

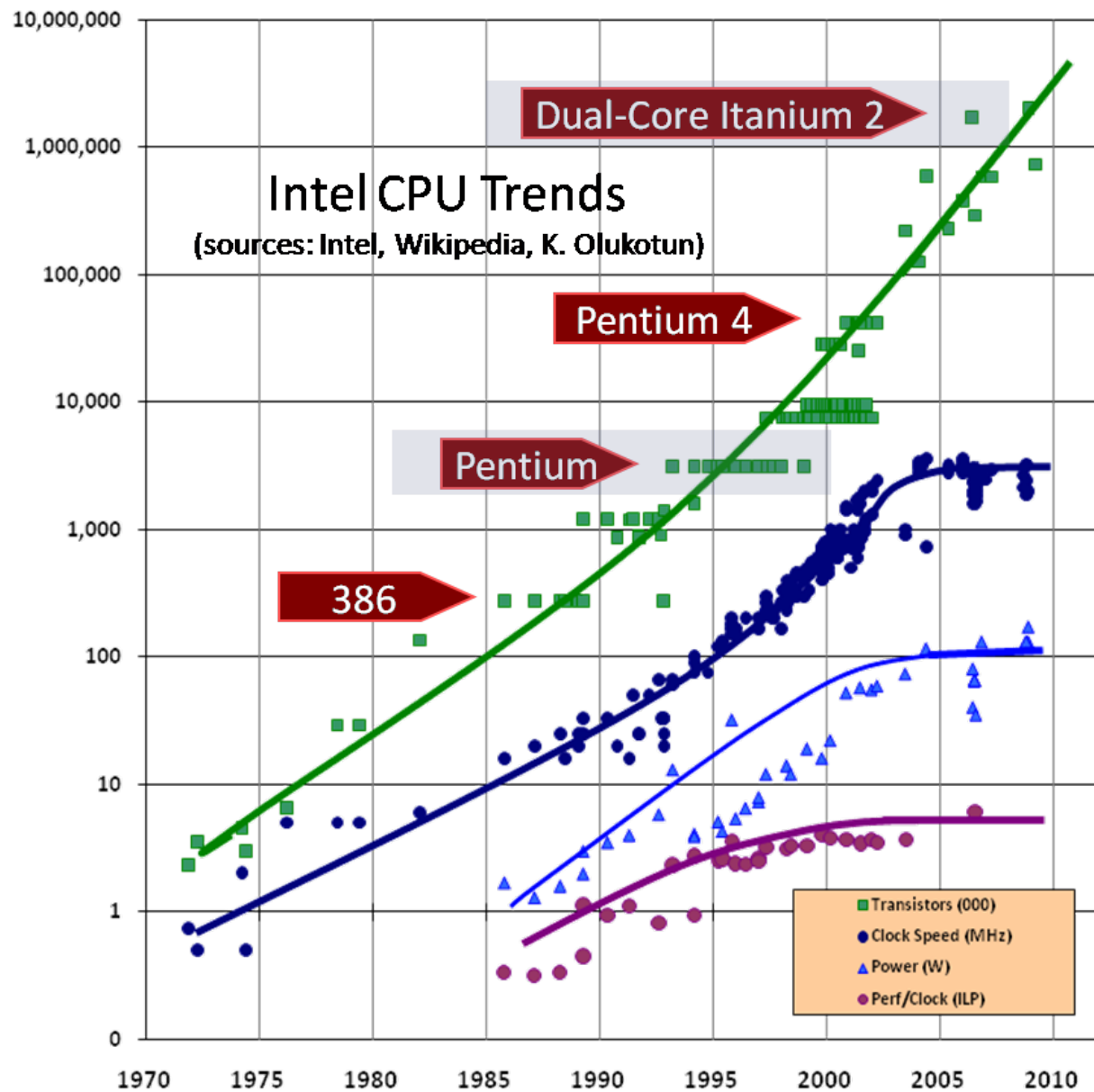
# Introduction to Java 8

Juan Manuel Gimeno Illa

[jmgimeno@diei.udl.cat](mailto:jmgimeno@diei.udl.cat)

# Evolution of Java

- **Java 1.0:** Initial version
- **Java 1.1:** Inner classes, JDBC, ...
- **Java 1.2:** Swing, Collections, ...
- **Java 5:** Generics, Annotations, Autoboxing, ...
- **Java 7:** Diamond operator, try-with-resources, ...
- **Java 8:** Lambda expressions, Streams, Default methods, ...



[The Free Lunch Is Over, by Herb Sutter \(2010\)](#)

# Concurrency in Java

- **Java 1.0:** `java.lang.Thread`, synchronized blocks
- **Java 5.0/6.0:** `java.util.concurrent` (JSR166)
- **Java 7.0:** Fork/Join Framework (JSR166)
- **Java 8.0:** Parallel Streams, Completable Futures

# LAMBDAS

# Behaviour parameterization

- It is the ability to take multiple behaviours as parameters and use them to accomplish different comportments
- It lets you make your code more adaptive to changing requirements
- Passing code is a way to give behaviours as arguments to a method
  - Using classes is very verbose
  - Even anonymous inner classes are verbose !!!

