Modern Java IN ACTION

Raoul-Gabriel Urma
Mario Fusco
Alan Mycroft

Lambdas, Streams,
Functional and Reactive Programming

Part 1 Fundamentals

1. Java 8, 9, 10, and 11: what’s happening

This chapter covers

- Why Java keeps changing
- Changing computing background
- Pressures for Java to evolve
- Introducing new core features of Java 8 and 9

1.1 So, what's the big story?

- Java Version History
  - JDK 1.0 (Jan. 1996) ~ Java SE6 (Dec. 2006)
  - Java SE 7 – July 2011
  - Java SE 8 – March 2014
    - **Lambda expressions**, **default methods**, and **functional-style operations** of **streams of elements**
  - Java SE 9 – September 2017
    - Modularization of the JDK under Project Jigsaw (Java Platform Module System)
    - More concurrency updates.[297] It includes a Java implementation of Reactive Streams
  - Java SE 10 – March 2018
  - …
  - Java SE 13 – September 2019

to write programs more easily. For example, instead of writing verbose code (to sort a list of apples in `inventory` based on their weight) like

```
Collections.sort(inventory, new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

in Java 8 you can write more concise code that reads a lot closer to the problem statement, like the following:

```
inventory.sort(comparing(Apple::getWeight));
```

The first Java 8 code of the book!

- Hardware Influence – Multicore CPU
- Prior to Java 8 – Use threads to use these cores
  - The problem is that working with threads is difficult and error-prone.
  - Java has followed an evolutionary path of continually trying to make concurrency easier and less error-prone.
- Java 1.0 – threads, locks, and a memory model
- Java 5 – thread pools and concurrent collections
- Java 7 – fork/join framework
- Java 8 – a new, simpler way of thinking parallelism
- Java 9 – reactive programming
  (RxJava and Akka reactive streams toolkits)

- More concise code and simpler use of multicore processors in Java 8.
  - The Streams API
  - Techniques for passing code to methods
    - Anonymous classes
    - Lambda expressions
    - Methods references
  - Default methods in interface
- Java 9 adds modules

1.2 Why is Java still changing?

