

Java 8

Playtime

Agenda

- Java 8 Why should you care?
- Passing code
- Lambda expressions
- Processing data with streams
- Optional: a better alternative to null

Passing code

To put it in many words

- Behavior parameterization is the ability for a method to take multiple different strategies as parameter and use them internally to accomplish different behaviors
- Behavior parameterization lets you make your code more adaptive to changing requirements and saves engineering efforts in the future
- The Java API contains many methods that can be parameterized with behavior, which include sorting, threads, and GUI handling

It is not totally new in Java

- Passing code is a way to give new behaviors as argument to a method.
- But it is currently verbose in Java
- Anonymous classes help a bit to get rid of the verbosity associated with declaring multiple concrete classes for an interface that is needed only once

Time for an example

- no matter what you do, user requirements will change
- Imagine an application that will help a farmer to examine his stock of apples
 - On day 1 the farmer is interested in GREEN one's
 - On day 2 he's interested in apples > 150 gram
 - On day 3 apples must be GREEN and > 150 gram
- How to cope with these changing requirements?

About our solution

- Minimising engineering effort is preferred
- Similar functionality should
 - Straightforward to implement
 - and maintainable in the long term
- Use the Abstract over behaviour pattern
 - take a block of code and make it available without executing it

Coping with changing requirements

- The requirement on day 1: Find all green apples

```
public static List<Apple> findGreenApples(List<Apple> apples) {  
    List<Apple> listOfApples=new ArrayList<>();  
    for (Apple apple : apples) {  
        if("green".equalsIgnoreCase(apple.getCollor())) {  
            listOfApples.add(apple);  
        }  
    }  
    return listOfApples;  
}
```


Coping with changing requirements

- The requirement on day 2: Find all red apples
- We could of course:

```
public static List<Apple> findRedApples(List<Apple> apples) {  
    List<Apple> listOfApples=new ArrayList<>();  
    for (Apple apple : apples) {  
        if("red".equalsIgnoreCase(apple.getCollor())) {  
            listOfApples.add(apple);  
        }  
    }  
    return listOfApples;  
}
```

- This strategy doesn't cope well with future new color wishes. A good strategy:
- After writing similar code, try to abstract

Coping with changing requirements

- Abstract away color changes with a parameter

```
public static List<Apple> findGreenApples(List<Apple> apples, String color) {  
    List<Apple> listOfApples=new ArrayList<>();  
    for (Apple apple : apples) {  
        if(color.equalsIgnoreCase(apple.getCollor())) {  
            listOfApples.add(apple);  
        }  
    }  
    return listOfApples;  
}
```

- But what to do when a new requirement appears which says: look for apples over 150 grams

Coping with changing requirements

■ A solution would be:

```
public static List<Apple> findApplesByWeight(List<Apple> apples,
                                             int minimumWeight) {

    List<Apple> listOfApples=new ArrayList<>();
    for (Apple apple : apples) {
        if(apple.getWeight()>minimumWeight) {
            listOfApples.add(apple);
        }
    }

    return listOfApples;
}
```

■ This is a good solution but it isn't DRY

- Most of the logic in the 3 find methods is similar:
 - Iterating over a collection
 - Applying a filter criteria

■ Potentially 3 spots to change your code

Where are we

- So far the find methods are parameterised with String or int value parameters
- In more complicated scenario's we could add flags to enable a finder to filter on more then one criteria
- This approach has its limits
- We need a better way to tell the appleFinder method how to filter the apples

Behaviour parameterisation

- a better way than adding lots of parameters to cope with changing requirements
- find a better level of abstraction
- One solution is:
 - model our selection criterion (or strategy)
 - you are working with apples
 - returning a boolean based on some attributes of Apple
 - We call this a predicate (that is, a function that returns a boolean).

Define an interface

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

```
public class AppleGreenPredicate implements ApplePredicate {  
    @Override  
    public boolean test(Apple apple) {  
        return "green".equalsIgnoreCase(apple.getCollor());  
    }  
}
```

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

- But how to use these actual implementations of ApplePredicates?

Behaviour parameterisation

- Our filter methods need to accept a parameter with a reference to these predicates to test a condition
- This is what behaviour parameterisation means:
 - the ability to tell a method to take multiple different strategies as parameters and use them internally to accomplish different behaviours
- Change the signature of our find method again

Behaviour parameterisation

```
public static List<Apple> findApplesByWeight(List<Apple> apples,
                                             ApplePredicate predicate) {
    List<Apple> listOfApples=new ArrayList<>();
    for (Apple apple : apples) {
        if(predicate.test(apple)) {
            listOfApples.add(apple);
        }
    }
    return listOfApples;
}
```

■ engineering benefit

- separation of the logic of iterating the collection inside the filterApples method
- with the behaviour you want to apply to each element of the collection

Prior to Java 8

- Unfortunately, because the `filterApples` method can only take objects
- The code must be wrapped inside an `ApplePredicate` object.
- This is similar to “passing code” inline, because you’re passing a boolean expression through an object that implements the test method

In code:

```
List<Apple> redApples = findApplesByWeight(stock, new AppleRedPredicate());  
for (Apple apple : redApples) {  
    System.out.println(apple);  
}
```

```
public class AppleRedPredicate implements ApplePredicate {  
  
    @Override  
    public boolean test(Apple apple) {  
        return "red".equalsIgnoreCase(apple.getCollor());  
    }  
  
}
```

Exercise

- Write a `forEach` method that expects a list of apples and a second argument that can be used to pass in an arbitrary behaviour
 - i.e. a method that zero's all the weights of the apples
 - or a method that prints all the apples
 - Use the starter code:

```
public static void forEachApple(List<Apple> apples, ?) {  
    for (Apple apple : apples) {  
        ?  
    }  
}
```

Solution

```
public interface Action<T> {  
    public void doAction(T something);  
}
```

```
public class PrintAction implements Action<Apple> {  
    @Override  
    public void doAction(Apple apple) {  
        System.out.println("{apple:color=" + apple.getCollor() +  
            ">::apple:weight=" + apple.getWeight() + "}" );  
    }  
}
```

```
public static void forEachApple(List<Apple> apples, Action<Apple> action) {  
    for (Apple apple : apples) {  
        action.doAction(apple);  
    }  
}
```

Tackling verbosity

- When the `forEachApple` method is invoked an instance of `PrintAction` must be supplied

```
forEachApple(stock, new PrintAction());
```

- How to make this one time usage of a class less verbose → use an Anonymous class!

```
forEachApple(stock, new Action<Apple>() {  
    @Override  
    public void doAction(Apple apple) {  
        System.out.println("Print apple" + apple);  
    }  
});
```

Anonymous class still not optimal

```
forEachApple(stock, new Action<Apple>() {  
    @Override  
    public void doAction(Apple apple) {  
        System.out.println("Print apple" + apple);  
    }  
});
```

- Still a lot of boilerplate code to express a simple action
- Also anonymous classes introduce some easy overlooked complexity

What is printed by this code?

```
public class MeaningOfThis {  
  
    public final int value = 4;  
  
    public void doIt() {  
        int value = 6;  
        Runnable r = new Runnable() {  
            public final int value = 5;  
  
            public void run() {  
                int value = 10;  
                System.out.println(this.value);  
            }  
        };  
        r.run();  
    }  
  
    public static void main(String...args){  
        MeaningOfThis m = new MeaningOfThis();  
        m.doIt();  
    }  
}
```

Preview of lambda expressions

```
forEachApple(stock, (Apple apple)->System.out.println("Print " + apple));
```

■ Read:

- (Apple apple)->System.out.println("Print " + apple)
- An anonymous function expects an argument of type Apple name apple
- The -> describes the body of the function, what should be done with this apple
- In the next chapter this is further explored

Preview of lambda expressions

```
forEachApple(stock, (Apple apple)->System.out.println("Print " + apple));
```

- This is much better
- The code reads more like the problem statement:
- take an apple and print it

List type can also be abstracted

```
public static <T> void forEachApple(List<T> ts, Action<T> action) {  
    for (T t : ts) {  
        action.doAction(t);  
    }  
}
```

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9);  
forEachApple(numbers, (Integer i)-> System.out.println("Print number " + i));
```

- Arbitrarily lists of something can be processed by the “doAction” method
- Flexibility and conciseness are nicely made available in Java 8

Resume

- behaviour parameterisation is a useful pattern to easily adapt to changing requirements
- this pattern lets you encapsulate an arbitrary strategy and parameterise the behaviour of methods
- similar to the strategy design pattern from the Gang of Four
- many methods existing in the Java API can be parameterised with different behaviours

Examples from the java api

- Sorting with a comparator
- Executing a block of code with Runnable
- GUI event handling

Lambda expressions

Finally
Chapter 3

As a starter

- The passing code pattern is useful for coping with frequent requirement changes in your code
- a block of code can be passed to another method
- the behaviour of this other method depends on and uses the code passed to it.

Lambda's in a nutshell

- a lambda expression can be understood as a kind of 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

More about Lambda's

- anonymous because it doesn't have an explicit name like a method would normally have
- called a function because a lambda isn't associated with a particular class like a method is
- but it has all the characteristics of a method
- a lambda expression can be passed as argument to a method.
- concise: no need to write a lot of boilerplate like you do for anonymous classes

As an example

■ Without lambda's

Boilerplate code!