# Behaviour parameterization

```
public class Apple {  
    private int weight = 0;  
    private String color = "";  
    public Apple(int weight, String color){  
        this.weight = weight;  
        this.color = color;  
    }  
    public Integer getWeight() { return weight; }  
    public void setWeight(Integer weight) { this.weight = weight;}  
    public String getColor() { return color;}  
    public void setColor(String color) { this.color = color;}  
    public String toString() {  
        return "Apple{" +  
            "color='" + color + '\'' +  
            ", weight=" + weight +  
            '}';  
    }  
}
```

# Behaviour parameterization

```
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;  
}
```



# Behaviour parameterization

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

# Behaviour parameterization

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

# Behaviour parameterization

```
public interface ApplePredicate{  
    boolean test(Apple a);  
}
```

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

# Behaviour parameterization

```
public class AppleWeightPredicate implements ApplePredicate{  
    public boolean test(Apple apple){  
        return apple.getWeight() > 150;  
    }  
}
```

```
public class AppleColorPredicate implements ApplePredicate{  
    public boolean test(Apple apple){  
        return "green".equals(apple.getColor());  
    }  
}
```

# Behaviour parameterization

```
List<Apple> greenApples = filter(inventory, new ApplePredicate() {  
    @Override  
    public boolean test(Apple a) {  
        return "green".equals(a.getColor());  
    }  
});
```

This is the only  
information that  
“matters”.  
The rest of the code is  
“boilerplate”

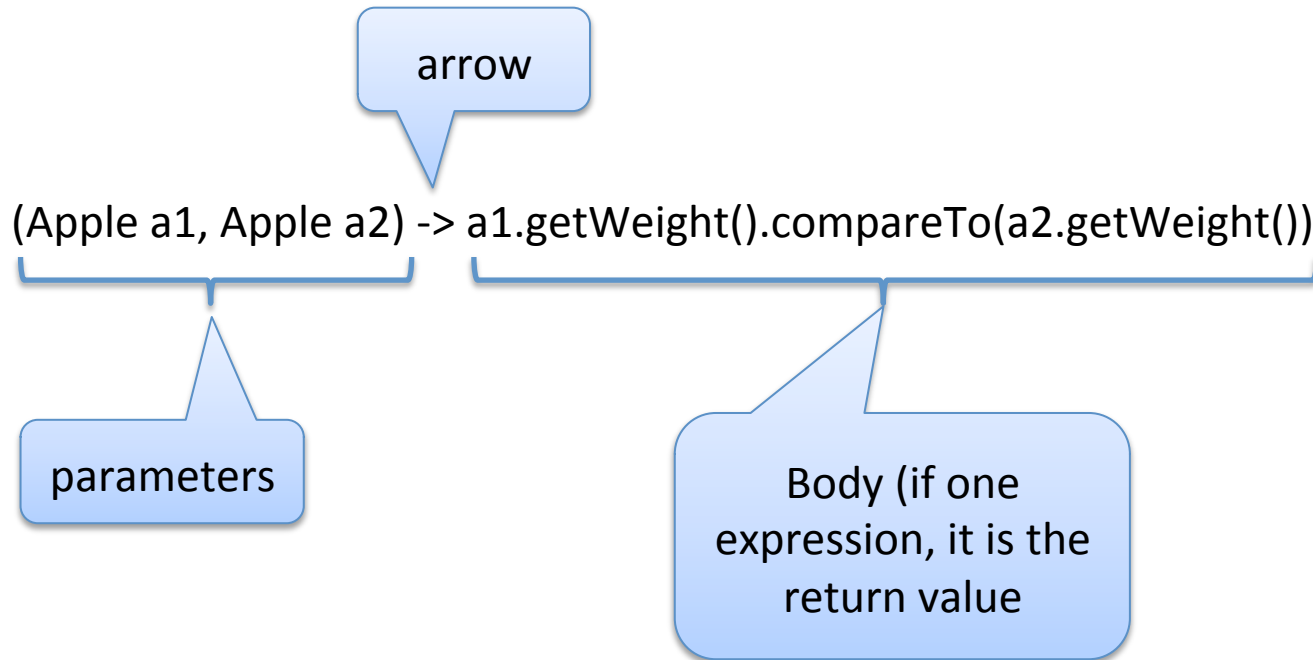
```
List<Apple> greenApples = filter(inventory, a -> "green".equals(a.getColor()));
```

We provide only the  
essential information

# Lambda expressions

*“A lambda expression can be understood as a concise representation of an anonymous function that can be passed around: it doesn’t have a name, but it has a list of parameters, a body, a return type and also possibly a list of exceptions that can be thrown”*

# Lambda expressions



- `(parameters) -> expression`
- `(parameters) -> { statements; }`

# Functional interface

- A functional interface is an interface that specifies exactly **one abstract method**

```
public interface Comparator<T> {  
    int compareTo(T o1, T o2);  
}  
public interface Runnable {  
    void run();  
}  
public interface ActionListener extends EventListener {  
    void actionPerformed(ActionEvent event);  
}  
public interface Callable<V> {  
    V call();  
}
```



# Functional interface

*“Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and treat the whole expression as an instance of a functional interface (more technically speaking, an instance of a concrete implementation of the functional interface)”*

```
Runnable r1 = () -> System.out.println("Hello World !!");
```