- Java is a well-designed object-oriented language with many useful libraries.
- Compile Java to JVM (Java Virtual Machine) bytecode for internet applet programs (applets).
- Scala, Groovy, and Kotlin programming language also run on the JVM.
- Climate change (multicore processors, new programming influences)
  - Programmers are increasingly dealing with big-data.
  - To exploit multicore computers or computing clusters effectively.

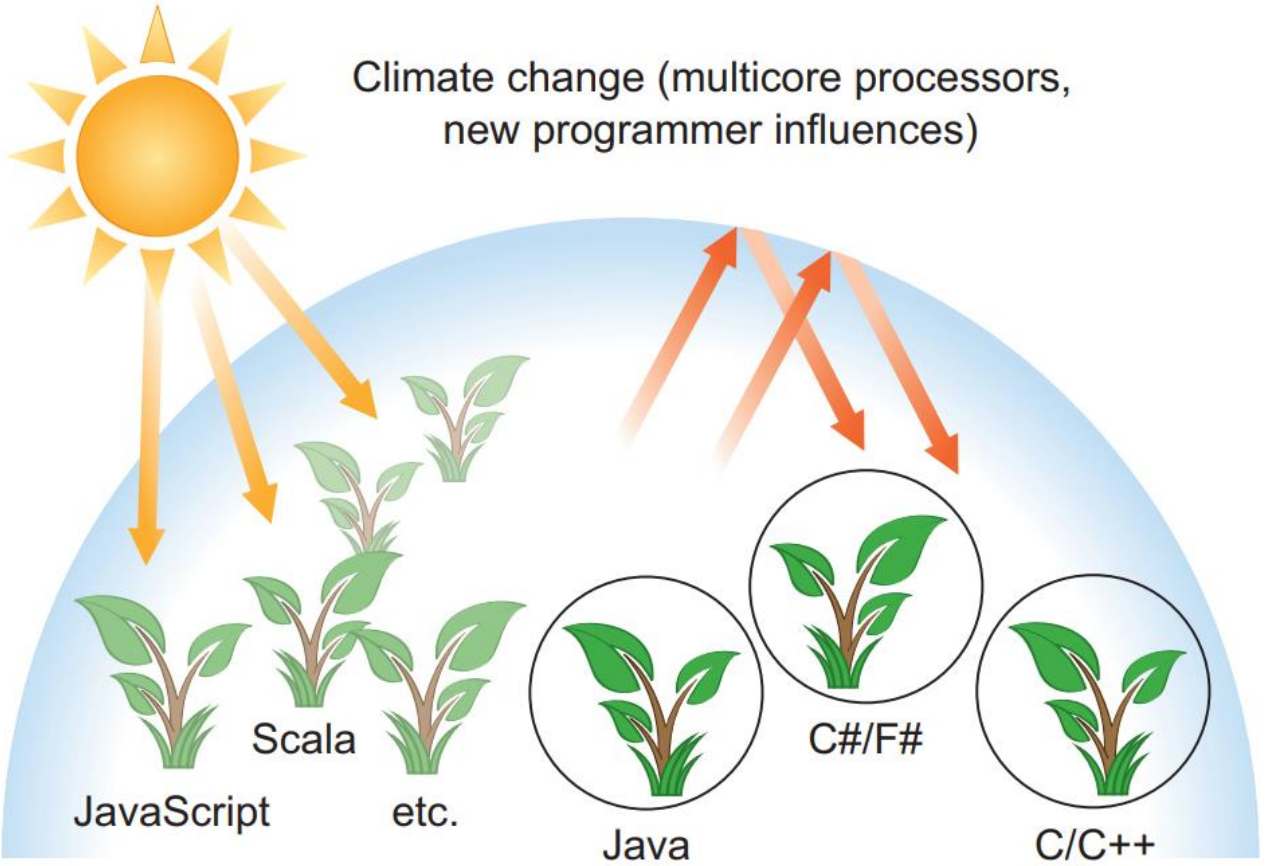


Figure 1.1 Programming-language ecosystem and climate change

Stream processing

- Unix command line
  - cat file1 file2 | tr “[A-X]” “[a-z]” | sort | tail -3
- Java 8 adds a Streams API in java.util.stream; Stream<T> is a sequence of items of type T.

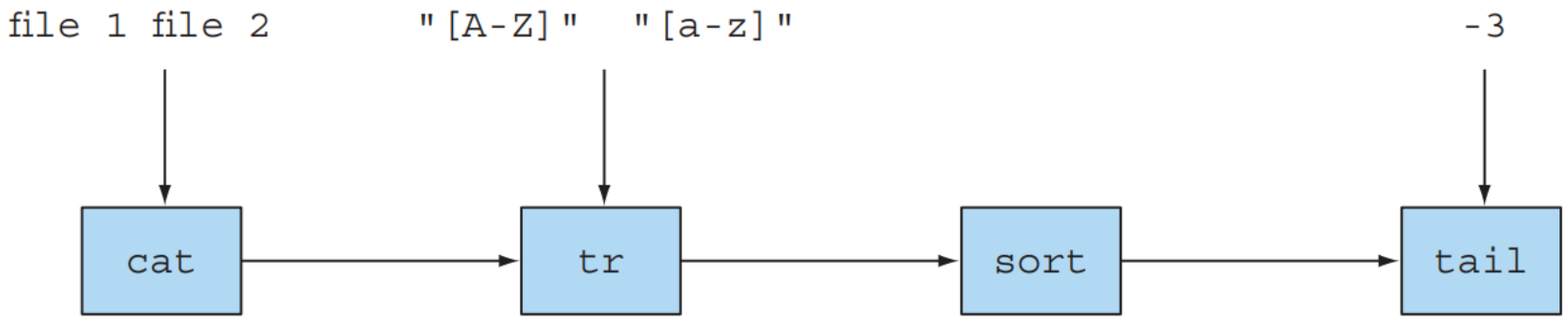


Figure 1.2 Unix commands operating on streams

Passing code to methods with behavior parameterization

- Java 8 has the ability to pass a piece of code to an API.
- For example, a collection of invoice IDs with a format like 2013UK0001, 2014US00e2, and so on.
  - four digits represent the year
  - two letters for a country code
  - the last four digits for the ID of a client
- Sort a collection of invoice IDS with different criterias

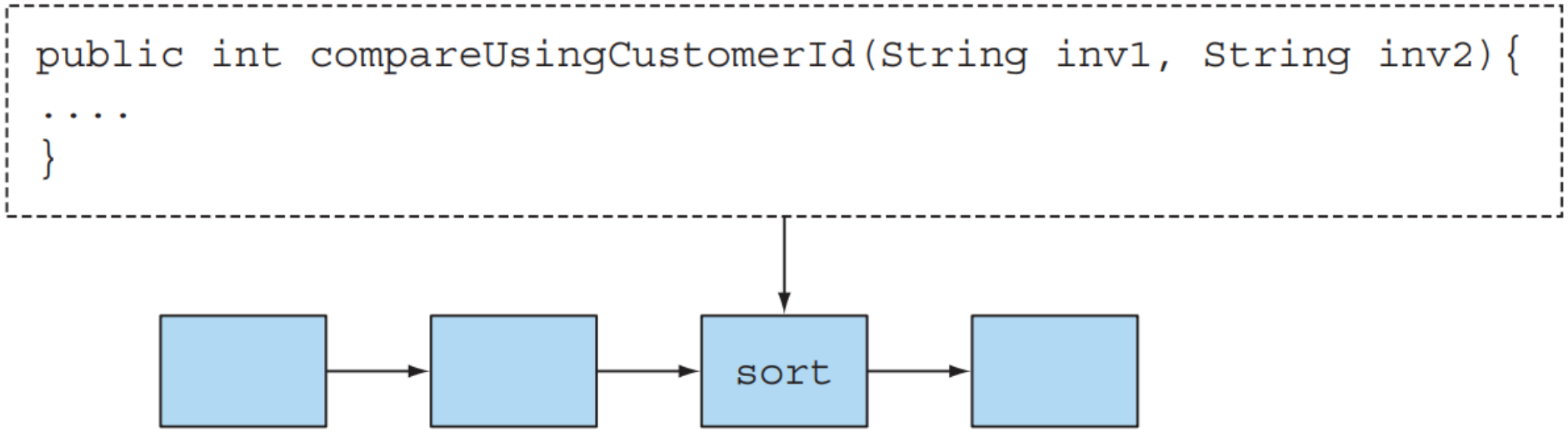


Figure 1.3 Passing method `compareUsingCustomerId` as an argument to `sort`

1.3 Functions in Java

- Function, Method, Procedure, Subroutine, ….
- *Mathematical function*, one without side effects.
- Java 8 adds functions as new forms of value.
  - In java programs, possible values are primitive values like 42 (type int) and objects (using new operator). Examples include “abc” (of type String) and new Integer(1111) (of type Integer).
  - Object references point to *instances* of a class.
  - Firs-class values (or citizens) vs. Second-class citizens (which can’t be passed around during program execution)
- Passing methods around at runtime, and hence making them first-class citizens, is useful in programming.

Methods and lambdas as first-class citizens

- The Java 8 feature of methods as values forms the basis of various other Java 8 features (such as Streams).
- Method References
  - Example: To filter all the hidden files in a directory.

