Lecture 1

Introduction

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

René Witte

Introduction
Motivation
Reactive Programming
Play Framework
Java 8
Introduction
Functional Programming
Streams
Multithreading
Default Methods
Notes and Further Reading

Outline

René Witte

Motivation

René Witte
Concordia University

Consider any modern large-scale web application:

What are some important implementation considerations?

- Scalability
- Robustness (fault-tolerance)
- Responsiveness
- . . .

But how do we build such systems?

René Witte
Concordia University

The Reactive Manifesto, https://www.reactivemanifesto.org/

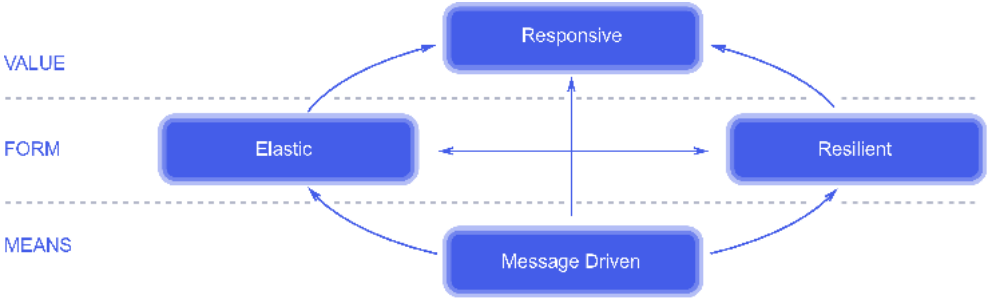
"Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems."

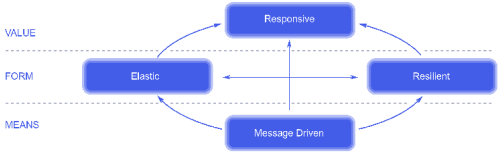
Reactive Systems

René Witte



Reactive Systems: Responsive Property

René Witte

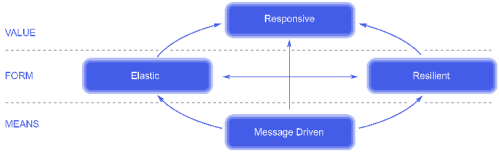


Responsive

“ The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.”

Reactive Systems: Resilient Property

René Witte

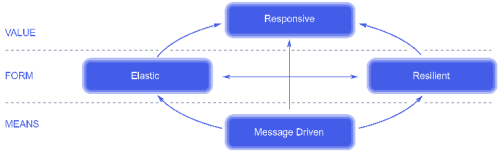


Resilient

"The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures."

Reactive Systems: Elastic Property

René Witte

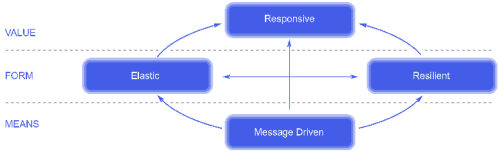


Elastic

"The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms."

Reactive Systems: Message Driven Property

René Witte



Message Driven

"Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency. This boundary also provides the means to delegate failures as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead."

René Witte

Programming foundations covered in this course:

- Functional Programming (Lambdas): Java 8
- Stream Data Processing: Java 8
- Concurrent (Multi-Core) Programming: Java 8
- Message Passing & Fault Tolerance: Akka library
- Full-stack web application: Play framework

Play Framework (https://www.playframework.com/)

René Witte
Concordia University

Java 8

René Witte
Concordia University

Timeline

- Java (JDK) 1.0: 1996
- Java 8: March 2014
- Java 9: September 2017
- Java 10: March 2018

More Concise Code

Instead of:

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

You can write:

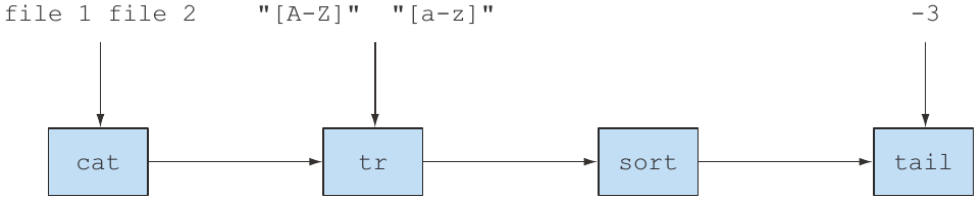
```
inventory.sort(comparing(Apple::getWeight));
```

Streams

René Witte
Concordia University

Unix/Linux pipes

```
cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3
```

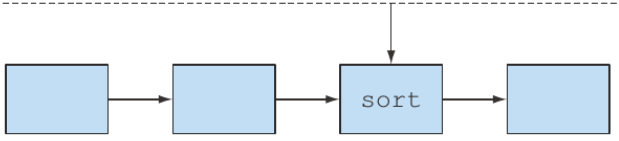


Java 8: Streams API (`java.utils.stream`)

Behavior Parameterization

René Witte


```
public int compareUsingCustomerId(String inv1, String inv2){
....
}
```



René Witte
Concordia University

Functions become first-class citizens

Example: filter hidden files using `isHidden()`, old-style Java:

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

Now with Java 8:

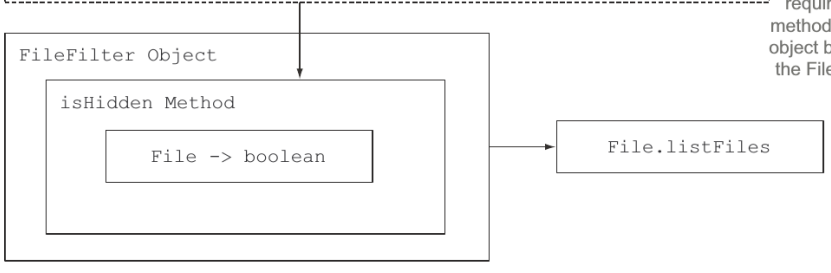
```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

Prior to Java 8: Can only pass object references

René Witte
Concordia University

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: Can now pass method references

René Witte

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

Passing Code

René Witte
Concordia University

Filtering a `List` of `Apple` objects

```
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;
}
```

Adding another filter...