```
stock.sort(  
    int compare(Apple apple1, Apple apple2) {  
        int result= apple1.getWeight()-apple2.getWeight();  
        return result ;  
    }  
);
```

As an example

■ Without lambda's

```
stock.sort(
```

```
    [redacted] [redacted] (Apple apple1, Apple apple2) {
```

```
        int result= apple1.getWeight()-apple2.getWeight();  
        return result ;
```

```
    }
```

```
});
```

return type can be inferred

Anonymous function

As an example

■ With lambda's

Introduction → operator to signify the lambda expression

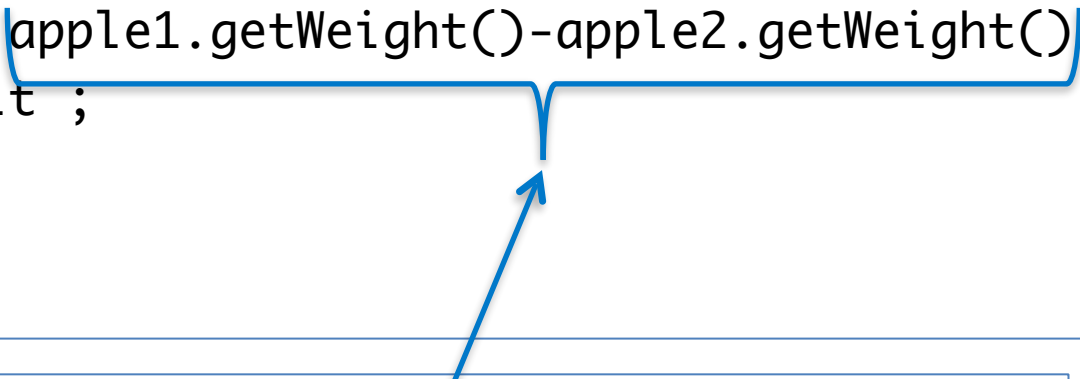


```
stock.sort(  
    (Apple apple1, Apple apple2) → {  
        int result= apple1.getWeight()-apple2.getWeight();  
        return result ;  
    }  
);
```

As an example

■ With lambda's

```
stock.sort(  
  
    (Apple apple1, Apple apple2) -> {  
  
        int result= apple1.getWeight()-apple2.getWeight();  
        return result ;  
    }  
);
```



This is an expression; the value of it is implicitly returned when it is the only statement within the body. As an extra bonus the {,} and ; can be omitted

In a concise form

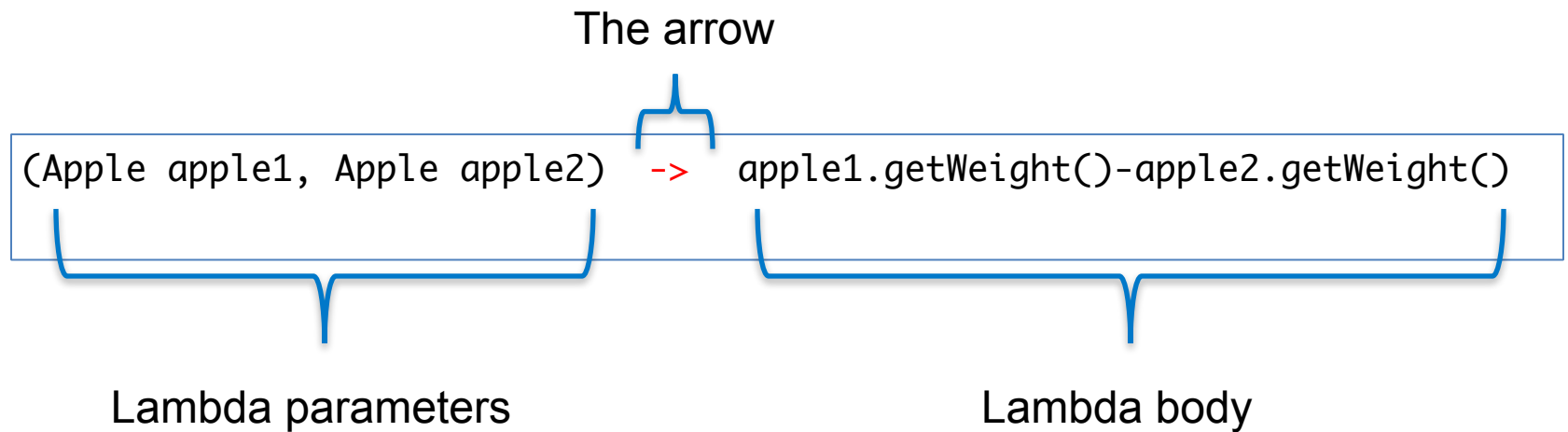
■ With lambda's

```
stock.sort(  
    (Apple apple1, Apple apple2) ->  
    apple1.getWeight()-apple2.getWeight()  
);
```

■ Written on one line

```
stock.sort((Apple apple1, Apple apple2) -> apple1.getWeight()-apple2.getWeight());
```

Composition of Lambda expression



Some Lambda expression examples

- `(String s) -> s.length()`
- `(Apple a) -> a.getWeight() > 150`
- `(int x, int y) -> { System.out.println(x+y); }`
 - The first expects a String argument and implicitly the length is returned
 - The second expects an parameter of type Apple and returns a boolean
 - The last expects two int parameters and returns void

Some Lambda expression examples

- `() -> 42`
- `(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`
 - The first lambda expression has no arguments and returns the int 42
 - The last expects two Apple parameters and returns an int

Basic syntax of lambda expression

(parameters) -> expression

OR

(parameters) -> { statements; }

Quiz which are valid lambda expressions?

- `() -> {}`
- `() -> "Raoul"`
- `() -> {return "Mario";}`
- `(Integer i) -> return "Alan" + i;`
- `(String s) -> {"Iron Man";}`

Where and how to use lambdas?

- Where are lambda expressions allowed?
- Java allows us to use lambdas where a type is expected
- That brings up the question?
 - What is the type of lambda expressions?

A functional interface

- A functional interface is an interface that declares exactly one abstract method
- Later on we will see that from java 8 on, interface can also define so called “default methods” , but that is for later
- i.e. Comparator, Runnable

```
public interface Comparator<T> {  
    public int compare(T o1, T o2);  
}
```

```
public interface Runnable{  
    public void run();  
}
```

@FunctionalInterface

- In the new Java 8 API the @FunctionalInterface is used to indicate that the interface is a functional one
- The compiler gives a warning when an interface is annotated with @FunctionalInterface but, for example, declares more than one abstract method

Quiz, which interface is functional?

```
public interface Adder{  
    public int add(int a, int b);  
}  
  
public interface SmartAdder extends Adder{  
    public int add(double a, double b);  
}  
  
public interface Nothing{  
}
```

Why functional interfaces?

- Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline
- the whole expression is treated as an instance of a functional interface.

Example with Runnable

■ Runnable is a functionalInterface

```
public class LambdaTest {  
  
    Runnable r1 = () -> System.out.println("Hello World 1");  
  
    Runnable r2 = new Runnable(){  
        public void run(){  
            System.out.println("Hello World 2");  
        }  
    };  
  
    public void process(Runnable r){  
        r.run();  
    }  
  
    @Test  
    public void runRuns() {  
        process(r1);  
        process(r2);  
        process(() -> System.out.println("Hello World 3"));  
    }  
}
```


Function descriptor

- The signature of the abstract method of the functional Interface describes the signature of the lambda expression
- This is called a function descriptor
- i.e. The Runnable interface describes the signature of a function that doesn't has a parameter and returns void
- Neither name of interface or method matter
- Actually () -> void is enough

What are valid usages of lambdas?

```
public class WhichAreValidExpressions {  
  
    public void execute(Runnable r){  
        r.run();  
    }  
  
    public Callable<String> fetch() {  
        return () -> "Tricky example ;-);  
    }  
    @Test  
    public void testLambdaExpressions() {  
        //1  
        execute(() -> {});  
        //2  
        fetch();  
        //3  
        ApplePredicate p = (Apple a) -> a.getWeight();  
    }  
}
```

Putting lambdas in practice

- A recurrent pattern in resource processing is
 - to open a resource
 - do some processing on it
 - and then close the resource
- The setup and cleanup phases are always similar and appear around the important code doing the processing
- This is called the execute around pattern

An example

```
public class ExecuteAroundTest {

    @Test
    public void test() throws IOException {
        processFile();
    }

    public static String processFile() throws IOException {
        try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
            String oneLineFromFile= br.readLine();
            System.out.println(oneLineFromFile);
            return oneLineFromFile;
        }
    }
}
```

- The current code is limited, only one line can be read
- What If 2 lines or something else is needed?

Make code more general

- Reuse code to do startup and cleanup and tell the `processFile` method what to do with the file
- We need to parameterize the behaviour of the `processFile` method
- How to supply this behaviour → lambdas
- Basically a lambda that needs a `BufferedReader` and returns a `String`, suffices

To use lambdas we need ...

- A functional interface with a signature:
 - (BufferedReader br) -> String

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    public String process(BufferedReader b) throws IOException;
}
```

- use this interface as the argument to your new processFile method

```
public static String processFile(BufferedReaderProcessor p) throws IOException {
    ...}
}
```

executing a behavior

- Lambda expressions of the form:
- (BufferedReader br) -> String can now be passed
 - Now, execute the behaviour

```
public static String processFile(BufferedReaderProcessor p) throws  
    IOException {  
    try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {  
        return p.process(br);  
    }  
}
```

Passing lambdas

```
@Test
public void testLambdaReturnOneLine() throws IOException {
    String output = processFile(
        (BufferedReader br) -> br.readLine()
    );
    System.out.println(output);
}
```

```
@Test
public void testLambdaReturningTwoLines() throws IOException {
    String output = processFile(
        (BufferedReader br) -> br.readLine() + br.readLine()
    );
    System.out.println(output);
}
```


Exercise

- Create a functional interface
 - The signature of the function accepts a `BufferedReader` and returns a `String`
- Create a static method that accepts a reference to this interface
 - In the body of this static method a `BufferedReader` is instantiated and coupled to an ad hoc File which you created and filled with test data
 - The processing of the file is specified in an anonymous function

Exercise continued

- The anonymous function should return a String which is printed to the console
- Make different Implementation for the anonymous function in different Unit tests
- A function that returns as a String:
 1. The first line
 2. The last line
 3. The longest line
 4. The longest word
 5. Try to implement functionality you think fit

New Functional interfaces in Java8!