```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
    public boolean accept(File file) {
        return file.isHidden();
    }
});
```

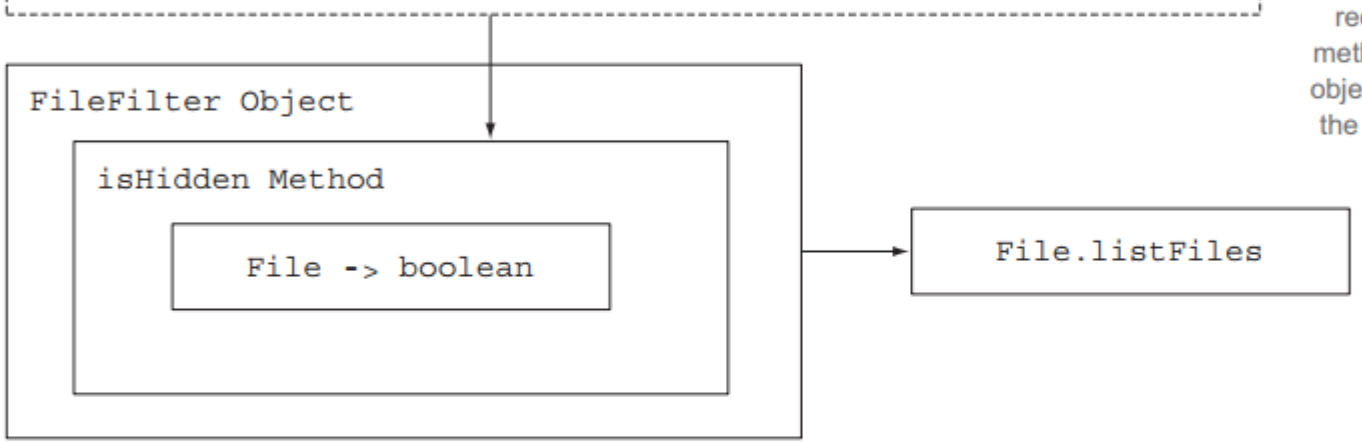
Filtering hidden files!

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

method reference

Old way of filtering hidden files

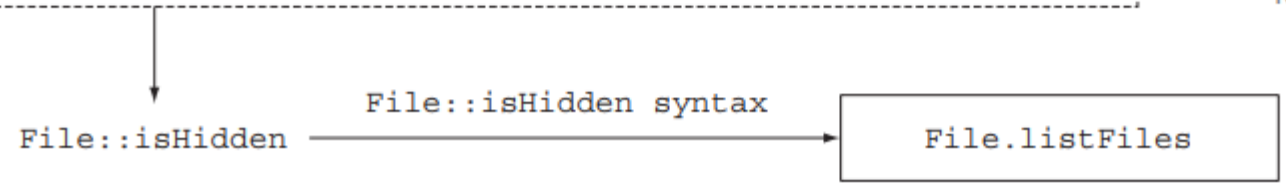
```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
   public boolean accept(File file) {
     return file.isHidden();
   }
});
```



Filtering files with the isHidden method requires wrapping the method inside a FileFilter object before passing it to the File.listFiles method.

Java 8 style

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden)
```