```
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;
  }
```

With Java 8: Passing Code

```
public static boolean isGreenApple(Apple apple) {
  return "green".equals(apple.getColor());
}

public static boolean isHeavyApple(Apple apple) {
  return apple.getWeight() > 150;
}

static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}

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

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

René Witte

Lambdas

René Witte

With anonymous functions (lambdas)

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

and

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

René Witte
Concordia University

Example: filter list of transactions by amount and currency, old-style Java

```
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);
    }
}
```

Working with Streams (II)

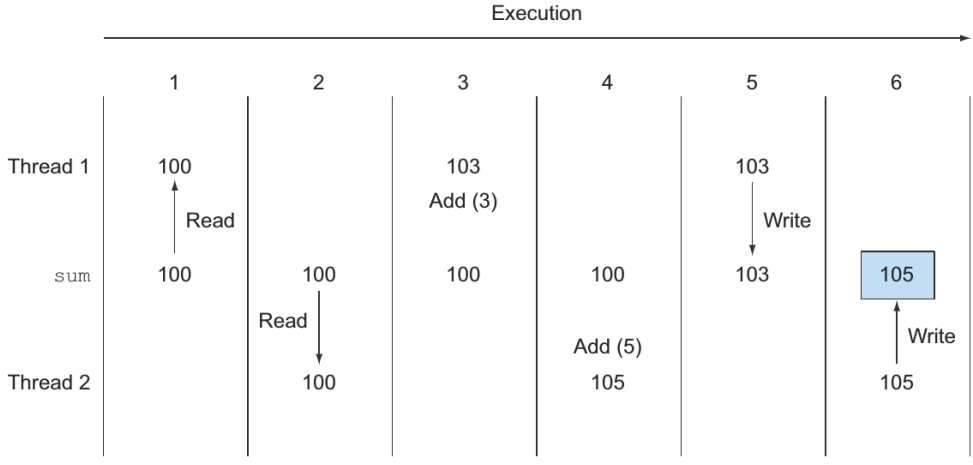
René Witte

Filter list of transactions by amount and currency, now with Java 8

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

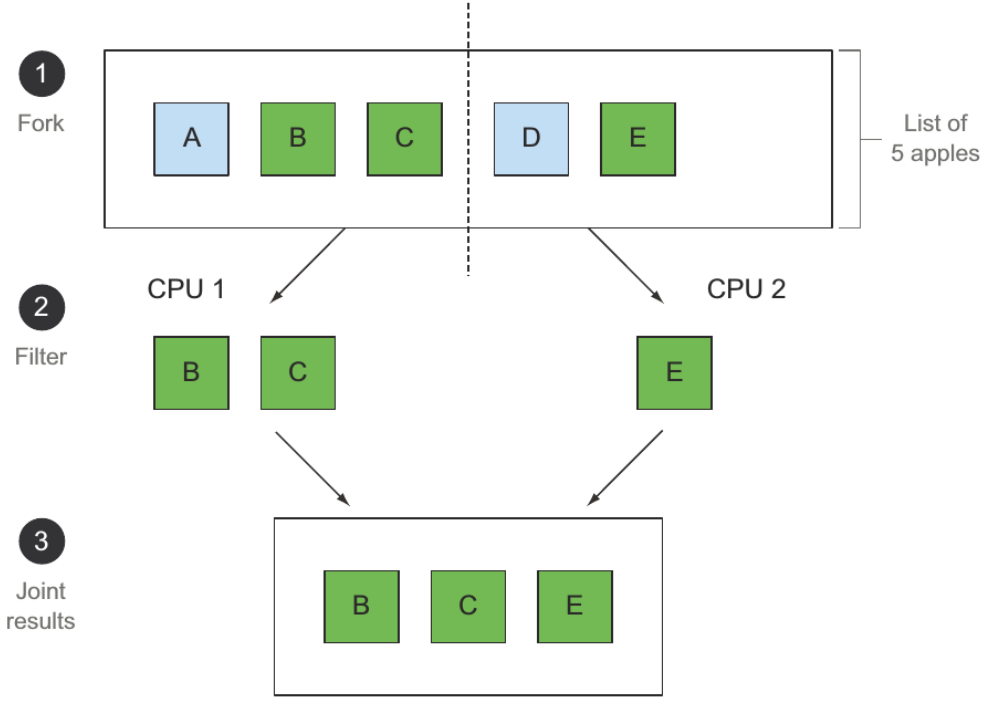
Multithreading: Shared Variables

René Witte



Thread 1: `sum = sum + 3;`
Thread 2: `sum = sum + 5;`

Multi-Core Processing



René Witte

Parallel Stream Processing

René Witte

Filter list, sequential processing

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

Filter list, parallel processing

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

Default Methods

René Witte
Concordia University

Problem: Evolving Interfaces

How do we get a new function like `stream()` or `sort()` into an existing interface like `List()`?

Java 8: Default Methods

In Java 8, we can now have implementations in an interface (default methods):

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

Outline

René Witte

Reading Material

René Witte

Required

- [UFM14, Chapter 1] (Java 8: why should you care?)

Supplemental

- [Ber16, Chapters 1] (Introduction to reactive applications)

References

[Ber16] Manuel Bernhardt.
Reactive Web Applications.
Manning Publications, 2016.
https://www.manning.com/books/reactive-web-applications.

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.