- There are already several functional interfaces available in the Java API such as Comparable, Runnable and Callable
- New in Java 8 a whole family of interfaces
 - Inside the `java.util.function` package
- Among others: Predicate, Consumer, Function

The Predicate interface

```
@FunctionalInterface
public interface Predicate<T>{
    public boolean test(T t);
}
```

- Interface can be used when you need a boolean expression using an object of type T

```
public class UsingPredicateTest {

    public static List<String> filter(List<String> list, Predicate<String> p) {
        List<String> results = new ArrayList<>();
        for(String s: list){
            if(p.test(s)){
                results.add(s);
            }
        }
        return results;
    }

    @Test
    public void test() {
        List<String> listOfStrings = Arrays.asList("a", "ab", "", "abc");
        Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
        List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
    }
}
```

Consumer, what is it's function?

```
@FunctionalInterface
public interface Consumer<T>{
    public void accept(T t);
}
```

```
public class ConsumerTest {

    public static void forEach(List<Integer> list, Consumer<Integer> c) {
        for (Integer i : list) {
            c.accept(i);
        }
    }

    @Test
    public void printIntegers() {
        forEach(Arrays.asList(1, 2, 3, 4, 5),
            (Integer i) -> System.out.println(i));
    }
}
```

use this interface when you need to access an object of type T and perform some operations on it

Function, what can you do with it?

```
public interface Function<T, R> {  
    public R apply(T t);  
}
```

```
public class TestFunctionInterface {  
  
    public static List<Integer> map(List<String> list, Function<String, Integer> f) {  
        List<Integer> result = new ArrayList<>();  
        for (String s : list) {  
            result.add(f.apply(s));  
        }  
        return result;  
    }  
    @Test  
    public void mapStringsToIntAccordingToTheirLength() {  
        List<Integer> lengths  
            = map(Arrays.asList("lambdas", "in", "action"), (String s) -> s.length());  
  
        for(int length:lengths) {  
            System.out.println(length);  
        }  
    }  
}
```

Predefined Functional interface

New in Java 8	In java.util.function
<i>Consumer<T></i>	<i>Represents an operation that accepts a single input argument and returns no result</i>
<i>Function<T,R></i>	<i>Represents a function that accepts one argument of type T and produces a result of R</i>
<i>Predicate<T></i>	<i>Represents a predicate (boolean-valued function) of one argument</i>
<i>Supplier<R></i>	<i>Represents a supplier of results R</i>
<i>UnaryOperator<T></i>	<i>Represents an operation on a single operand that produces a result of the same type T</i>
<i>BinaryOperator<T></i>	<i>Represents an operation upon two operands of the same type T, producing a result of T</i>
<i>And a lot of specialisations</i>	<i>BiConsumer<T,U>, BiFunction<T,U,R>, BiPredicate<T,U>, DoubleFunction<R>,</i>

Quiz:

■ Which functional interface would you use?

1. $T \rightarrow R$
2. $(\text{int}, \text{int}) \rightarrow \text{int}$
3. $T \rightarrow \text{void}$
4. $() \rightarrow T$
5. $(T, U) \rightarrow R$

Answers:

■ Which functional interface would you use?

1. $T \rightarrow R$

- `Function<T,R>` typical usage is converting of type `T` into something of type `R`

2. $(\text{int}, \text{int}) \rightarrow \text{int}$

- `int IntBinaryOperator` has one method `applyAsInt`

3. $T \rightarrow \text{void}$

- `Consumer<T>` has one method `accept()`

4. $() \rightarrow T$

- `Supplier<T>` has a single method `get()`

5. $(T,U) \rightarrow R$ See 1 but then `BiFunction<T,U,R>`

What is printed by this code?

```
@Test
public void meaningOfThis() {
    Runnable r = new Runnable() {
        public void run() {
            System.out.println(this.getClass().getName());
            System.out.println(OuterClass.this.getClass().getName());
        }
    };
    r.run();
}
```

//Compare output with:

```
@Test
public void meaningOfThis() {
    Runnable r = () -> {
        System.out.println(this.getClass().getName());
        System.out.println(OuterClass.this.getClass().getName());
    };
    r.run();
}
```

Type checking, inference and restrictions

It's like climbing a steep
hill

Type of a Lambda

- Lambda expressions enable the generation of an instance of a functional interface
- But a lambda expression doesn't contain the information itself about which functional interface it's implementing
- So what is the actual type of a lambda?

Typechecking

- The type of a lambda is deduced from context
 - assignment
 - method invocation
 - cast expression
 - and so on
- The same lambda expression can be associated with different functional interfaces
 - Example: both PrivilegedAction and Callable expect a function that expects nothing and returns a generic type T

An example

```
@Test
public void test() {
    //1.
    Callable<Integer> c = () -> 42;
    //2.
    PrivilegedAction<Integer> p = () -> 42;
}
```

- The type expected inside a context for the lambda expression is called the target type
- In 1. the target type is `Callable<Integer>`
- In 2. target type is `PrivilegedAction<Integer>`

How does type checking occur?

- Take as an example:

- `Callable<Integer> c = () -> 42;`

1. Lookup the one method on this interface
 - This is the public `T call()` method
2. This interface describes a function (`call()`) that expects no arguments and returns something of type `T`
3. The lambda expression is consistent with this description

Special void compatibility rule

- If a lambda has a statement expression as body, it's compatible with a function descriptor that returns void

```
@Test
public void test() {
    List<String> list = Arrays.asList("1","2","3","4");
    // Predicate has a boolean return
    Predicate<String> p = s -> list.add(s);
    // Consumer has a void return
    Consumer<String> b = s -> list.add(s);
}
```


Type inference

- The compiler can deduce an appropriate signature for the lambda, because the function descriptor is available through the target type
- the compiler has access to the types of the parameters of a lambda expression, and they can be omitted in the lambda syntax
- That is, the compiler infers the types of the parameters of a lambda

Example

■ Without inference

```
@Test
public void sortAllApplesOnWeightWithLambda() {

    stock.sort((Apple apple1, Apple apple2) ->
                apple1.getWeight()-apple2.getWeight());

    forEachT(stock, (Apple a)-> System.out.println(a));
}
```

■ With inference

```
@Test
public void sortAllApplesOnWeightWithLambda() {

    stock.sort((apple1, apple2) ->
                apple1.getWeight()-apple2.getWeight());

    forEachT(stock, (a)-> System.out.println(a));
}
```

■ when a lambda has one parameter the parentheses can be omitted

Capturing variables within lambda

- lambda expressions are allowed to use variables defined in an outer scope just like anonymous classes can
- They're called capturing lambdas
 - Example this lambda captures the variable `portNumber`:

```
@Test
public void testCapturingLocalVariable() {
    int portNumber = 1337;
    Runnable r = () -> System.out.println(portNumber);
    new Thread(r).start();
}
```

Restrictions on capturing

- Lambdas are allowed to capture without restrictions
 - instance variables
 - static variables
- local variables have to be explicitly
 - declared final or
 - Are effectively final

```
@Test
public void test() {
    int portNumber = 1337;
    Runnable r = () -> System.out.println(portNumber);
    new Thread(r).start();
    portNumber++;
}
```

✗ Local variable portNumber defined in an enclosing scope must be final or effectively final

Method references

- Method references allow reuse of existing method definitions and pass them like lambdas
- Can increase readability and feel more natural than using lambda expressions

In a nutshell

- Method references are a shorthand to lambdas only calling a specific method
- Often enhance readability
- They are compiled differently

Examples of static methods

- the following lambda expressions forward their arguments to static methods

```
Runnable r= ()-> Thread.dumpStack();  
Consumer<String> c=(str) -> System.out.println(str);
```

- Repeating the parameters isn't necessary
- a method reference is a lambda expression from an existing method implementation
- Use the :: operator

```
Runnable r1= Thread::dumpStack;  
Consumer<String> cs1=System.out::println;
```

An example, note ::

```
@Test
public void test() {

    store.sort((apple1, apple2)-> apple1.getWeight()-apple2.getWeight());

    forEach(store, apple-> {System.out.println(apple);} );
}

private void forEach(List<Apple> store, Consumer<Apple> consumer) {
    for (Apple apple : store) {
        consumer.accept(apple);
    }
}
```

```
store.sort((apple1, apple2)-> apple1.getWeight()-apple2.getWeight());

forEach(store, System.out::println);
}
```


A method reference ::

```
store.sort( (apple1, apple2)-> apple1.getWeight()-apple2.getWeight() );  
forEach(store, System.out::println);  
}
```

- With a special method comparing from `java.util.Comparator.comparing`

```
store.sort(comparing(Apple::getWeight));  
forEach(store, System.out::println);  
}
```

- reverse comparing with:

```
store.sort(comparing(Apple::getWeight).reversed());  
}
```

Examples of instance methods

```
ToIntFunction<Apple>f=a -> a.getWeight();
```

```
ToIntFunction<Apple>f=Apple::getWeight;
```

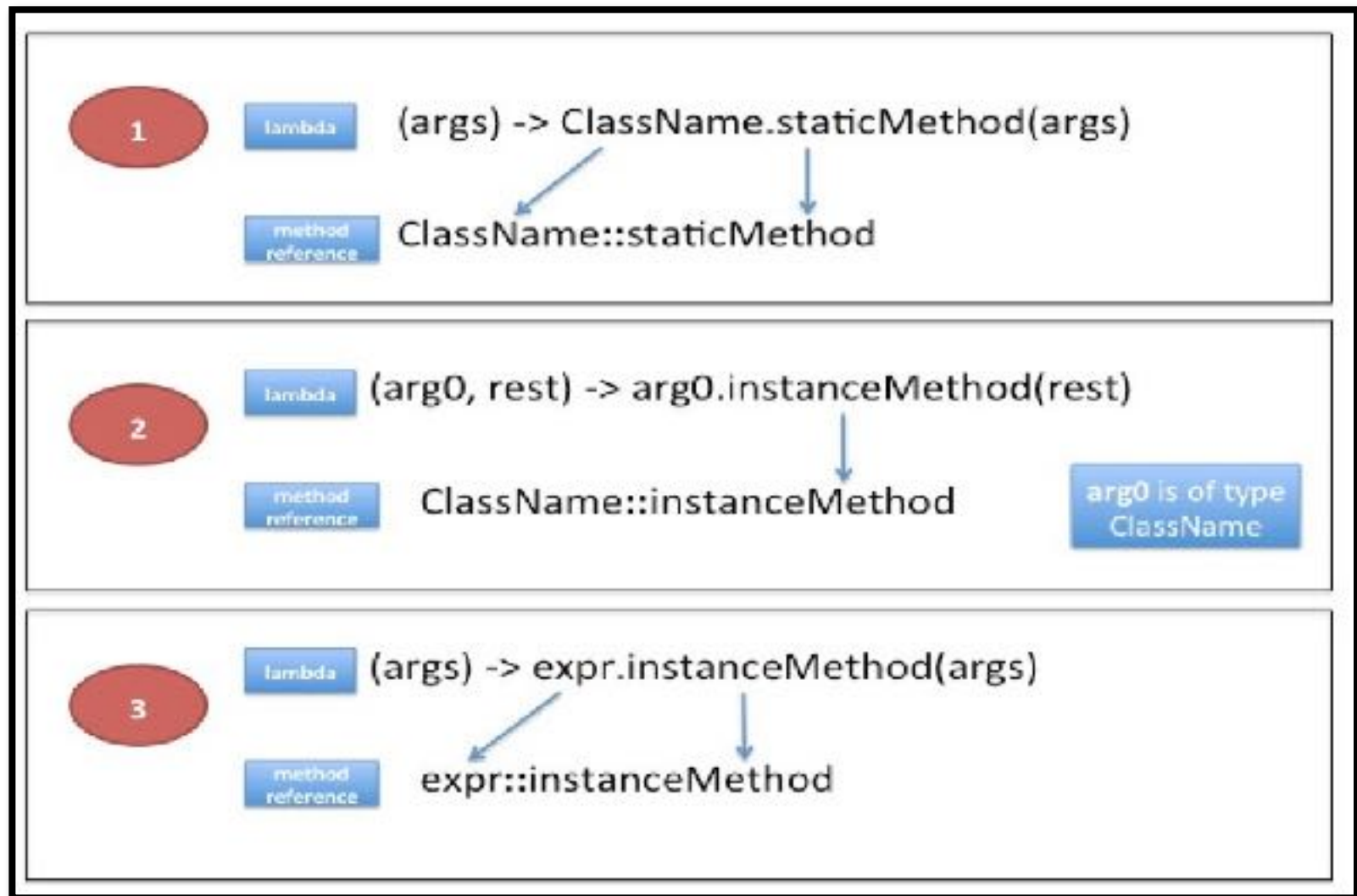
- the following lambda uses the first argument as a receiver object and forward the other arguments to an instance method

```
BiFunction<String, Integer,String> bf=(str, i) -> str.substring(i);
```

```
BiFunction<String, Integer,String> bf=String::substring;
```

