system is important to developers. It provides, for example, a clear declaration of which parts of your code are public APIs, which are implementation, and which are (possibly multiple) implementations of a public interface. You should be migrating to modular code if you haven’t already done so.

If you are not using modules yet, refer to *Java 9 Modularity* by Sander Mak and Paul Bakker.
JShell (introduced with Java 9)

One of the hurdles many people have while learning Java is having to type the public class `Foo` (which must match the filename the class is in) followed by

```java
public static void main(String[] args) {...}
```

simply to be able to add `2+2`.

JShell is the answer to this excessive complexity. JShell is an interactive shell, also known as a REPL (read-evaluate-print loop), where Java's syntax is a bit relaxed and you don't have to put `class` and `main` around statements.

The argument `PRINTING` tells JShell to allow `println()` as a shortcut for `System.out.println`. Values that are not stored in variables are output immediately; that's the `P` part of `REPL`.

```bash
$ jshell PRINTING
|  Welcome to JShell -- Version 16
|  For an introduction type: /help intro
jshell> println(2+2)
4
jshell> 2+2
$23 ==> 4
jshell> import java.time.*;
jshell> var d = LocalDate.of(2027,12,5);
d ==> 2027-12-05
jshell> /exit
|  Goodbye
$
```

This tool is not just for beginners, of course: I use JShell a lot when exploring APIs. As always, there is more to the topic. Try `jshell --help` and `/intro` in JShell for more information. The Java SE documentation has a good introduction to JShell.

Running a single file (introduced with Java 11)

Java 11 allows you to run a self-contained Java program without first compiling it. For example

```bash
$ java HelloWorld.java
Hello, world
$
```

You must add the extension `.java` to the filename, lest the command think you're referring to an already-compiled `HelloWorld.class`. You can add arguments such as `-classpath`, but you cannot ask this facility to compile more than one file. Being able to run a single file can be a time-saver when you are exploring APIs, although for that use, JShell may be more convenient. This capability is often useful for making sure a single file will compile and do something useful.

Text blocks (introduced with Java 13)

Consider the following code:
Text blocks, also known as multiline strings, make this easier. With text blocks (previewed in Java 13 but now standard), the above example would become the following:

```
String paraV1 =
    "Creating long strings over multiple lines is tedious."
    "The developer could forget the '+' at the end of the sentence commonly forget the space or the '\n' at the end of the line"

String paraV2 = ""
    Creating long strings over multiple lines is tedious.
    The developer could forget the '+' at the end of the sentence commonly forget the space or the '\n' at the end of the line
```

The text block requires a newline character after the initial opening triple quote, discards leading indentation, and processes new lines seamlessly.

In this example, `paraV1.equals(paraV2);` returns `true`.

Text blocks are a big saving in time and typing when you're writing helpful messages to the user or any other long strings. Learn more in Mala Gupta's article “Text blocks come to Java.”

**Records (introduced with Java 14)**

Java developers have poured countless hours into writing and tweaking what are essentially plain old Java object (POJO) data classes. You know the drill: Create a class, add some fields, add a constructor to make sure the fields are initialized, generate getters and setters, generate `equals()` and `hashCode()`, write a `toString()`, and so on.

Records automate this process. A `record` class extends the new class `java.lang.Record` with a list of fields and all the boilerplate methods listed above—all generated automatically. Here is a very simple `Person` record, which contains a person’s name and email address.

```
public record Person(String name, String email) {

The syntax seems a bit odd at first and looks like a merger between a class definition and a method definition. That's because that's basically what it is. You pass the arguments for the constructor, and they all become constructor arguments, fields, and accessor. The compiler will generate all the other necessary pieces.

Here's what is actually generated as seen by the `javap` disassembler on the compiled `Person` record and on its parent class, `java.lang.Record`.

```
$ javap 'structure.RecordDemoPerson$Person'
Compiled from "RecordDemoPerson.java"
public final class structure.RecordDemoPerson$Person extends java.lang.Record
```
In the following simple demo, the constructor arguments are in the same order as on the record’s definition. The getter methods do not follow the `getName` pattern; you simply request the field or invoke it as a method name.

