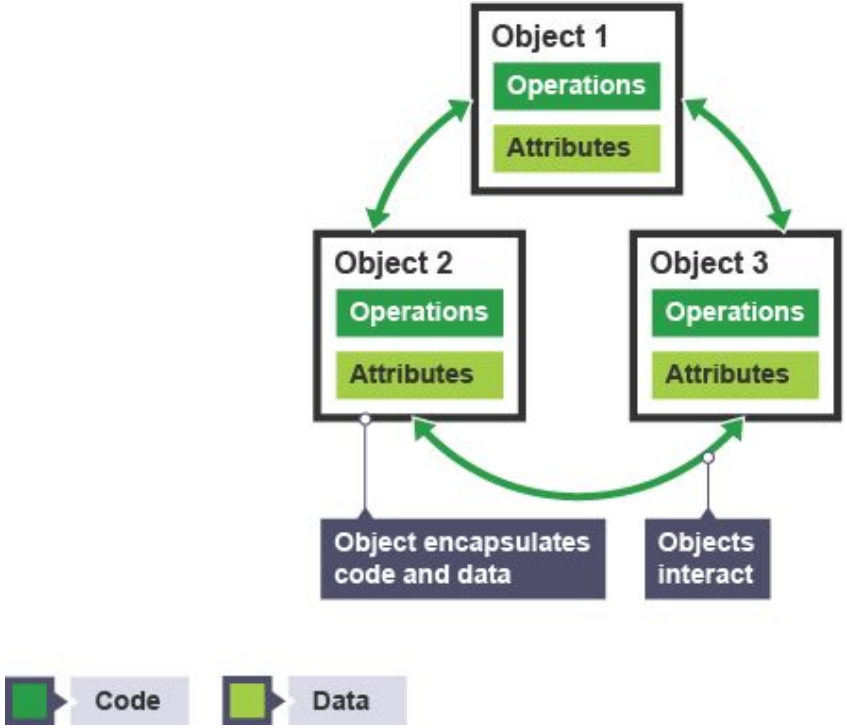
Lambdas and streams

- a survival guide -

http://bit.ly/jsurvive

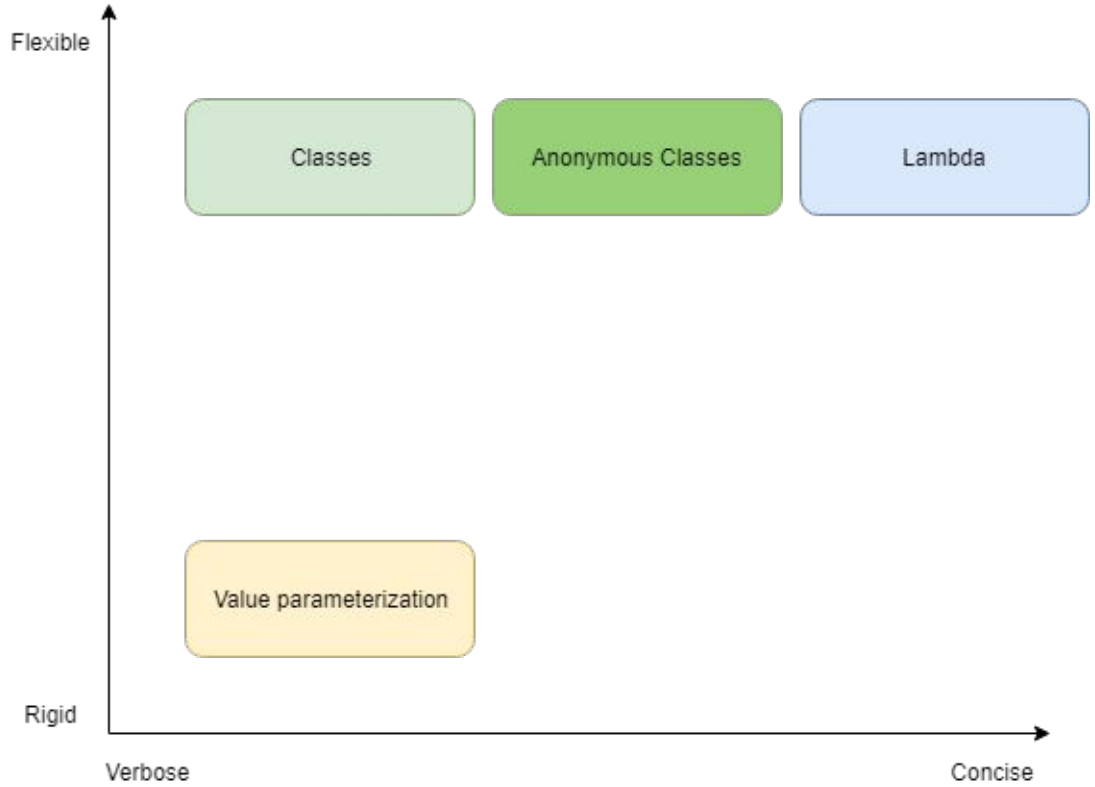
Better OOP!?