# Functional descriptor

- The signature of the abstract method of the functional interface describes the signature of the lambda expression
- We call this abstract method a **functional descriptor**
  - E.g. the Runnable interface can be viewed as a signature of a function which accepts nothing (no parameters) and returns nothing (void)

# Common functional interfaces

Functional Interface	Functional Descriptor
Predicate<T>	T -> boolean
Consumer<T>	T -> void
Function<T, R>	T -> R
Supplier<T>	() -> T
UnaryOperation<T>	T -> T
BinaryOperation<T>	(T, T) -> T
BiPredicate<L, R>	(L, R) -> boolean
BiConsumer<T, U>	(T, U) -> void
BiFunction<T, U, R>	(T, U) -> R

# @FunctionalInterface

- If you explore the new Java API you'll notice that functional interfaces are annotated with `@FunctionalInterface`
  - Used to indicate the interface is intended to be a functional interface
  - So the compiler will return a meaningful error if it is not
  - This annotation is not mandatory but it is good practice (such as `@Override`)

# Primitive specializations

- Every type in Java is either
  - A primitive type (int, double, ...)
  - A reference type (Integer, Double, ...)
- Since Java 5 boxing/unboxing can be done implicitly
- Java 8 brings a specialized version of the functional interfaces on order to avoid auto-boxing operations when the inputs or outputs are primitives
  - E.g. IntPredicate avoids the boxing that would be done by using Predicate<Integer>

# What about exceptions?

- None of the functional interfaces presented before allow for a **checked exception** to be thrown
- You have to options:
  - Define your own functional interface
  - Wrap the lambda body with a try/catch block

# Method references

- A shorthand for lambdas calling only a specific method

Lambda	Method reference
<code>(Apple a) -&gt; a.getWeight()</code>	<code>Apple::getWeight</code>
<code>() -&gt; Thread.currentThread().dumpStack()</code>	<code>Thread.currentThread()::dumpStack()</code>
<code>(str, i) -&gt; str.substring(i)</code>	<code>String::substring</code>
<code>(String s) -&gt; System.out.println(s)</code>	<code>System.out::println</code>

# Constructor references

- A shorthand for lambdas creating a new instance in the body

Lambda	Functional interface	Constructor reference
<code>() -&gt; new Apple()</code>	<code>Supplier&lt;Apple&gt;</code>	<code>Apple::new</code>
<code>(weight) -&gt; new Apple(weight);</code>	<code>Function&lt;Integer, Apple&gt;</code>	<code>Apple::new</code>
<code>(str, i) -&gt; new Apple(str, i);</code>	<code>BiFunction&lt;String, Integer, Apple&gt;</code>	<code>Apple::new</code>



# STREAMS

# Streams

- Collections is the most used Java API
  - But its operations are “low level” (e.g. they lack the expressiveness of database-like operations)
  - Difficult to leverage multicore
- Streams are an update to the Java API that
  - Let's you manipulate data in a declarative way
  - And that can be processed in parallel transparently

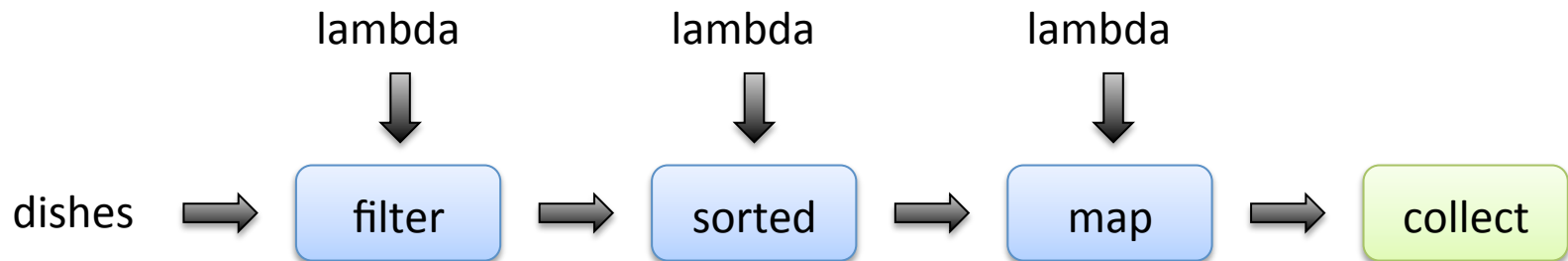
# Before (Java 7)

```
public static List<String> getLowCaloricDishesNamesInJava7(List<Dish> dishes){  
    List<Dish> lowCaloricDishes = new ArrayList<>();  
    for(Dish d: dishes){  
        if(d.getCalories() < 400){  
            lowCaloricDishes.add(d);  
        }  
    }  
    Collections.sort(lowCaloricDishes, new Comparator<Dish>() {  
        public int compare(Dish d1, Dish d2){  
            return Integer.compare(d1.getCalories(), d2.getCalories());  
        }  
    });  
    List<String> lowCaloricDishesName = new ArrayList<>();  
    for(Dish d: lowCaloricDishes){  
        lowCaloricDishesName.add(d.getName());  
    }  
    return lowCaloricDishesName;  
}
```

# Now (Java 8)

```
import static java.util.Comparator.comparing;  
import static java.util.stream.Collectors.toList;
```