```java
public static void main(String[] args) {
    Person p = new Person("Covington Roderick Smythe III", "rodrick.smythe@demo.com");
    System.out.println(p);
    System.out.println(p.name);
    System.out.println(p.name());
}
```

Executing that code shows the following:

```
$ java RecordDemoPerson.java
Person[name=Covington Roderick Smythe III, email=rodrick.smythe@demo.com]
Covington Roderick Smythe III
Covington Roderick Smythe III
```

The demo retrieves the name both as a field and as a method to show that you can use either, so the name prints twice in the output.

You can add additional methods, but all fields are immutable. A good clue is that no setter methods are provided. If you need to change a record, simply create a new one; they are very lightweight.

For more on records, see “Records come to Java” by Ben Evans. The immutability and the lack of setter and getter method names can be an issue with some frameworks; see “Diving into Java records: Serialization, marshaling, and bean state validation” by Frank Kiwy for ideas on how to cope with that.

Record classes are short, simple, and sweet. You should be using them; they will save you a lot of typing and significantly cut the amount of code you have to read (and maintain).

**jpackage (introduced with Java 14)**

`jpackage` produces installers to load your software onto desktop computers. These installers include a minimized JDK containing the
On each platform, `jpackage` drives another tool to do the actual creation of the installer. On systems commonly used by developers, there are built-in package creation tools, such as the `pkg` and `dmg` tools on macOS or `rpm` and `deb` on Linux. On Windows, `jpackage` requires a couple of additional tools. You must run `jpackage` on a given platform to generate an installer for that platform.

Given the range of features people expect from an installer, it's not surprising that there are many command-line options for `jpackage`. For example, I use `jpackage` to create an installer for my PDF presentation tool, `pdfshow`, on GitHub. Rather than trying to remember all the options, I wrote a script that uses the operating system type to provide the desired flags. The script is too long and tangential to include here; you can see it in the GitHub repository under the name `mkinstaller`. The heart of it is the call to `jpackage`, which looks like the following (several variables have been set earlier in the script):

```bash
jpackage \
  --name PDFShow \
  --app-version ${RELEASE_VERSION} \n  --license-file LICENSE.txt \
  --vendor "${COMPANY_NAME}" \
  --type "${inst_format}" \
  --icon src/main/resources/images/logo.${icon_for} \n  --input target \
  --main-jar pdfshow-${RELEASE_VERSION}-jar-with-d\n  ${OS_SPECIFIC}
```

The resulting installers have been used to install `pdfshow` on all three major platforms. If you want to try one of the installers, check the `Releases` tab on the GitHub page.

**Sealed types (introduced with Java 15)**

"Unlimited subclassing is the root of all evil," somebody once said in a discussion of software architecture. Sealed types, previewed in Java 15 and 16, provide the author of a class with control over its subclasses. For example, in the following

```java
public abstract sealed class Person
    permits Customer, SalesRep, Manager {...}
```

only the three named classes (in the same package) are allowed to subclass directly from `Person`. However, the sealed types mechanism can be used to open up subclassing one level below the sealed class, as follows:

```java
sealed class A permits B {
  ...
}
non-sealed class B extends A {
  ...
}
```
You can read more about sealed types in "Inside the language: Sealed types" by Ben Evans.

**Conclusion**

Many wonderful improvements have been made to Java throughout its long history. The majority have to do with making code clearer, more expressive, more powerful, and more concise. If you are not using all of these wonderful improvements in your own code, I encourage you to use them to modernize your software.

**Dig deeper**

- 4 ready-to-try Java tools your team may not know about
- Behind the scenes: How do lambda expressions really work in Java?
- A peek into Java 17: Continuing the drive to encapsulate the Java runtime internals
- 12 recipes for using the Optional class as it’s meant to be used
- Text blocks come to Java
- Records come to Java
- Diving into Java records: Serialization, marshaling, and bean state validation
- Inside the language: Sealed types

---

**Ian Darwin**

Ian Darwin (@Ian_Darwin) is a Java Champion who has done all kinds of development, from mainframe applications and desktop publishing applications for UNIX and Windows, to a desktop database application in Java, to healthcare apps in Java for Android. He’s the author of *Java Cookbook* and *Android Cookbook* (both from O'Reilly). He has also written a few courses and taught many at Learning Tree International.