Object 1
Operations
Attributes
Object 2
Operations
Attributes
Object 3
Operations
Attributes
Object encapsulates code and data
Objects interact
Code
Data

How do your methods look like?

```java
int[] arr = { 1, 2, 3, 4, 5, 6 };

//imperative
List<Integer> result = new ArrayList<>(  );
for( int i = 0; i < arr.length; i++ )
{
    if( arr[i] % 2 == 0 )
    {
        result.add( arr[i] );
    }
}

System.out.println( result );

//declarative
List<Integer> result = Arrays.stream( arr ).filter( n -> n % 2 == 0 );

System.out.println( result );
```

Behaviour parametrization



Lambdas

Anonymous function (not method, those are tied to classes)

Can be:

- passed around as arguments
- stored in variables

It’s type is defined by a functional interface

Java 8 `interface`

Gained a new purpose beyond serving as a tool for OOP abstractization or as a simple markup:

- It became functional
- It can hold default implementations
- It can hold static methods

Of particular interest are the first and second newly gained abilities, remember them.

Functional `interface`

Simple definition: any interface (past, present or future) that holds exactly one abstract method. A convention of sorts that enables the use of the sole method as the type of the lambda expression.

Combining the above definition with default methods can result in a new way of changing behavior.

More lambdas

But functional interfaces are not the sole way to define lambdas

Any existing non-abstract method can become a lambda expression

Strip away the name and the remaining signature behaves just like a functional interface

The result?

- Bridges the gap between regular and functional code
- Promotes reuse of already written code

Method references

- First class methods:
  (Apple a) -> a.getWeight() => Apple::getWeight
- Static methods:
  (String s) -> Integer.parseInt(s) => Integer::parseInt
- Methods of existing instances:
  (Integer index) -> existingList.get(index) => existingList::get
- Constructors:
  (String name) -> new Book(name) => Book::new

Streams

Streams

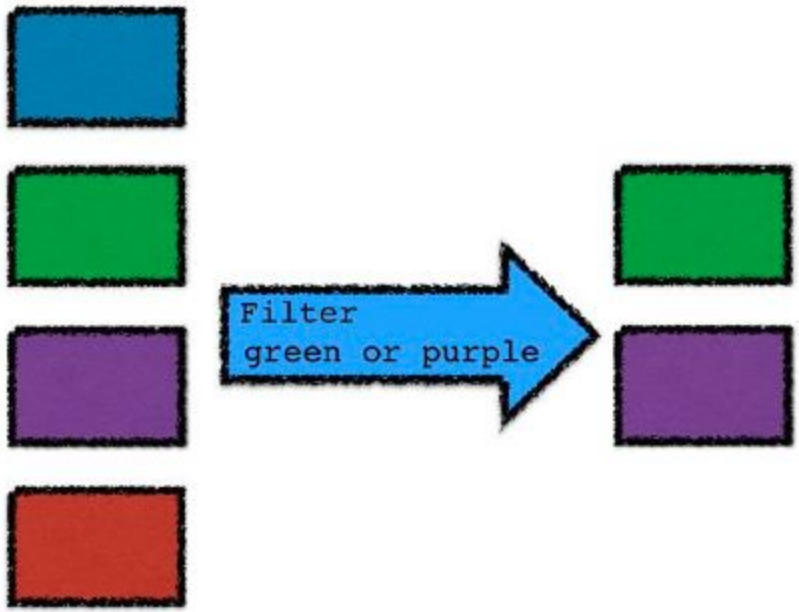
The goal is to have SQL-like, declarative, pipelined operations for sources of data:

- Collections
- Files
- Arrays

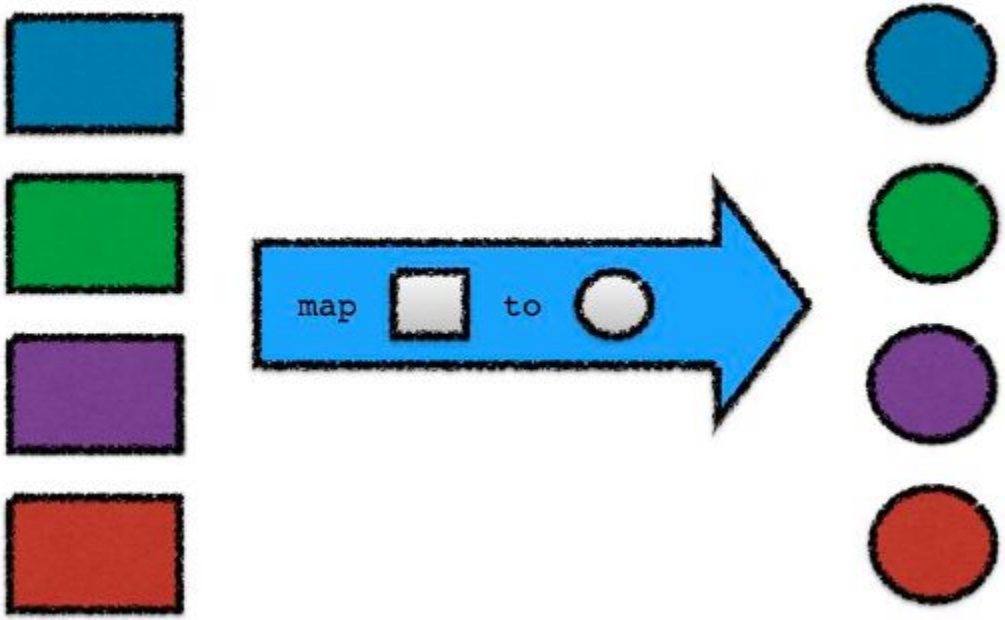
Built around the idea of internal iteration and lazy evaluation

Putting it together: a stream is a sequence of elements from a source that supports aggregate operations