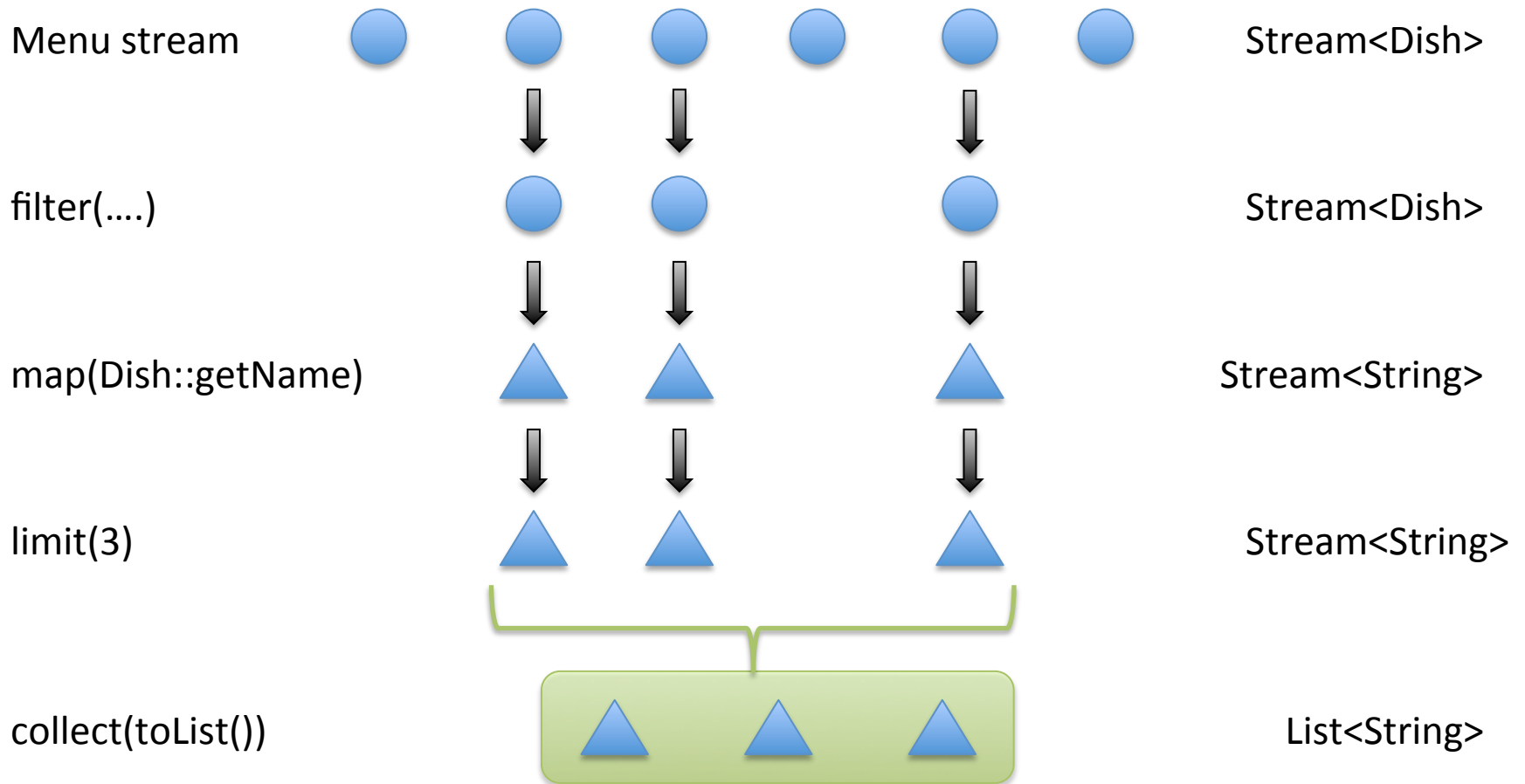
```
public static List<String> getLowCaloricDishesNamesInJava8(List<Dish> dishes){  
    return dishes.stream()  
        .filter(d -> d.getCalories() < 400)  
        .sorted(comparing(Dish::getCalories))  
        .map(Dish::getName)  
        .collect(toList());  
}
```



# What is a stream?

- A sequence of elements from a source that supports data processing operations
  - Sequence of elements
    - Collections are about data
    - Streams are about computations
  - Source
    - Streams consume data from a providing source
  - Data processing operations
    - Database-like & Functional programming
    - Pipelining & Internal iteration

# Visualizing stream processing



# Streams from files

- We can create a Stream from the contents of a file
  - Files.lines is the Stream<String> of the lines of the file

```
long uniqueWords = Files.lines(Paths.get("lambdasinaction/chap5/data.txt"),  
                                Charset.defaultCharset())  
    .flatMap(line -> Arrays.stream(line.split(" ")))  
    .distinct()  
    .count();
```

# Better separation of concerns

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```

```
List<String> errors =
    Files.lines(Paths.get(fileName))
        .filter(l->l.startsWith("ERROR"))
        .limit(40)
        .collect(toList());
```



