Informatics Large Practical: Java 11 Language Features

Stephen Gilmore and Paul Jackson

School of Informatics

Thursday, 1st October 2020

Choosing a programming language

- More than any other aspect, the choice of programming language has a massive impact on the difficulty of a programming task.
- Choice of programming language for a project can literally be the difference between success and failure.
- Programming languages come in different versions. We have chosen Java 11 for this practical.

Why do we have to upgrade to Java 11?

- You might ask, *"Java 8 works fine for me, which do I have to switch to Java 11?"*
- Programming languages are software, and as software gets older it becomes more expensive and difficult to maintain.
- Commercial updates for Java 8 ended in December 2019, and private-use updates will end in December 2020.
- Additionally, newer versions of Java have languages features and libraries which could be helpful to us.

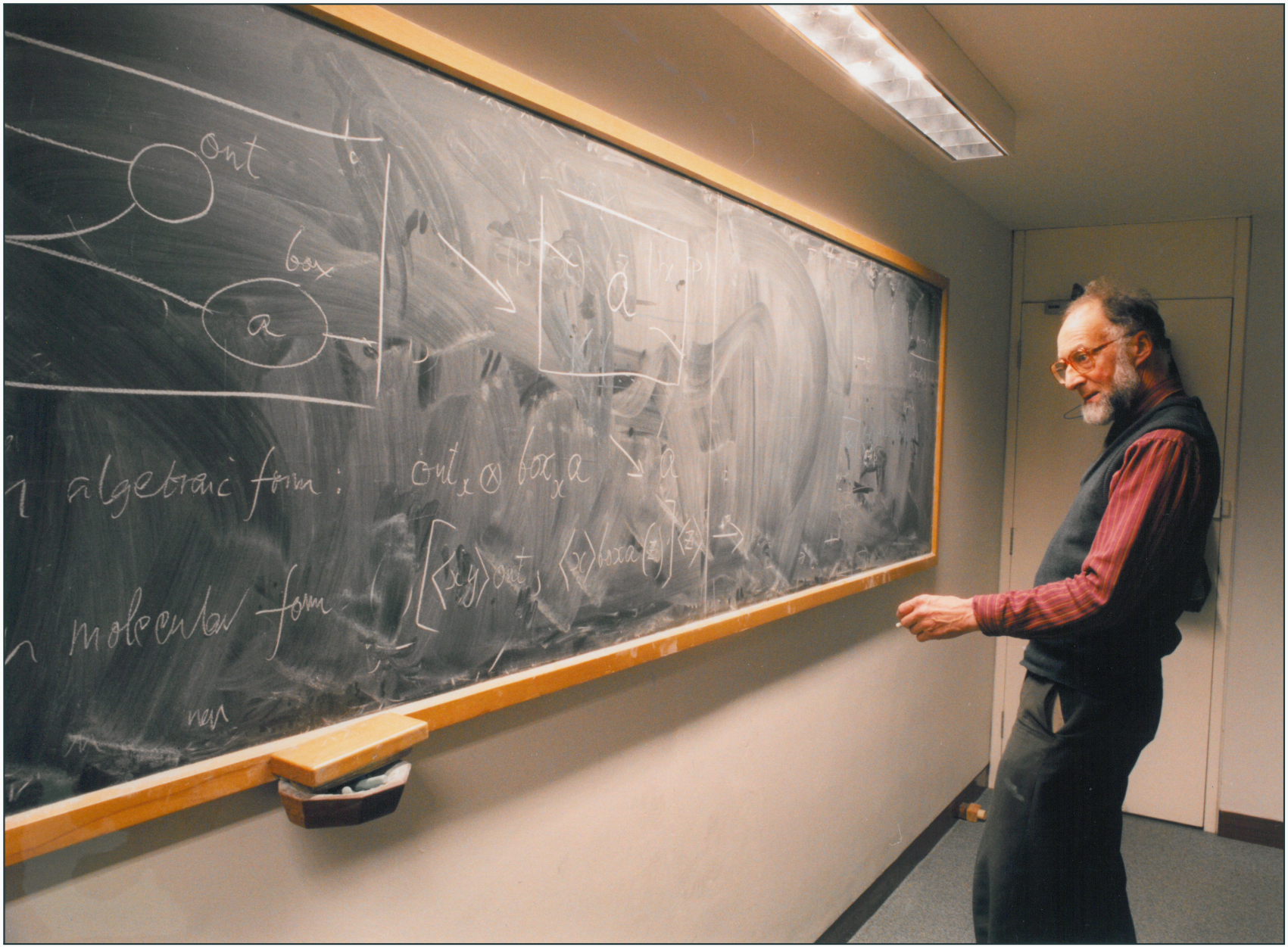
Why don't we upgrade to Java 15?

- You might ask, *"If we want to be up-to-date, why aren't we using Java 15 (released Sept 2020)?"*
- The answer is that some Java versions have a special status, known as Long Term Support (LTS) releases, meaning that they are maintained for a longer period of time than standard releases.
  - Java 8 and Java 11 are LTS releases: Java 15 is not. The next LTS release will be Java 17 (coming in 2021).
- Java 8 has had long-term support. Java 11 is just beginning its long-term support, so it is a good Java version to choose.

An advantage of Java 11

- In software development, a primary concern should be writing clear and readable code which others can understand.
- Even seemingly small improvements in a program text can make its meaning clearer and make it easier to read.
- One such improvement for Java — introduced in Java 10 — is type inference for local variables.

Edinburgh University's Robin Milner, inventor of Algorithm W



Local variable type inference in Java

- A common criticism of Java is that it requires too much ceremony. That is, a lot of text can be required to achieve a relatively simple task, and sometimes the text is repetitive.
- For example, the declaration of a local variable can be this

```
List<List<Integer>> matrix = new ArrayList<List<Integer>>();
```

Local variable type inference in Java

- A common criticism of Java is that it requires too much ceremony. That is, a lot of text can be required to achieve a relatively simple task, and sometimes the text is repetitive.
- For example, the declaration of a local variable can be this

```
List<List<Integer>> matrix = new ArrayList<List<Integer>>();
```

- With local variable type inference this becomes:

```
var matrix = new ArrayList<List<Integer>>();
```

  - The type of the matrix variable is inferred from the expression on the right-hand side of the equals, and the type declaration is removed.

Uses of local variable type inference

- Use it with local variables which are declared in methods.
- Local variable type inference combines with Java modifiers such as final.

```
// Declare a constant, origin, of type Point.
final var origin = Point.fromLngLat(longitude, latitude);
```

- It can be used with iteration variables in enhanced for loops.

```
for (var line : lines) {
        System.out.println(line);
}
```

Non-uses of local variable type inference

- Local variable type inference cannot be used for a class field.
- It cannot be used for uninitialised local variables.

```
var x; // not enough information for type inference
```

- It cannot be used with variables initialised with null.

```
var x = null; // not enough information for type inference
```

- It cannot be used for method parameters.

```
void print (var x) { // conflicts with method overloading
    ...
}
```

That's it!

- Local variable type inference is an easy way to make Java code cleaner and simpler.
- Please use it in your code!

Thank you for listening.