- by referring to a method name explicitly, your code can gain better readability

Recipes



Exercise

- Assign method references of some methods of the System class to function variables
- As a reference: a table of general FunctionalInterfaces is shown on the next slide
- See the eclipse project for a more in depth description

Predefined Functional interface

New in Java 8	In java.util.function
<i>Consumer<T></i>	<i>Represents an operation that accepts a single input argument and returns no result</i>
<i>Function<T,R></i>	<i>Represents a function that accepts one argument of type T and produces a result of R</i>
<i>Predicate<T></i>	<i>Represents a predicate (boolean-valued function) of one argument</i>
<i>Supplier<R></i>	<i>Represents a supplier of results R</i>
<i>UnaryOperator<T></i>	<i>Represents an operation on a single operand that produces a result of the same type T</i>
<i>BinaryOperator<T></i>	<i>Represents an operation upon two operands of the same type T, producing a result of T</i>
<i>And a lot of specialisations</i>	<i>BiConsumer<T,U>, BiFunction<T,U,R>, BiPredicate<T,U>, DoubleFunction<R>,</i>

Exercise (similar to the former)

- Similar to the last exercise
- Assign method references of some methods of the String class to function variables
- As a reference: a table of general FunctionalInterfaces is shown on the former slide

Constructor references

■ Use ClassName::new to create new instance

```
@Test
public void useConstructorReferenceToCreateInstances() {
    List<Integer> weights = Arrays.asList(1,2,3,4,5);

    Function<Integer,Apple> f=Apple::new;

    List<Apple> apples= new ArrayList<>();
    for (Integer integer : weights) {
        apples.add(f.apply(integer));
    }
}
```

■ Define map function to increase readability

Example with map function

```
@Test
public void useMapFunctionToIncreaseReadability() {
    List<Integer> weights = Arrays.asList(1,2,3,4,5);
    Function<Integer,Apple> f=Apple::new;

    List<Apple> apples = map(weights, f);

    for (Apple apple : apples) {
        System.out.println(apple);
    }
}

public <T> List<T> map(List<Integer> list,Function<Integer,T> newClass) {
    List<T> freshInstancesList = new ArrayList<>();
    for (Integer integer : list) {
        freshInstancesList.add(newClass.apply(integer));
    }
    return freshInstancesList;
}
```


FactoryPattern and Constructor ref

- The capability of referring to a constructor without instantiating it enables interesting applications

```
Function<Integer, Apple> fruitFactory=Apple::new;
```

- For example, a Map can be used to associate constructors with a string value

Exercise

- Create an appleFactory method
 - Make a test in which you define an array of String, length m and an array of int, length n
 - Call the appleFactory method with these 2 arrays and an anonymous function which adheres the signature `BiFunction<String,Integer,Apple>`
 - The appleFactory should return a `List<Apple>` which in size equals $n*m$
- See a more in depth description in the eclipse project

Processing data with Streams

Now it really becomes
interesting!

Introduction to stream processing

- A stream is a sequence of data items produced one at a time
- A practical existing example is on the unix,linux platforms where many programs
 - operate by reading data form stdin
 - operate on the data
 - then writing data to stdout
- The unix cmdline allows these programs to be linked together with pipes (|)
 - `cat file1 file2 | tr “[A-Z]” “[a-z]” | sort | tail -3`

Explanation Unix example

- `cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3`
 - Prints the 3 words which appear latest in dictionary order after translating to lowercase
- You can say, sort takes a stream of lines as input and produces an other stream of lines as output
- Note in Unix the programs, cat, tr, sort and tail are executed in parallel
 - sort can process the first few lines from tr before cat and tr are finished

About Collections

- The Collections API is the most heavily used
- Collections are extremely fundamental
- They allow to group data and to process it
- But pre java 8 collections have still a lot of disadvantages
 - Much business logic entails SQL-like operations such as “grouping” or “finding” but we lack a sql like declarative style in java
 - Parallel processing of large collections is difficult and error prone

The answer to our problems:

- Streams!
- And although there is a lot to say about collections an easy start point for streams are collections

Setup 1

```
public class StreamTest {  
  
    @Test  
    public void aStartPointOfStreams() {  
  
        java.util.stream.Stream<Dish> stream = menu.stream();  
    }  
  
    List<Dish> menu = Arrays.asList(  
        new Dish("pork", false, 800, Dish.Type.MEAT),  
        new Dish("beef", false, 700, Dish.Type.MEAT),  
        new Dish("chicken", false, 400, Dish.Type.MEAT),  
        new Dish("french fries", true, 530, Dish.Type.OTHER),  
        new Dish("rice", true, 350, Dish.Type.OTHER),  
        new Dish("season fruit", true, 120, Dish.Type.OTHER),  
        new Dish("pizza", true, 550, Dish.Type.OTHER),  
        new Dish("prawns", false, 300, Dish.Type.FISH),  
        new Dish("salmon", false, 450, Dish.Type.FISH) );  
  
}
```


Setup 2 Note Dish is immutable

```
public class Dish {  
  
    private final String name;  
    private final boolean vegetarian;  
    private final int calories;  
    private final Dish.Type type;  
  
    public Dish(String name, boolean vegetarian, int calories, Dish.Type type) {  
        this.name = name;  
        this.vegetarian = vegetarian;  
        this.calories = calories;  
        this.type = type;  
    }  
  
    public getters ...  
  
    public enum Type {  
        MEAT, FISH, OTHER;  
    }  
}
```



Collection as a starting point

- A short definition is “a sequence of elements from a source that supports aggregate operations
 - Stream provides an interface to a sequenced set of values
 - streams don’t store elements: they are computed “on demand”
- Streams consume from a data-providing source such as Collections, or IO resources
- Aggregate operations: Streams support SQL-like operations

2 important characteristics streams

■ Pipelining:

- Many stream operations return a stream themselves. This allows operations to be chained and form a larger pipeline
- This enables certain Optimizations like laziness and short-circuiting
- Compare A pipeline of operations as a “query” on the data source

■ Internal iteration

- no for loop like with collections, it is done behind the scenes

An example

```
@Test
public void bDropWeightMenu() {

    List<String> lowCaloricMenues
        = menu.stream().
                filter(d-> d.getCalories()<300).
                map(Dish::getName).
                limit(3).
                collect(toList());

    for (String menu : lowCaloricMenues) {
        System.out.println(menu);
    }
}
```

- Filter a stream of dishes, only allow dishes with calories < 300, of the dish “row” select the name (map function), take only the first 3 elements(limit) → sql-like

Note

- filtering, extracting (map) or truncating (limit) functionalities are available through the Streams library
- As a result, the Streams API has more flexibility to decide how to optimize this pipeline

Differences Collection and Stream

- Both Collections and the new notion of Streams provide interfaces to a sequenced set of elements
- A Collection is an in-memory data structure, which holds all the values that the data structure currently has
- By contrast a Stream is a conceptually fixed data structure, but in reality elements are computed on demand

A different view

- a Stream is like a lazily constructed Collection: values are computed when they are solicited by a consumer (demand-driven, or even just-in-time, manufacturing)
- In contrast, a Collection is eagerly constructed (supplier-driven)

TRAVERSABLE ONLY ONCE

- Note that, similarly to Iterators, a Stream can only be traversed once
- After that a Stream is said to be “consumed”

```
@Test
public void printAllMenues() {
    Stream<Dish> dishStream = menu.stream();
    //1
    dishStream.forEach(System.out::println);
    //2
    dishStream.forEach(System.out::println);
}
```

- Second call results in a runtime exception

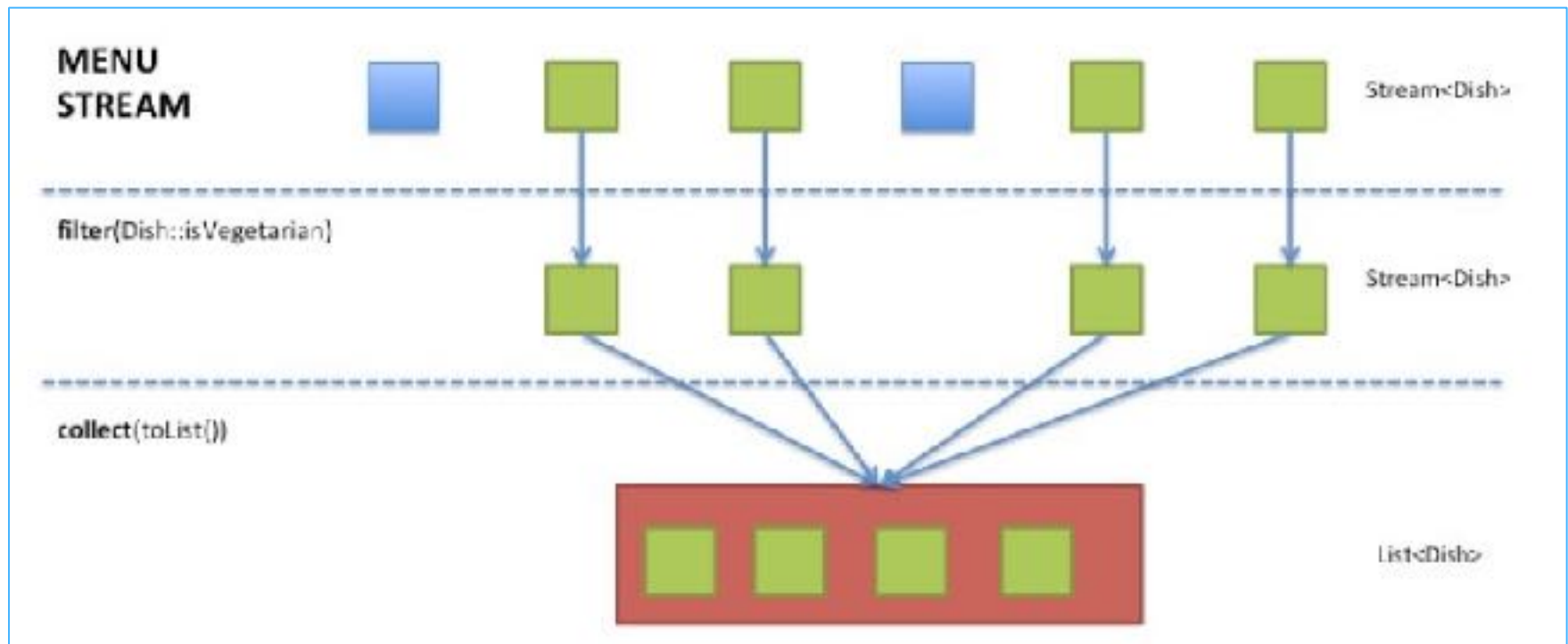
java.lang.IllegalStateException: stream has already been operated upon or closed