# Grouping transactions by currency

```
private static void groupImperatively() {  
    Map<Currency, List<Transaction>> transactionsByCurrencies = new HashMap<>();  
    for (Transaction transaction : transactions) {  
        Currency currency = transaction.getCurrency();  
        List<Transaction> transactionsForCurrency =  
            transactionsByCurrencies.get(currency);  
        if (transactionsForCurrency == null) {  
            transactionsForCurrency = new ArrayList<>();  
            transactionsByCurrencies.put(currency, transactionsForCurrency);  
        }  
        transactionsForCurrency.add(transaction);  
    }  
    System.out.println(transactionsByCurrencies);  
}
```

# Grouping transactions by currency

```
private static void groupFunctionally() {  
    Map<Currency, List<Transaction>> transactionsByCurrencies =  
        transactions.stream()  
            .collect(groupingBy(Transaction::getCurrency));  
    System.out.println(transactionsByCurrencies);  
}
```

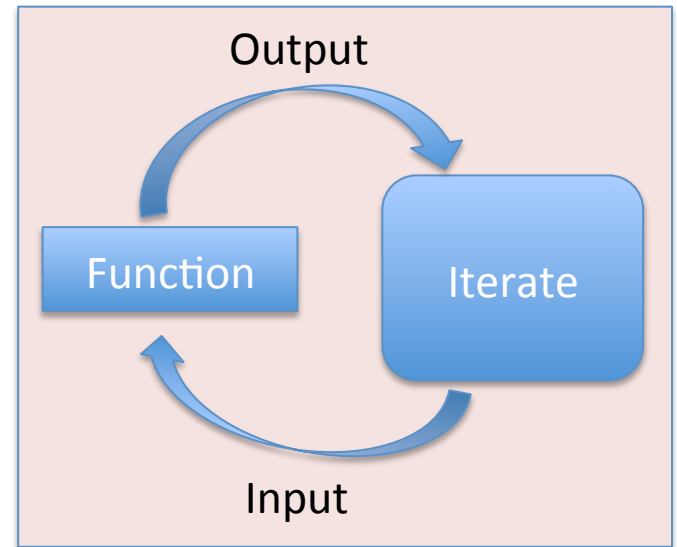
# Parallel streams

- You can turn a collection into a parallel stream by invoking the method `parallelStream` on it
- A parallel stream is a stream that splits its elements into multiple chunks, processing each chunk with a separate thread
  - Uses Fork/Join framework
  - `SplitIterator`
- But
  - Parallelization is not free
  - Not all collections or operations perform equally
  - Side effects are bad !!!!

# Parallel streams

```
public static long parallelSum(long n) {  
    return Stream.iterate(1L, i -> i + 1)  
        .limit(n)  
        .parallel()  
        .reduce(0L, Long::sum);  
}
```

```
public static long parallelRangedSum(long n) {  
    return LongStream.rangeClosed(1, n)  
        .parallel()  
        .reduce(0L, Long::sum);  
}
```



# Parallel streams

Source	Decomposability
ArrayList	Excellent
LinkedList	Poor
IntStream.range	Excellent
Stream.iterate	Poor
HashSet	Good
TreeSet	Good

# DEFAULT METHODS

# The problem with interfaces

- Traditionally a Java interface groups related methods together into a contract
- Any class that implements the interface must provide implementation for all of them
- This causes a problem when designers need to update the interface adding new methods
  - ALL IMPLEMENTATIONS GET BROKEN !!!!

# Default methods

- Interfaces in Java 8 can now declare methods **and implement them**
  - First, it allows the definition of **static methods**
  - Second, introduces a new feature called **default methods** that provides a default implementation for methods in an interface
  - So, existing classes implementing an interface will automatically inherit this default implementation if they do not provide one
- So interfaces now can evolve non-intrusively !!!



# Default methods

- For instance, the `removeIf` method was added to the `Collection` interface as follows:

```
default boolean removeIf(Predicate<? super E> filter) {  
    Objects.requireNonNull(filter);  
    boolean removed = false;  
    final Iterator<E> each = iterator();  
    while (each.hasNext()) {  
        if (filter.test(each.next())) {  
            each.remove();  
            removed = true;  
        }  
    }  
    return removed;  
}
```

# Static methods

- A common pattern in Java is to both define an interface and a utility companion class with many abstract methods
  - Collection: defines the interface
  - Collections: defines static utility methods
- Now that we can define static methods in interfaces we can move these utility methods in the interface

# Abstract classes vs. Interfaces

- Now that Interfaces can implement methods, what is the difference with Abstract Classes?
  - A class can extend only from one abstract class, but a class can implement multiple interfaces.
  - An abstract class can enforce a common state through instance variables (fields), but interfaces cannot have instance variables

# Optional methods

- One possible use for default methods is to define optional methods in an interface
- For instance, in the Iterator interface the method remove is optional

```
default void remove() {  
    throw new UnsupportedOperationException("remove");  
}
```