In Java 8 you can pass the isHidden function to the listFiles method using the method reference :: syntax.

Figure 1.4 Passing the method reference `File::isHidden` to the method `listFiles`

Passing code: an example

- Example 1: Select all green apples and return them in a list.
  - Before Java 8

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (GREEN.equals(apple.getColor())) {
            result.add(apple);
        }
    }
    return result;
}
```

The result list accumulates the result; it starts as empty, and then green apples are added one by one.

The highlighted text selects only green apples.

Passing code: an example (cont.)

- Example 2: Return the list of heavy apples (say over 150g)
  - Before Java 8

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getWeight() > 150) {
            result.add(apple);
        }
    }
    return result;
}
```

Here the highlighted text selects only heavy apples.

Passing code: an example (cont.)

- Java 8 makes it possible to pass the code of the condition as an argument.

```
public static boolean isGreenApple(Apple apple) {
    return GREEN.equals(apple.getColor());
}
public static boolean isHeavyApple(Apple apple) {
    return apple.getWeight() > 150;
}
public interface Predicate<T>{
    boolean test(T t);
}
static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
```

Included for clarity (normally imported from java.util.function)

A method is passed as a Predicate parameter named p (see the sidebar "What's a Predicate?").

```
        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

Does the apple match the condition represented by p?

And to use this, you call either

```
filterApples(inventory, Apple::isGreenApple);
```

or

```
filterApples(inventory, Apple::isHeavyApple);
```

From passing methods to lembdas

- Anonymous Functions, or lambdas

```
filterApples(inventory, (Apple a) -> GREEN.equals(a.getColor()) );
```

or

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

or even

```
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
                                     RED.equals(a.getColor()) );
```

You don't even need to write a method definition that's used only once;

From passing methods to lembdas (Cont.)

- static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
- filter(inventory, (Apple a) -> a.getWeight() > 150);

1.4 Streams

- Nearly every Java program *makes* and *processes* collections.
- Example:

  You need to filter expensive transactions from a list and then group them by currency.
  Sample codes: the next page. (*external iteration*)
- Using the Stream API: (*internal iteration*)

```
import static java.util.stream.Collectors.groupingBy;
Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream()
                .filter((Transaction t) -> t.getPrice() > 1000)
                .collect(groupingBy(Transaction::getCurrency));
```

Filters expensive transactions

Groups them by currency

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
    new HashMap<>();
for (Transaction transaction : transactions) {
    if(transaction.getPrice() > 1000){
        Currency currency = transaction.getCurrency();
        List<Transaction> transactionsForCurrency =
            transactionsByCurrencies.get(currency);
        if (transactionsForCurrency == null) {
            transactionsForCurrency = new ArrayList<>();
            transactionsByCurrencies.put(currency,
                                         transactionsForCurrency);
        }
        transactionsForCurrency.add(transaction);
    }
}
```

Creates the Map where the grouped transaction will be accumulated

Iterates the List of transactions

Filters expensive transactions

Extracts the transaction's currency

If there isn't an entry in the grouping Map for this currency, create it.