Filtering and slicing

- How to select elements from a stream
 - filtering with a predicate,
 - filtering only unique elements
 - ignoring the first few elements of a stream
 - or truncating a stream to a given size

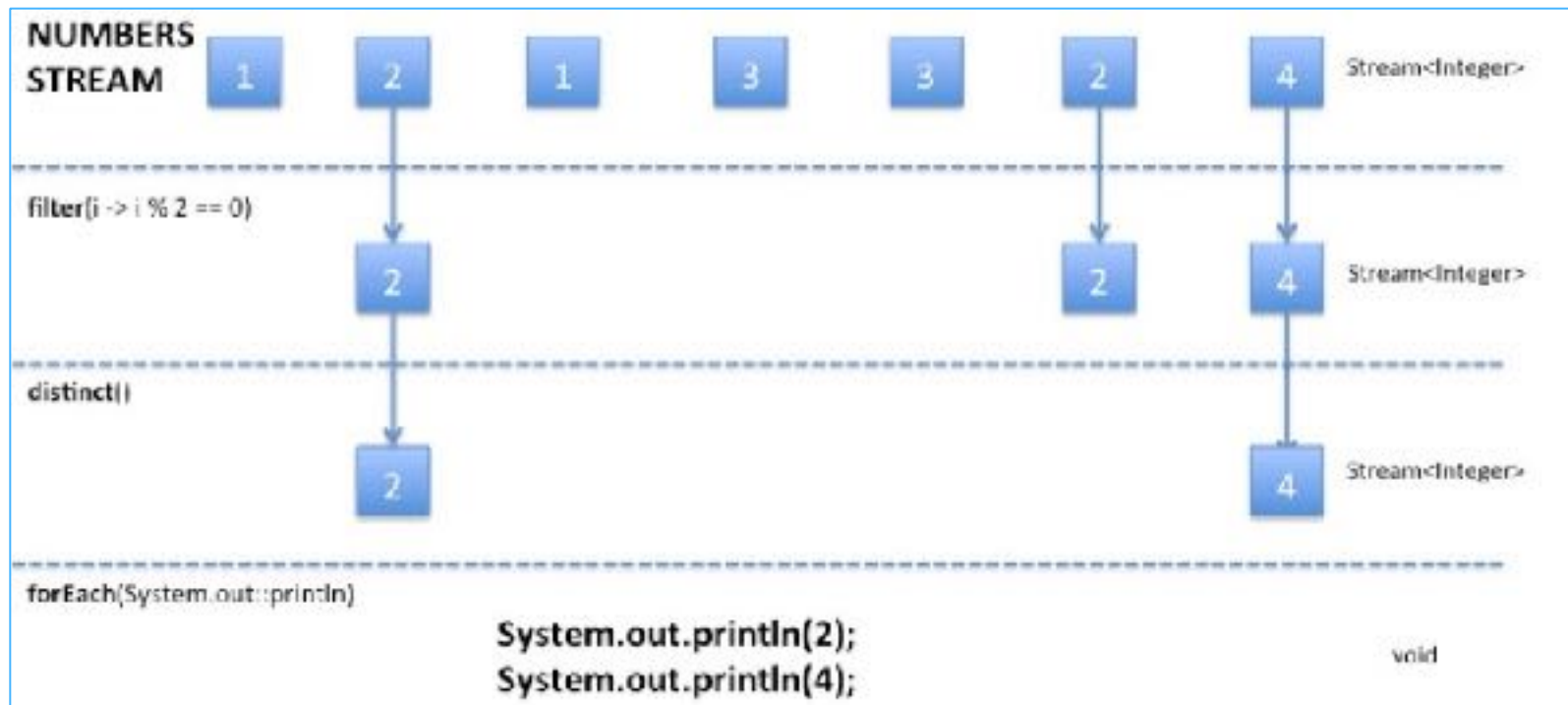
Filtering with a predicate

```
List<Dish> vegetarianMenu =  
    menu.stream()  
        .filter(d-> d.isVegatarian())  
        .collect(toList());
```