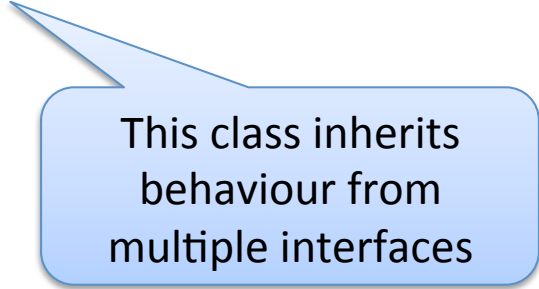
# Multiple inheritance of behaviour

```
public interface Rotatable {  
    void setRotationAngle(int angle);  
    int getRotationAngle();  
    default void rotateBy(int angle) {  
        setRotationAngle((getRotationAngle() + angle) % 360);  
    }  
}  
  
public interface Movable {  
    int getX();  
    int getY();  
    void setX(int x);  
    void setY(int y);  
    default void moveHorizontally(int distance) {  
        setX(getX() + distance);  
    }  
    default void moveVertically(int distance) {  
        setY(getY() + distance);  
    }  
}
```

# Multiple inheritance of behaviour

```
public interface Resizable {  
    int getWidth();  
    int getHeight();  
    void setWidth(int width);  
    void setHeight(int height);  
    void setAbsoluteSize(int width, int height);  
    default void setRelativeSize(int wFactor, int hFactor) {  
        setAbsoluteSize(getWidth() * wFactor, getHeight() * hFactor);  
    }  
}
```

```
public class Monster implements Rotatable, Movable, Resizable {  
    ...  
}
```



This class inherits  
behaviour from  
multiple interfaces

# Resolution rules

1. Classes always win. A method declaration in a class or a superclass takes priority over any default method implementation
2. Otherwise, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected
3. Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly

# OPTIONAL



# “my billion dollar mistake”

- Tony Hoare introduced null references in 1965 while designing ALGOL W
  - despite his goal that all use of references could be absolutely safe
  - he decided to make an exception for null references
  - because he thought this was the most convenient way to represent absence of value
- After many years he regretted this decision calling it “my billion dollar mistake”

# Modelling the absence of value

```
public class Person {  
    private Car car;  
    public Car getCar() { return car; }  
    ...  
}  
public class Car {  
    private Insurance insurance;  
    public Insurance getInsurance() { return insurance; }  
    ...  
}  
public class Insurance {  
    private String name;  
    public String getName() { return name; }  
    ...  
}
```

NullPointerException !!!

```
public String getCarInsuranceName(Person person) {  
    return person.getCar().getInsurance().getName();  
}
```

# Modelling the absence of value

```
public String getCarInsuranceName(Person person) {  
    if (person != null) {  
        Car car = person.getCar();  
        if (car != null) {  
            Insurance insurance = car.getInsurance();  
            if (insurance != null) {  
                return insurance.getName();  
            }  
        }  
    }  
    return "Unknown";  
}
```

**Each `null` check increases the nesting level of the remaining part of the invocation chain**

# Modelling the absence of value

```
public String getCarInsuranceName(Person person) {  
    if (person == null) {  
        return "Unknown";  
    }  
    Car car = person.getCar();  
    if (car == null) {  
        return "Unknown";  
    }  
    Insurance insurance = car.getInsurance();  
    if (insurance == null) {  
        return "Unknown";  
    }  
    return insurance.getName();  
}
```

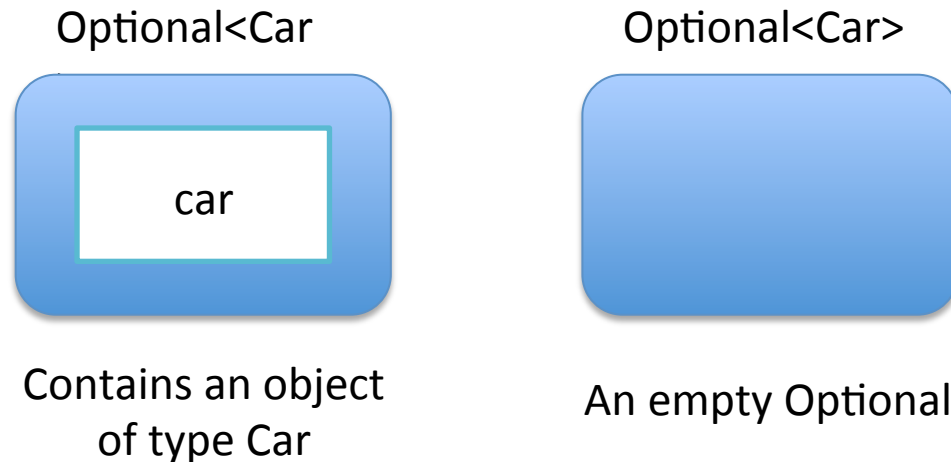
Each null check adds a further exit point.