Adds the currently traversed transaction to the List of transactions with the same currency

Multithreading is difficult

- Exploit Parallelism by writing multithreaded code (using the Thread API form previous version of Java) is difficult.
  - Threads can access and updates shared variables at the same time.
    For example, two threads try to add a number to a shared variable sum if they're not synchronized properly.
- Java 8 provides the Stream API (java.util.stream)
  - Data processing patterns – *filtering* data, *extracting* data, *grouping* data.
  - Such operations can often be parallelized.
    For example, filtering a list on two CPU's.
    - Forking Step (1), Filtering Step (2), and Joining Step (3).
- Collections vs. Streams
  - Collections for storing and accessing data.
  - Streams for describing computations on data.

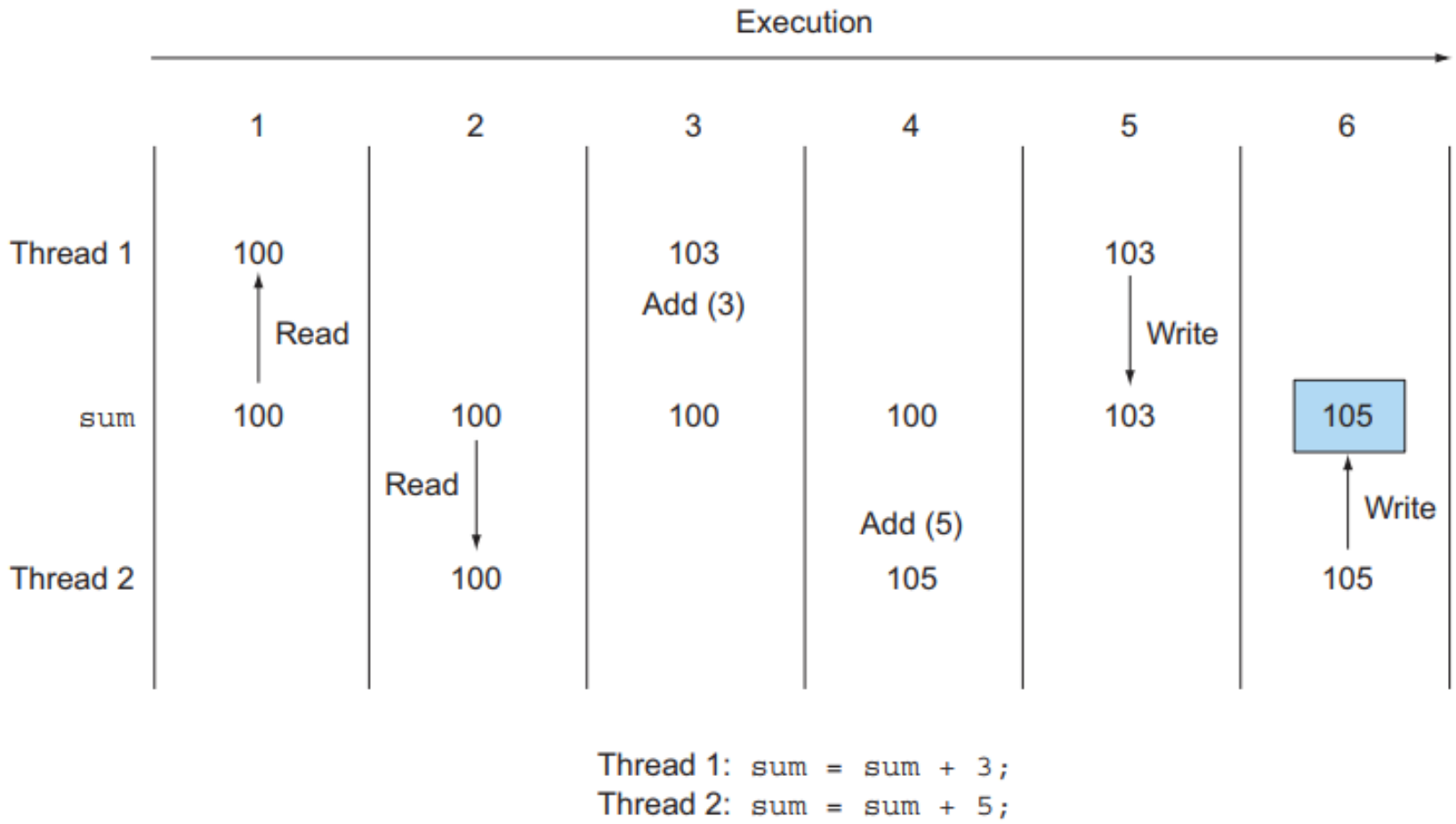


Figure 1.5 A possible problem with two threads trying to add to a shared `sum` variable. The result is 105 instead of an expected result of 108.