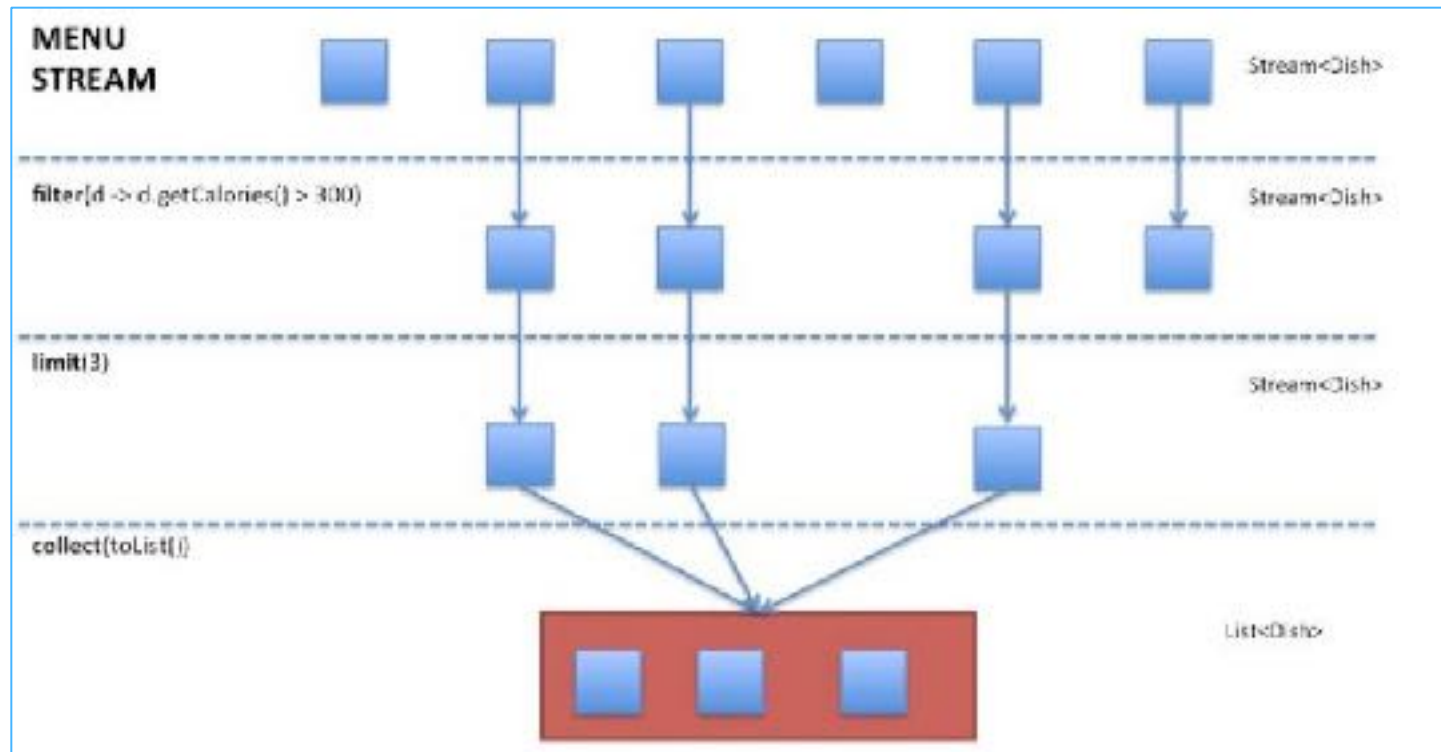
Filtering unique elements

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);  
    numbers.stream()  
        .filter(i -> i % 2 == 0)  
        .distinct()  
        .forEach(System.out::println);
```



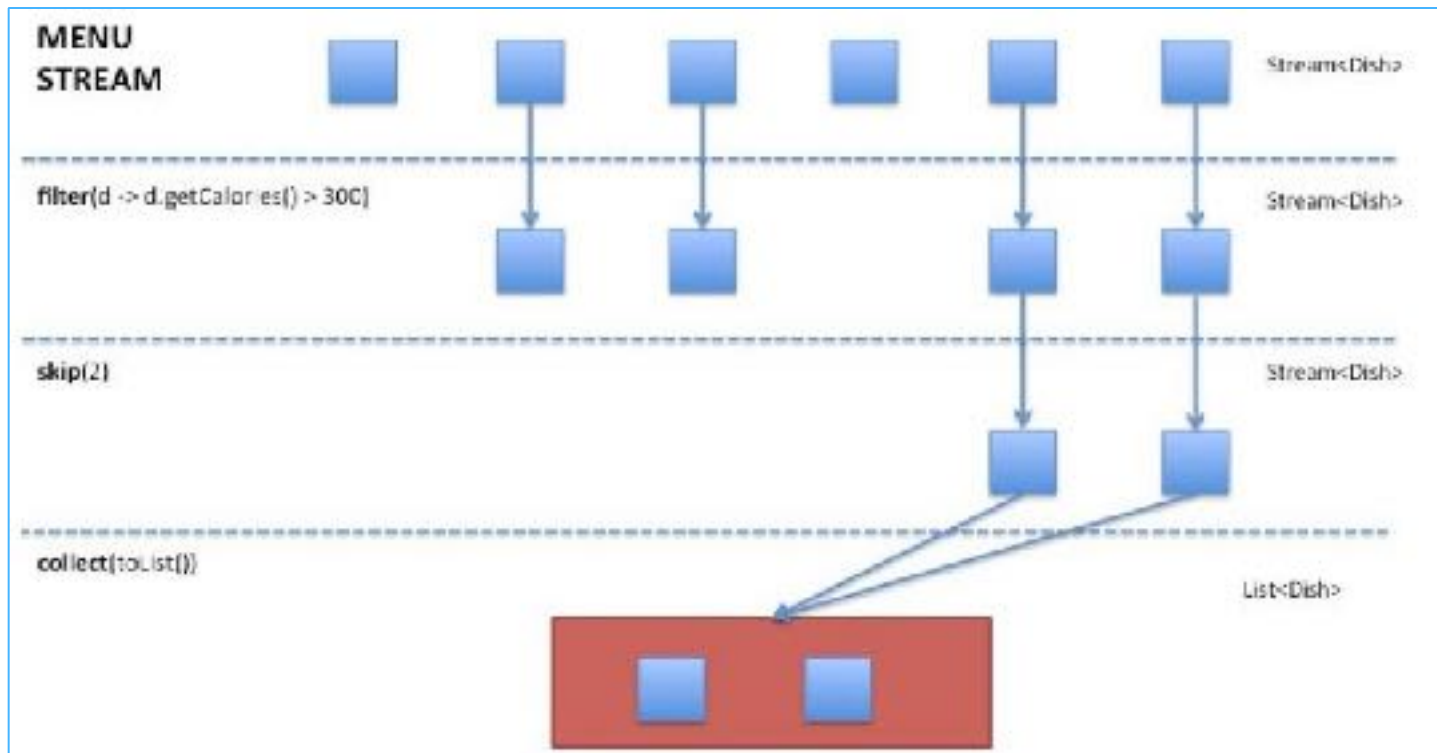
Truncating a stream

```
List<Dish> dishesLimit3 =  
    menu.stream()  
        .filter(d -> d.getCalories() > 300)  
        .limit(3)  
        .collect(toList());
```



Skipping elements

```
List<Dish> dishesSkip2 =  
    menu.stream()  
        .filter(d -> d.getCalories() > 300)  
        .skip(2)  
        .collect(toList());
```



Mapping

- How to select information from objects?
- The SQL equivalent from: “select” a particular column from a row or rows
- The Streams API provides map and flatMap
- About map
 - Map takes a function as an argument to transform the elements of a stream into another form

Mapping dish to string (dish name)

```
List<String> dishNames = menu.stream()
                                .map(d -> d.getName())
                                .collect(toList());
```

- `d -> d.getName()` results in a stream of Strings
- map the strings on int's by taking their length

```
List<Integer> nameLengths = menu.stream()
                                .map(d -> d.getName())
                                .map(s -> s.length())
                                .collect(toList());
```

Flattening Streams

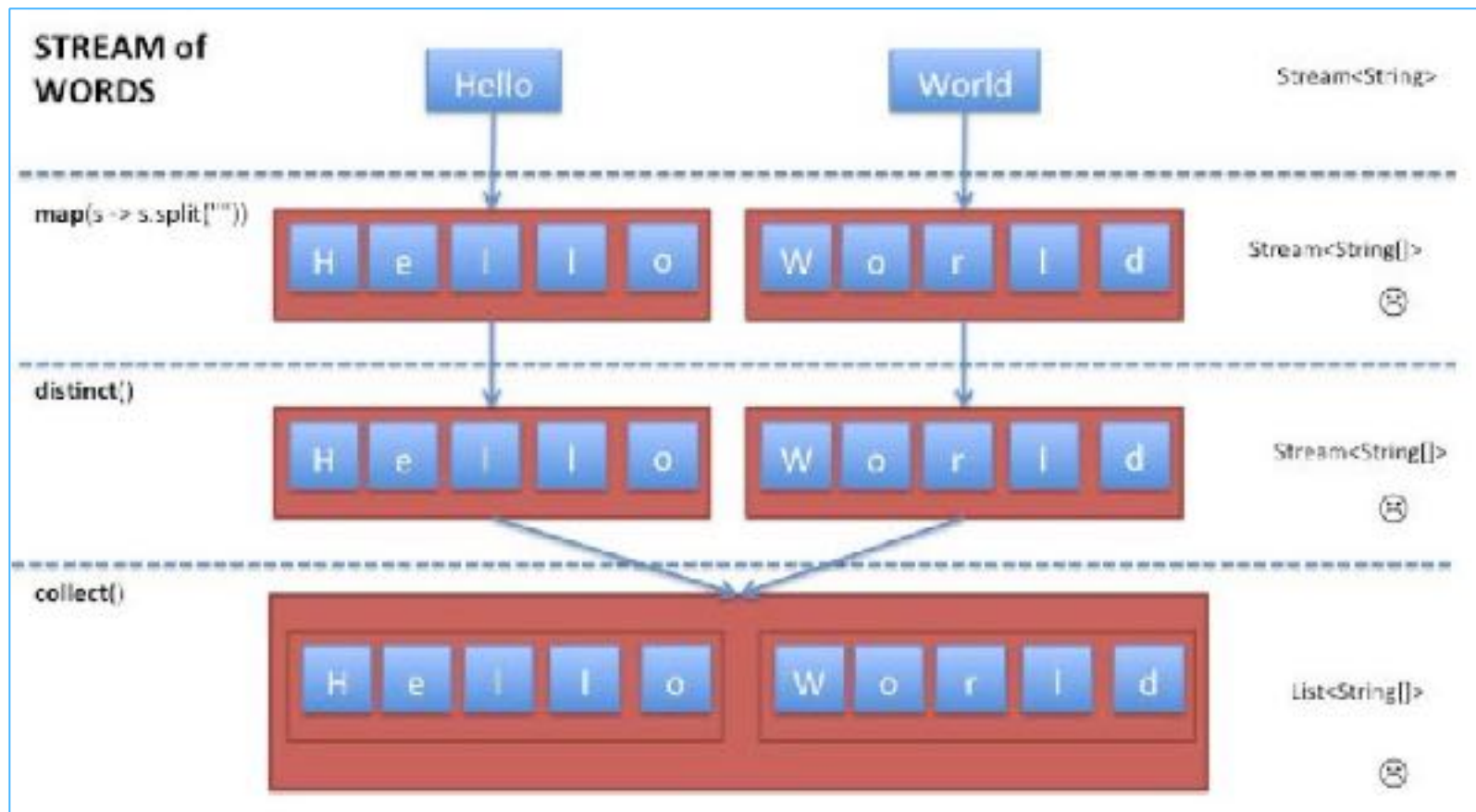
- given the list of words ["Hello", "World"]
return the list ["H","e","l","o","W","r","d"]

```
List<String> lines = Arrays.asList("Hello", "World");  
lines.stream()  
    .map((String line) -> Arrays.stream(line.split("")))   
    .distinct()  
    .forEach(System.out::println);
```

- Note: Arrays.stream(array) makes stream functionality available for arrays

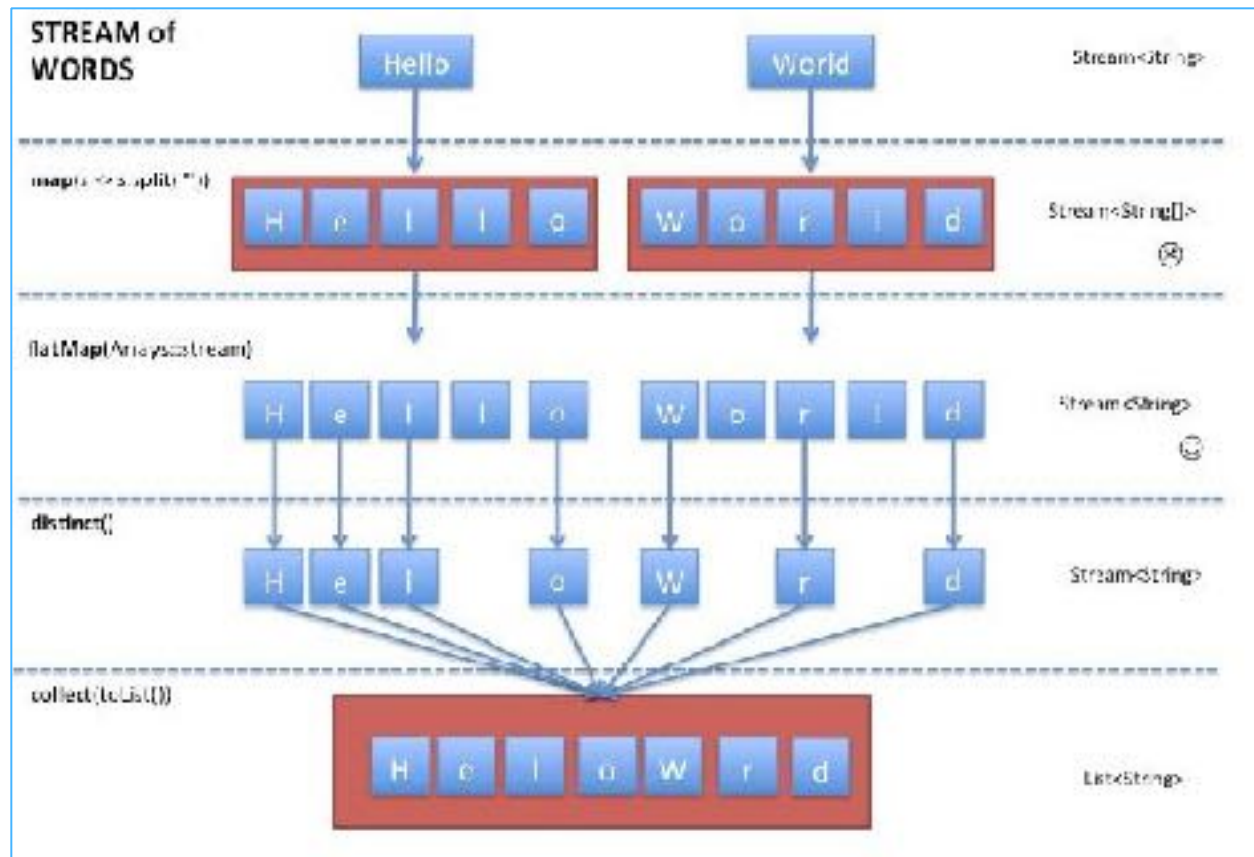
Result, hmm where are the chars?