# Problems with null

- It's a source of error
- It bloats your code
- It's meaningless
- It breaks Java philosophy
- It creates a hole in the typesystem

# Java.util.Optional<T>

- Inspired by ideas in Haskell and Scala
  - The Maybe Monad
- It's a class that encapsulates an optional value



# Modelling with Optional<T>

```
public class Person {  
    private Optional<Car> car;  
    public Optional<Car> getCar() {  
        return car;  
    }  
}
```

A person might or  
might not own a  
car

```
public class Car {  
    private Optional<Insurance> insurance;  
    public Optional<Insurance> getInsurance() {  
        return insurance;  
    }  
}
```

A car might or  
might not have an  
insurance

```
public class Insurance {  
    private String name;  
    public String getName() {  
        return name;  
    }  
}
```

An insurance company  
must have a name

# Creating Optional objects

- Empty optional

```
optCar = Optional.empty();
```

- Optional from “non-null value”

```
optCar = Optional.of(car);
```

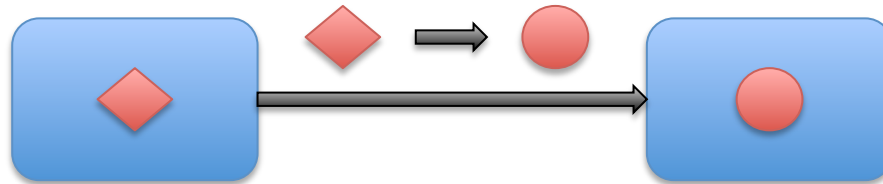
- Option from “null value”

```
optCar = Optional.ofNullable(car);
```



# Transforming values with map

```
String name = null;  
if (insurance != null) {  
    name = insurance.getName();  
}
```



```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);  
Optional<String> name = optInsurance.map(Insurance::getName);
```

# Chaining transformations

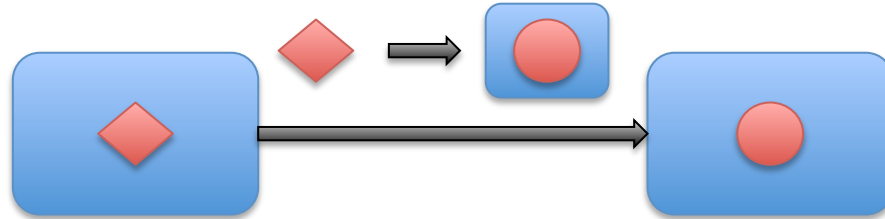
```
public String getCarInsuranceName(Person person) {  
    return person.getCar().getInsurance().getName();  
}
```

```
Optional<Person> optPerson = Optional.of(person);  
Optional<String> optName =  
    optPerson.map(Person::getCar)  
              .map(Car::getInsurance)  
              .map(Insurance::getName);
```

This does not compile !!!



# Chaining transformations with flatMap



```
public String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(Person::getCar)  
                .flatMap(Car::getInsurance)  
                .map(Insurance::getName)  
                .orElse("Unknown");  
}
```

# Correct way of modelling

- The purpose of Optional is to support the optional-return idiom only
- It wasn't intended to be used as a field
- That's why it is not Serializable
- If you need serialization

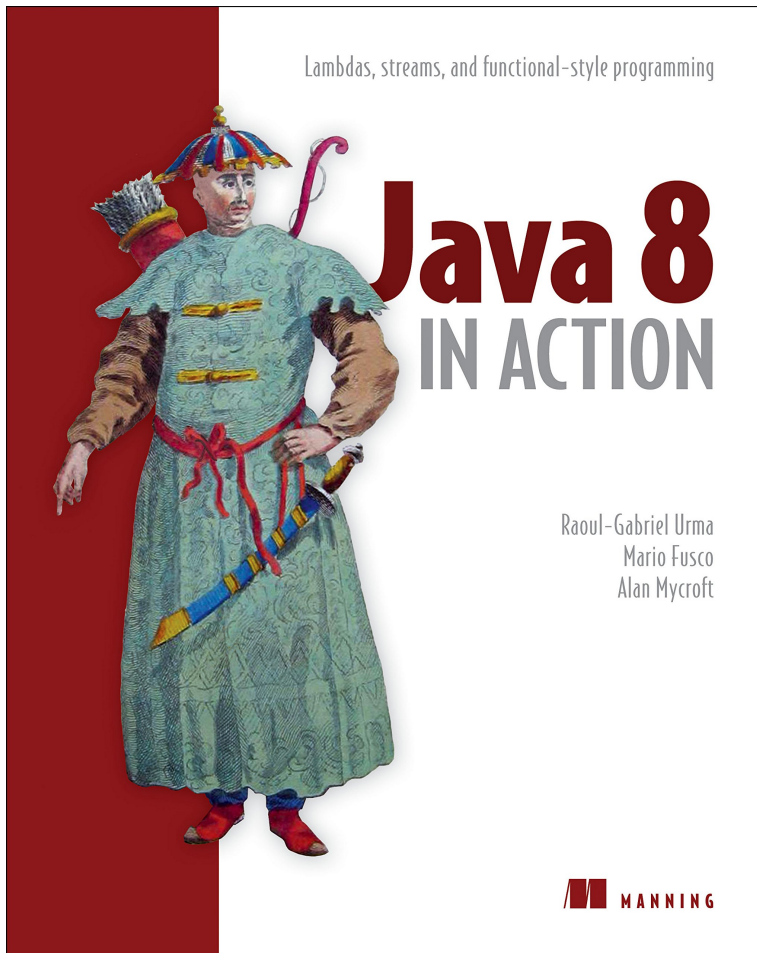
```
public class Person {  
    private Car car;  
    public Optional<Car> getCar() {  
        return Optional.ofNullable(car);  
    }  
}
```