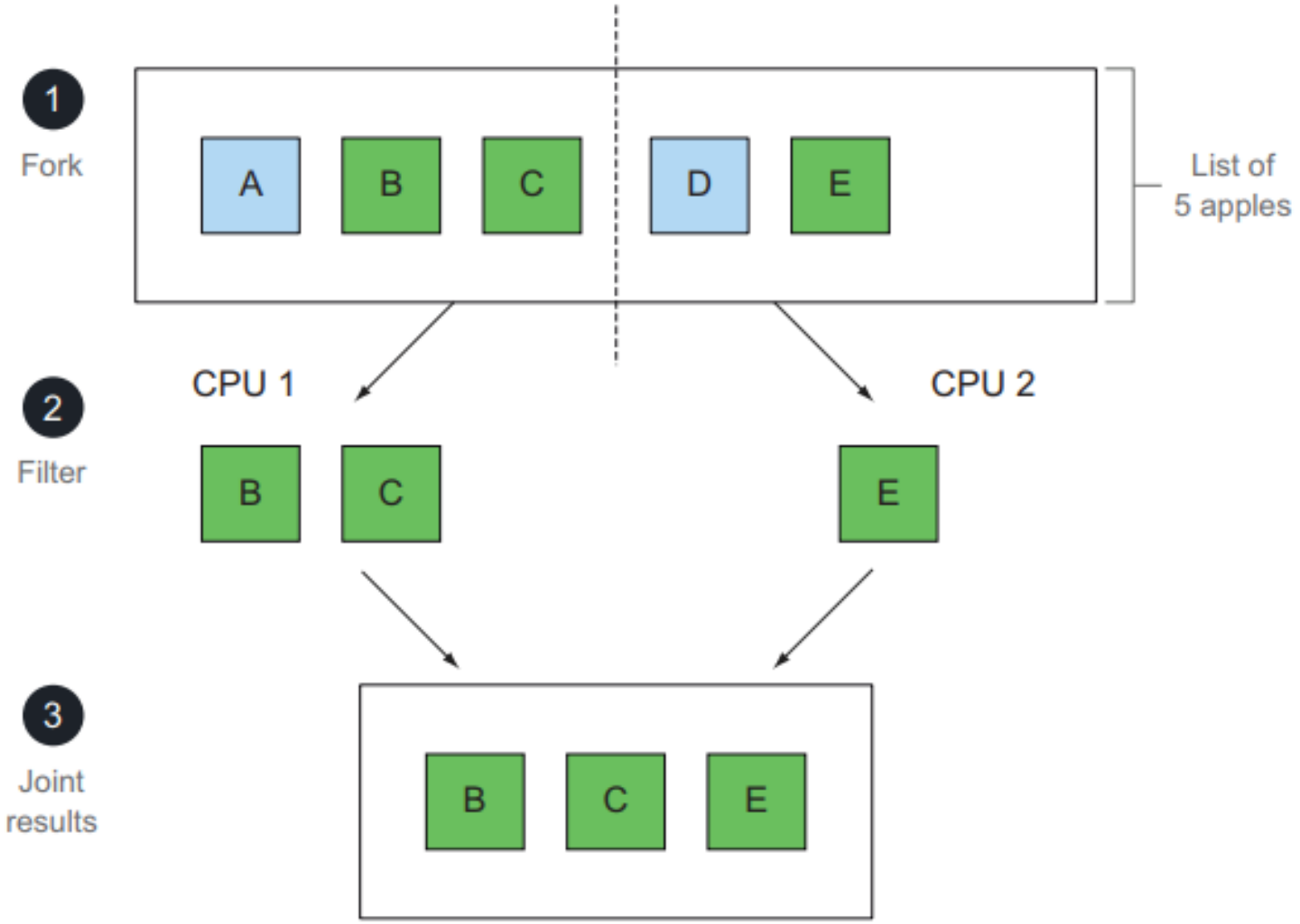


Figure 1.6 Forking `filter` onto two CPUs and joining the result

Sequential processing vs. Parallel processing

- Sequential Processing

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
```

- Parallel Processing

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

1.5 Default methods and Java modules

- a Jar file containing a set of java packages with no particular structure.
  - evolving interfaces to such packages was hard – changing a Java interface meant changing every class that implements it.
- Java 9 provides a module system that provide you with syntax to define *modules* containing collections of packages.
  - User documentation
  - Machine checking
- Java 8 added default methods to support *evolvable interfaces*.

In section 1.4, we gave the following example Java 8 code:

```
List<Apple> heavyApples1 =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
List<Apple> heavyApples2 =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

A List<T> prior to Java 8 doesn't have stream or parallelStream methods – and neither does the Collection<T> interface that it implements – because theses method hadn't been conceived of.
The Java 8 List interface has a default method sort calling the static Collections.sort method.

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

1.6 Other good ideas from functional programming

- Two core ideas from functional programming that are now part of Java.
  - using methods and lambdas as first-class values.
  - calling to functions or methods can be efficiently and safely executed in parallel in the absence of mutable shared state.
- Java 8 introduced the Optional<T> class to help you avoid null-pointer exceptions.
  - It is a container object that may or may not contain a value.

1.6 Other good ideas from functional programming (Cont.)

- (Structural) Pattern Matching
  - It can express programming ideas more concisely compared to using if-then-else.
  - For example:

```
f(0) = 1
f(n) = n*f(n-1) otherwise
```

- Java 8 doesn't have full support for pattern matching.

In Scala, you can write the following code to decompose an Expr into its part and then return another Expr:

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e
    case BinOp("-", e, Number(0)) => e