```
java.util.stream.ReferencePipeline$Head@2f2c9b19  
java.util.stream.ReferencePipeline$Head@31befd9f
```



USING FLATMAP

```
lines.stream()  
  .flatMap((String line) -> Arrays.stream(line.split("")))  
  .distinct()  
  .forEach(System.out::println);
```



Exercise

- Given a list of numbers, how would you return a list of the square of each number? e.g. given [1,2,3,4] you should return [1, 4, 9, 16]
- Given two list of numbers, how would you return all pairs of numbers?
 - E.g. given a list (1, 2, 3) and (3, 4) you should return ((1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]. (represent a pair as an array with two elements)
- extend 2) to only return pairs whose sum is divisible by 3?

Special predicates

- Checking a predicate matches at least one element

```
public void getAVegatarianFriendlyMeal() {  
    if(streamOfDishes.anyMatch(Dish::isVegatarian)){  
        System.out.println("Yes some vegatarian food!");  
    }  
}
```

- Checking a predicate matches all elements

```
boolean isHealthy = streamOfDishes  
    .allMatch(d -> d.getCalories() < 1000);
```

The dual of anyMatch

- Checking a predicate that matches non of the elements

```
@Test
public void getColoricLowMe2() {

    boolean isHealthy = streamOfDishes
        .noneMatch(d -> d.getCalories() >= 1000);

    if(isHealthy){
        System.out.println("WeightWatch approved");
    }
}
```

Shortcircuiting operations

- These three operations:
 - anyMatch, allMatch and noneMatch
- make use of shortcircuiting,
- a Stream version of familiar Java short-circuiting `&&` and `||` operators
- As soon as an element is found a result can be produced
 - findFirst, findAny and limit are also shortcircuiting operations

Optional

```
@Test
public void findAVegatarianMeal() {

    Optional<Dish> vegaDishesOptional =
        streamOfDishes
            .filter(Dish::isVegatarian)
            .findAny();

}
```

- The stream pipeline is optimized
 - it performs a single pass and finish as soon as a result is found by using short-circuiting
- But, what is an Optional

Optional in a nutshell

- The `Optional<T>` class (`java.util.Optional`) is a container class to represent the existence or absence of a value
- It is introduced to avoid bugs related to null-checking! Handy methods are:
 - `isPresent`
 - `T orElse(T default)`

The signature of an Optional

- `isPresent()`: returns true if it contains a value
- `ifPresent(Consumer<T> block)`: executes the given block if a value is present
- `T get()`: returns the value if it is present otherwise throws a `NoSuchElementException`
- `T orElse(T other)`: returns the value if present, otherwise returns a default value

Is there an arbitrarily vega-meal?

```
@Test
public void findAVegatarianMeal() {
    streamOfDishes
        .filter(Dish::isVegetarian)
        .findAny()
        .map(Dish::getName)
        .ifPresent(System.out::println);
}
```

- Where does this `map(d -> d.getName())` comes from?
- It is a function defined on the `Optional`!

Finding the first element

```
@Test
public void findTheFirstVegetarianMeal() {
    List<Integer> someNumbers =
        Arrays.asList(1, 2, 3, 4, 5);
    Optional<Integer> firstSquareDivisibleByThree =
        someNumbers.stream()
            .map(x -> x * x)
            .filter(x -> x % 3 == 0)
            .findFirst();
}
```

- When to use findFirst and findAny?
 - Finding the first element is more constraining in parallel
 - don't care about which element is return use findAny it is less constraining

Reducing

- Characterizing reduce operations
- Queries that combine all the elements in the stream repeatedly, to produce a single value such as an integer.
- These queries can be classified as a reduction operation
- A stream is reduced to a value
- In functional jargon this is called a fold
 - this operation is seen as “folding” repeatedly a long piece of paper (your stream)

Investigate summing the elements

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

- The list of numbers is reduced into one number by repeatedly using addition
- There are two parameters in this code:
 1. An initial value of the sum variable (here) 0
 2. An operation to combine all the elements (+)

The reduce operation to the rescue

- Wouldn't it be great if all the numbers could also be multiply without having to copy and paste this code around
- This is where the reduce operation which abstracts over this pattern of repeated application can help

```
@Test
public void introductionReduce() {

    List<Integer> numbers = Arrays.asList(1,2,3,4,5);
    int sumTotal = numbers.stream().reduce(0, (sum, x) -> sum + x);
    assertThat(sumTotal, is(15));
}
```

A better look to reduce

```
int sum2 = numbers.stream().reduce(0, (a, b) -> a + b);
```

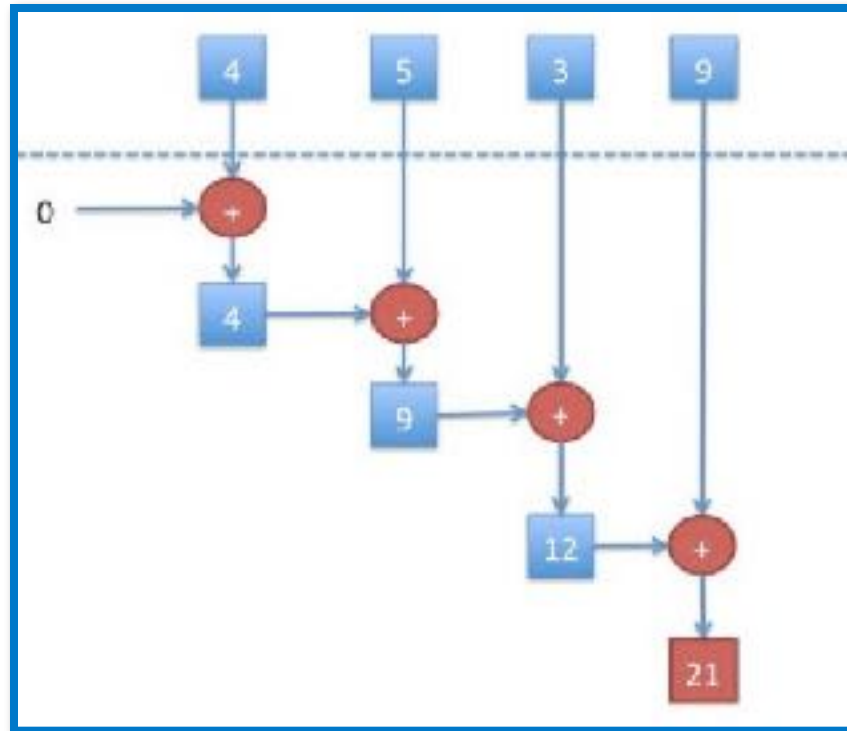
■ Reduce takes two arguments:

1. An initial value, here 0
2. A BinaryOperator<T> to combine two elements and produce a new value, here the lambda (a, b) -> a + b is used

■ Just as easily all the elements could be multiplied by passing (a, b) -> a * b to reduce

```
int sum2 = numbers.stream().reduce(1, (a, b) -> a * b);
```

`reduce(0, (a, b) -> a + b)`



- Use method reference to make this code more concise, new in Java 8 `Integer.sum`

```
numbers.stream().reduce(0, Integer::sum);
```


Maximum and minimum

- Use reduction to compute maxima and minima
- reduce takes two parameters:
 1. An initial value
 2. A lambda to combine two stream elements, and produces a new value, for maximum:

```
Optional<Integer> max = numbers.stream()  
    .reduce((a,b)-> a>=b ?a:b);
```

OR

```
Optional<Integer> max = numbers.stream()  
    .reduce(Integer::max);
```

Quiz Reducing

- How to count the number of dishes in a stream using the map and reduce methods?

Quiz Solution

- Map each element of a stream into the number “1” and then sum them using reduce

```
@Test
public void findNumberOfDishes() {
    Integer numberOfDishes =
        streamOfDishes
            .map(d -> 1)
            .reduce(0, (a, b) -> a + b);
    System.out.println(numberOfDishes);
}
```

- A chain of map and reduce is commonly known as the “map-reduce” pattern
 - made famous by Google’s use of it for web-searching because it can be easily parallelized

Exercises on memoryscottdb

- Familiarize yourself with the object model of Employees, Departments and SalaryGrades
- Print all the employees and departments
- Find all employees with job is Clerk
- Find all employees with job is Clerk and sort by salary (small to high).
- What are all the unique jobs of the employees?
- Find all employees working on department 10 and sort them by name.

Exercises on memoryscottdb

- Return a string of all employees names sorted alphabetically
- Are there any employees based in New York?
- Print all employees having a salary in scale 2
- How many employees earn a salary in scale 2 or 3
- What's the highest value of all the salaries?
- Find the salary with the smallest value

Exercises on memorypubsdb

- Familiarize yourself with the object model of this domain
- Print all Author objects and all Title objects
- Print all Authors living in CA
- Print all the full names, (auFname + auLname) sort them alphabetically
- Count the number of titles
- Print the values of the advance property
- Does some property advance contain a null?

Exercises on memorypubsdb

- Print all business books
- Count the number of authors that wrote business books
- Print all the authors that wrote business books starting from the Title stream
- Idem as former question but now starting from the Author stream
- Print all the authors that did not write business books

Numeric streams

■ Numeric streams: Why do we need them?

```
Integer numberOfDishes =  
    streamOfDishes  
        .map(d -> 1)  
        .reduce(0, (a, b) -> a + b);
```

■ The problem with the code above is:

- Boxing costs (can be large)

■ In addition, wouldn't it be nicer if we could call a `sum()` method directly as follows?