# CONCLUSIONS

# Conclusions

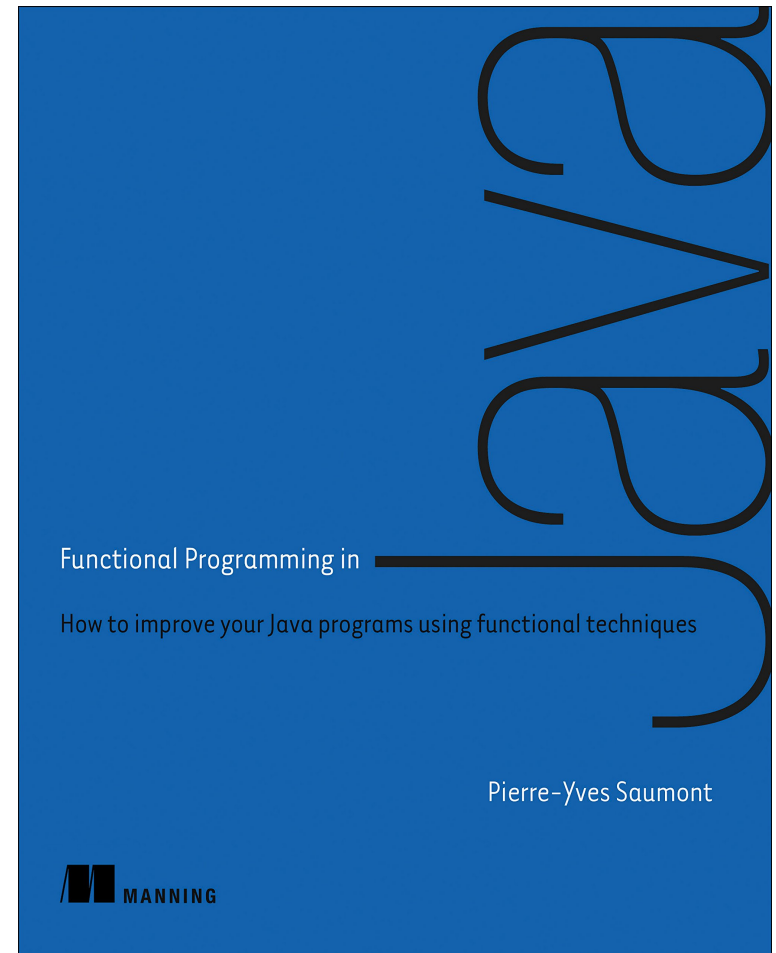
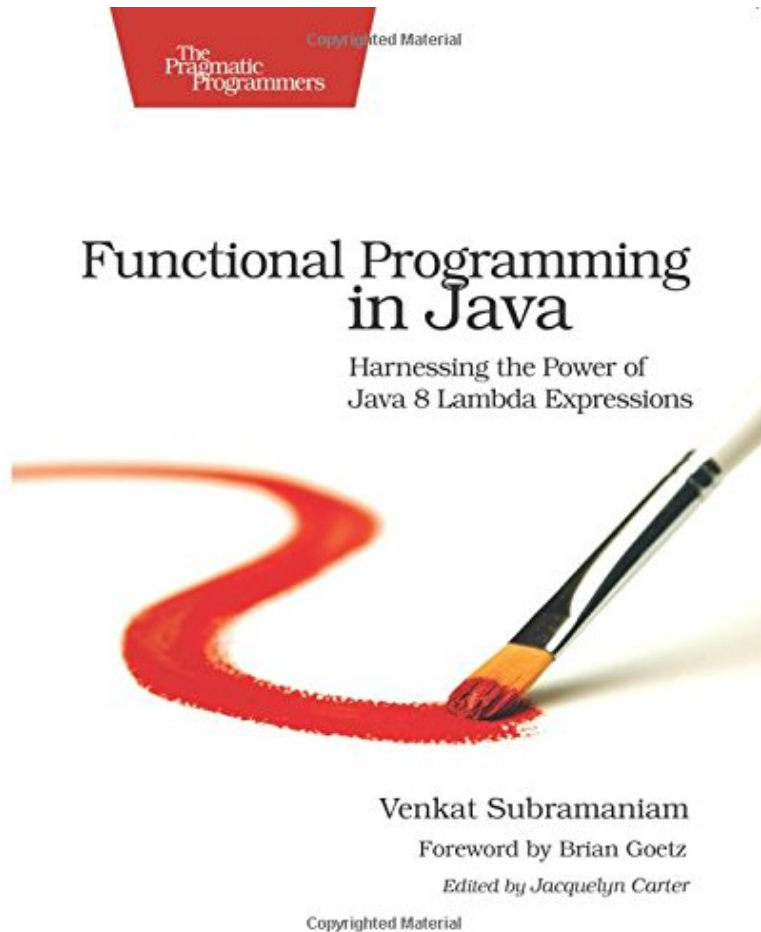
- Java 8 forces us to think different
  - More declarative, less imperative
- We must learn functional programming as the foundation for new idioms and practices
  - Better in a more functional language (such as Haskell, Scala, Clojure, ...)
- Java is here to stay so it will continue to evolve
  - **Java 9**: modularization
  - **Java 10**: value classes ?

# Bibliography



- Code samples
  - <https://github.com/java8/Java8InAction>
- Mario Fusco's slides
  - <http://www.slideshare.net/mariofusco/presentations>

# Bibliography





# Bibliography