    case BinOp("*", e, Number(1)) => e
    case BinOp("/", e, Number(1)) => e
    case _ => expr
}
```

Adds 0

Subtracts 0

Multiplies by 1

Divides by 1

Can't be simplified with these cases, so leave alone

Here Scala's syntax `expr match` corresponds to Java's `switch (expr)`.

JVM Languages

- Clojure, a modern, dynamic, and functional dialect of the Lisp programming language[1]
- Groovy, a dynamic programming and scripting language[1]
- JRuby, an implementation of Ruby
- Jython, an implementation of Python
- Kotlin, a statically-typed language from JetBrains, the developers of IntelliJ IDEA[1]
- Scala, a statically-typed object-oriented and functional programming language[2]

JVM Languages (Cont.)

- A Complete Guide to JVM Languages
- A Quick Guide to the JVM Languages
- JVM Programming Languages – The Expert's Guide To Creating Software For The Java Virtual Machine
- 3 JVM Languages Modern Java Developers Should Learn in 2018
- InfoQ – JVM Languages
- What are the best languages that run on the JVM?

References

- Pair programming? That's so 2017. Try out this deep-learning ...
- **Pairing coders with artificial intelligence to write software**
- Top five programming languages for AI and machine learning you should learn this year
- Codota Offers Pair Programming with Artificial Intelligence ...
  - Codota - https://www.codota.com/
- How AI and Software 2.0 will change the role of programmer

Git References

- https://backlog.com/git-tutorial/tw/

Apache Maven and Gradle

- Apache Maven
  https://en.wikipedia.org/wiki/Apache_Maven
- Apache Maven Project
  https://maven.apache.org/
- Gradle
  https://en.wikipedia.org/wiki/Gradle
- Gradle Build Tool
  https://en.wikipedia.org/wiki/Gradle

Jenkins and Jenkins X

- CI (Continuous Integration)
  - Continuous Integration: What is CI? Testing, Software ...
- CD (Continuous Delivery)
  - What is Continuous Delivery? - Continuous Delivery
- CD (Continuous Deployment)
  - What Is Continuous Deployment? | Atlassian
- Continuous integration vs. continuous delivery vs. continuous ...
- https://www.atlassian.com/continuous-delivery
- References
  - https://en.wikipedia.org/wiki/Jenkins_(software)
  - Jenkins - https://jenkins.io/ + Jenkins X - https://jenkins-x.io/
  - [Jenkins]持續整合之路(一)Jenkins Master Server安裝| 史丹利 ...