```
Integer numberOfDishes =  
    streamOfDishes  
        .map(d -> 1)  
        .sum()
```


Primitive streams specializations

- The method `map` generates a `Stream<T>` which disallows generation of a stream of primitives
- Primitive streams come to the rescue

```
IntStream intStream =  
    streamOfDishes.mapToInt(Dish::getCalories);
```

- Other methods are:
 - `mapToLong` producing a stream of long type
 - `mapToDouble` producing a stream of double type
- PrimitiveStreams offer functions like `sum()`, `min()`, `max()`

To,From object -> primitive stream

```
IntStream intStream =  
    streamOfDishes.mapToInt(Dish::getCalories);  
  
Stream<Integer> stream = intStream.boxed();
```

■ Default values

```
OptionalInt maxCalories =  
    streamOfDishes  
        .mapToInt(Dish::getCalories)  
        .max();
```

- max operation has no default, so an Optional is returned, choose your own default

```
int max = maxCalories.orElse(1);
```

Numeric ranges

- A common use case with numbers is working with ranges of numeric values
- on IntStream, DoubleStream and LongStream
 - range (exclusive upperbound)
 - rangeClosed (inclusive upperbound)

```
IntStream numbers = IntStream.rangeClosed(1, 100);
```

Building streams

- Streams can be created in many ways
- Examples follow for create a stream
 - from a sequence of values
 - from an array
 - from a file
 - from a generative function to create infinite streams

Stream from values

■ Using the static method Stream.of

```
Stream<String> stream =  
    Stream.of("Java 8 ", "Lambdas ", "Streams");
```

■ Using the empty method to get empty stream

```
Stream<String> emptyStream = Stream.empty();
```

■ Using the static method Arrays.stream

```
int[] numbers = {2, 3, 5, 7, 11, 13};  
int sum = Arrays.stream(numbers).sum();
```

Stream from a File

- Java's NIO API has been updated to take advantage of the Stream API
- Many static methods in `java.nio.file.Files` return a stream

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

Creating infinite streams

- The Stream API provides two static methods to generate a stream from a function
 1. `Stream.iterate`
 2. `Stream.generate`
- Such Streams create values on demand given a function and can calculate values forever
- It is generally sensible to use `limit(n)` on such Streams

Examples of “infinite” streams

```
Stream.iterate(0, n -> n + 2)  
    .foreach(System.out::println);
```

- About the example
- The iterate method takes an
 - initial value, here 0
 - a lambda (of type `UnaryOperator<T>`)
- The lambda is successively applied on each new value produced
- This results in an infinite stream, the stream is unbounded

The generate method

- the method generate also produces an infinite stream of values
- It doesn't apply successively a function on each new produced value however
- It takes a lambda of type `Supplier<T>` to provide new values

Example generate method

```
Stream.generate(Math::random)  
  .limit(5)  
  .forEach(System.out::println);
```

```
0.029727101463219885  
0.2628896161656359  
0.3841613118657313  
0.3668537270009897  
0.574106550966172
```

- The supplier that is used is stateless (a referece to the `Math.random()` methode)
- A stateful supplier can be used but that limits the use of it to sequential streams which is not to be preferred

Optional<T>

A better alternative to
null

Agenda

- What's wrong with null references and why you should avoid them
- From null to Optional: rewriting your domain model in a null-safe way
- Putting Optionals to work: removing null checks from your code
- Different ways to read the value possibly contained in an Optional
- Rethinking programming given potentially missing values

How to model the absence of a value?

■ The datamodel

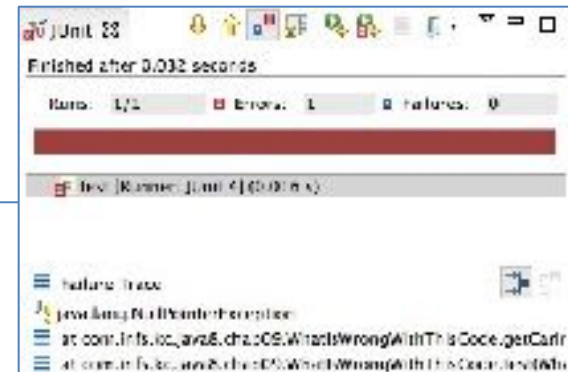
```
public class Person {  
    private Car car;  
  
    public Car getCar() {  
        return car;  
    }  
  
    public void setCar(Car car) {  
        this.car = car;  
    }  
}
```

```
public class Insurance {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
public class Car {  
    private Insurance insurance;  
  
    public Insurance getInsurance() {  
        return insurance;  
    }  
  
    public void setInsurance(Insurance insurance) {  
        this.insurance = insurance;  
    }  
}
```

Brain breaker

```
public class WhatIsWrongWithThisCode {  
  
    @Test  
    public void test() {  
        Person person = new Person();  
        String carInsuranceName = getCarInsuranceName(person);  
        System.out.println(carInsuranceName);  
    }  
  
    public String getCarInsuranceName(Person person) {  
        return person.getCar().getInsurance().getName();  
    }  
}
```



Code looks reasonable

- But persons don't necessary own a car
- What to return when no car is present?
- Unfortunately the null reference is often returned to indicate the absence of a value
- What to do about it?

Standard reflex: null checking

```
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";  
}
```

- Every time a variable is dereferenced that could be null perform a check
- Not null check on name (getName) is omitted because in this domain name can't be null
- This not null property is not derivable

Other line of attack

- Former approach results in nested if statements which is poorly readable
- Trying to get rid of nested if results in:

```
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();  
}
```

- This approach results in multiple exit points
- Easy to forget to check a null property

Problems with null

- It's a source of NullPointerException, by far the most common Exception in Java
- null worsens readability by code with often deeply nested null checks
- null doesn't have any semantic meaning
- null represents the wrong way to model the absence of a value in a statically typed language.
- Java always hides pointers from developers except in one case: the null pointer

Even more problems with null

- null creates a hole in the type system
 - Null carries no type or other information, meaning it can be assigned to any reference type
 - This is a problem because, when it's propagated to another part of the system, you have no idea what that null was initially supposed to be
- ok, null's are bad but what is the alternative?
- is there an alternative?

Look at Groovey

- Groovey introduced a safe navigation operator, represented by “?.”
- The java code becomes in Groovey:
 - `def carInsuranceName = person?.car?.insurance?.name`
- The Groovy `?.` operator allows navigation without throwing a `NullPointerException` by propagating the null reference returning a null in the when any value in the chain is a null

A safe navigation operator in Java ?

- Proposed for Java 7 but discarded
- Often a developer's reflex to a `NullPointerException` is introducing an if to check that a value is not null
 - Can be ok but you must first ask if it is allowed that this value is null in this particular situation?
 - Is it allowed that an algorithm or data model presents a null in this specific situation
- If this question is overlooked a bug may not be fixed but will be hidden
- A “?.” operator doesn't address this pitfall

Inspiration from Haskell & Scala

- Haskell and Scala, functional languages , take a different approach
- Haskell includes a *Maybe* type, which essentially encapsulates an optional value
- A value of type *Maybe* contains either a value of a given type or nothing
- There is no concept of a null reference.
- Scala has a similar construct called `Option[T]` to encapsulate the presence or absence of a value of type `T`

Java Optional<T>

- With these types you have to explicitly check whether a value is present or not using operations available on these types
- This enforces the idea of "null checking"
- No longer is it possible to "forget to do it" because it is enforced by the type system!
- Along the same lines Java adopted a new type `Optional<T>`

Introducing the Optional class

- Java 8 introduces a new class called `java.util.Optional<T>`
 - When a value is present, the Optional class just wraps it
- When a value is not present the absence of a value is modelled with `Optional.empty()`
- `Optional.empty()` is a static factory method which returns a singleton instance of the Optional class

Internals of Optional I.

```
public final class Optional<T> {  
    /**  
     * Common instance for empty()  
     */  
    private static final Optional<?> EMPTY = new Optional<>();  
  
    /**  
     * If non-null, the value;  
     * if null, indicates no value is present  
     */  
    private final T value;  
  
    private Optional() {  
        this.value = null;  
    }  
}
```

Internals of Optional II.

```
public T get() {
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
}
public boolean isPresent() {
    return value != null;
}
public T orElse(T other) {
    return value != null ? value : other;
}
public Optional<T> filter(Predicate<? super T> predicate) {
    Objects.requireNonNull(predicate);
    if (!isPresent())
        return this;
    else
        return predicate.test(value) ? this : empty();
}
```

Examples with Option type

- In the following examples given variables and types are used:

```
public void print(String s) {  
    System.out.println(s);  
}
```

```
String x = //  
Optional<String> opt//
```

- x may be null but opt is never null , but may or may not contain some value

How to create Optional instances?

```
String notNull="if null an exception is thrown!";  
opt = Optional.of(notNull);
```

```
String maybeNull="either return empty or present (set) ";  
opt = Optional.ofNullable(maybeNull);
```

```
String remark=" always returns empty , " +  
              "corresponding to null (Singleton) ";  
opt = Optional.empty();
```

Do something when optional is set

- if statements to safeguard us for NullPointerExceptions can be omitted

```
if (x != null) {  
    print(x);  
}
```

- Use the methods on the optional instead

```
opt.ifPresent(s -> print(s));  
opt.ifPresent(this::print);
```

reject, filter certain optional values

- Sometimes not only the reference must be set but also a certain condition must be met
- Pre Java 8 approach:

```
if (x != null && x.contains("criteria")) {  
    print(x);  
}
```

- Approach using Optional:

```
Opt.filter(s -> s.contains("ab"))  
    .ifPresent(this::print);
```

transform value if present

- Often a transformation on a value must be applied, but only if it's not null

```
if (x != null) {  
    String t = x.trim();  
    if (t.length() > 1) {  
        print(t);  
    }  
}
```

- Achieve this using the map function

```
opt.  
    map(String::trim).  
    filter(t -> t.length() > 1).  
    ifPresent(this::print);  
}
```

Some trickery look at opt.map()

```
opt.  
  map(String::trim).  
  filter(t -> t.length() > 1).  
  ifPresent(this::print);  
}
```

- If opt contains no value nothing happens and empty() is returned
- The transformation is type-safe

```
Optional<String> opt =Optional.of("1") ;  
  
Optional<Integer> len = opt.map(String::length);
```

- if opt was empty, map() does nothing except changing generic type.

Turn empty Optional in default value

■ Pre Java 8 solution

```
int length = (x != null)? x.length() : -1;
```

■ Java 8 solution

```
int length = opt.map(String::length).orElse(-1);
```

- if computing default value is slow, expensive or has side-effects use the `Supplier<T>` version

```
length = opt.  
    map(String::length).  
    orElseGet(() -> slowOrExpensiveDefault());
```

What if method returns Optional

```
public Optional<String> tryFindSimilar(String s){..}
```

■ Flatten Optional<Optional<T> -> Optional<T> with flatmap

```
Optional<Optional<String>> bad =  
    opt.map(this::tryFindSimilar);  
  
System.out.println(bad.get().get());  
  
Optional<String> nicer =  
    opt.flatMap(this::tryFindSimilar);  
  
System.out.println(nicer.get());
```