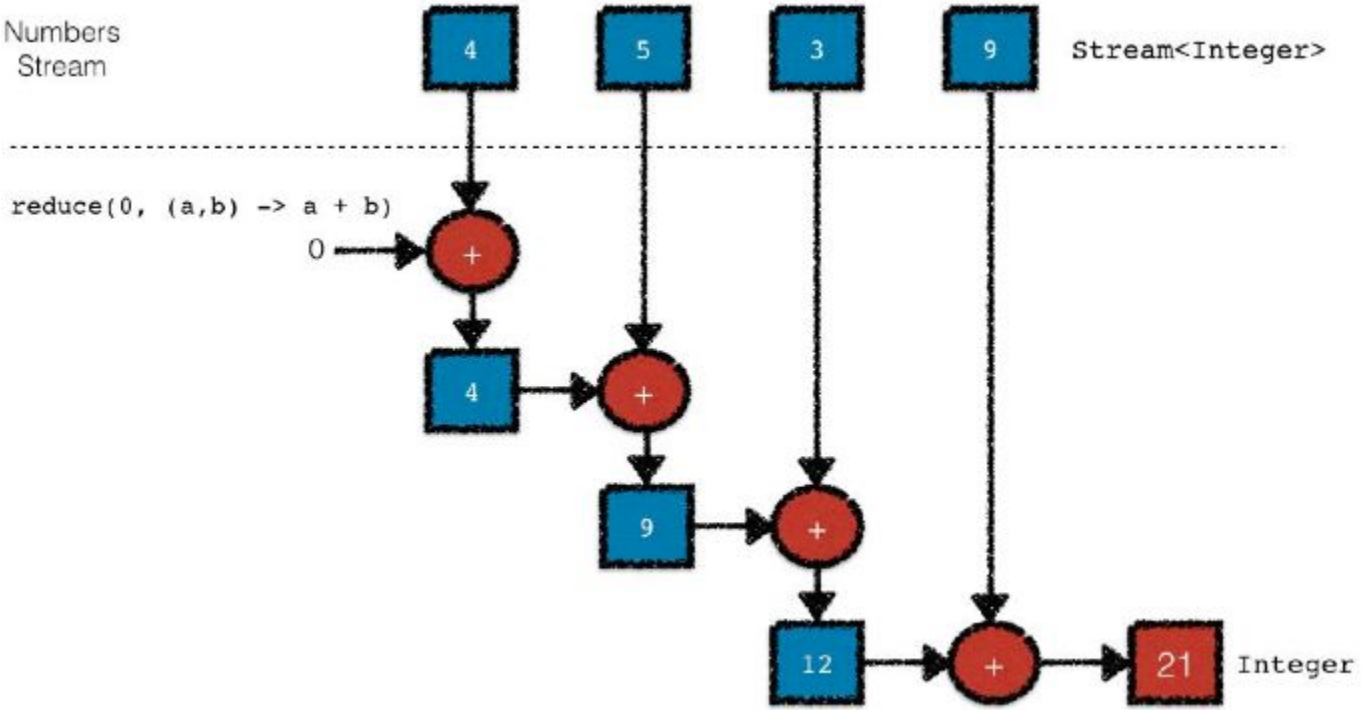
Filter



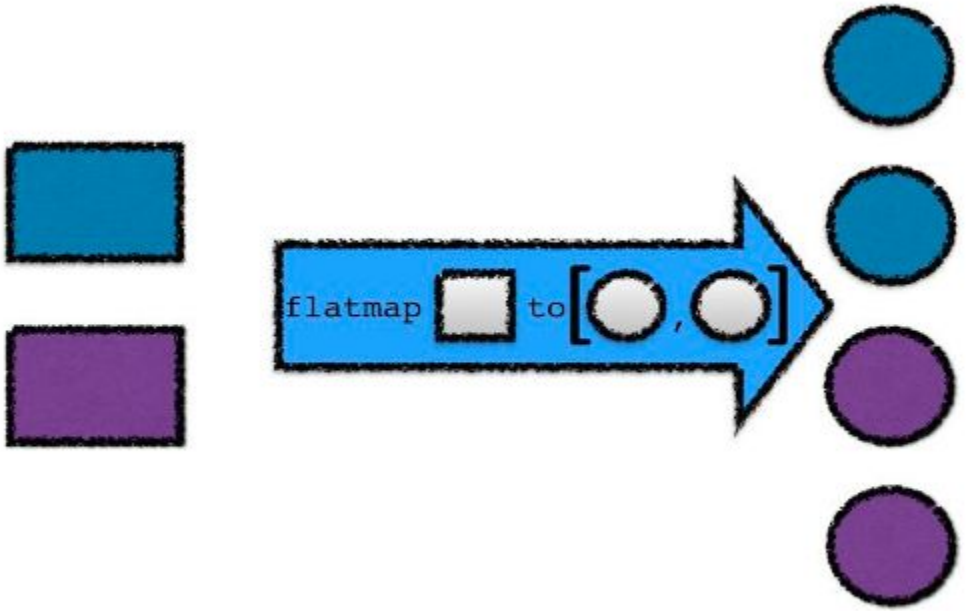
Map



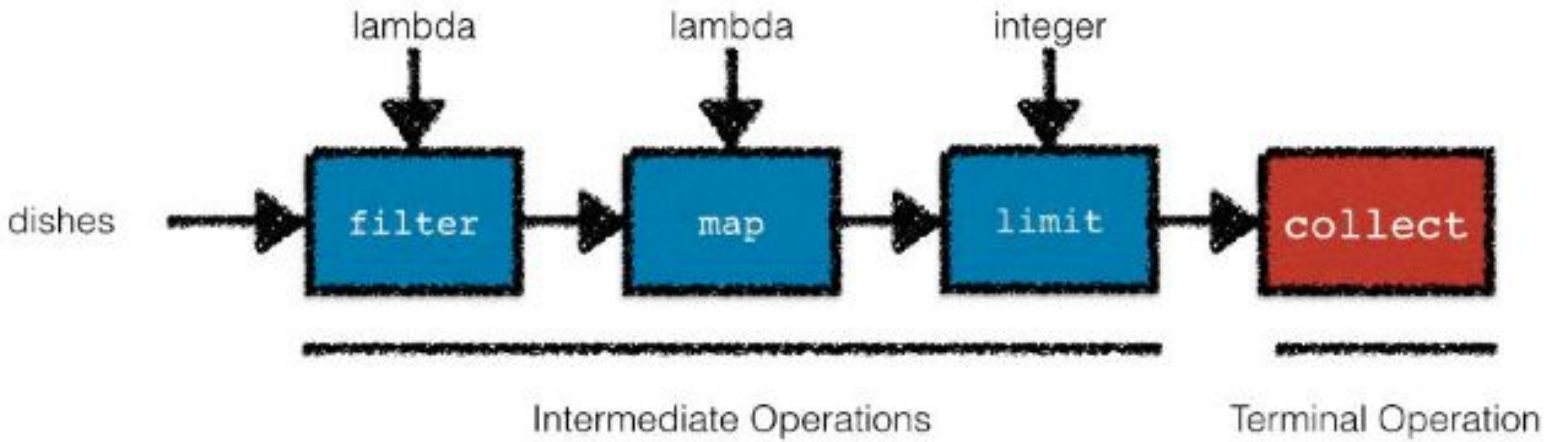
Reduce


Numbers
Stream
4
5
3
9
Stream<Integer>
reduce(0, (a,b) -> a + b)
0
+
4
+
9
+
12
+
21
Integer

Flat map

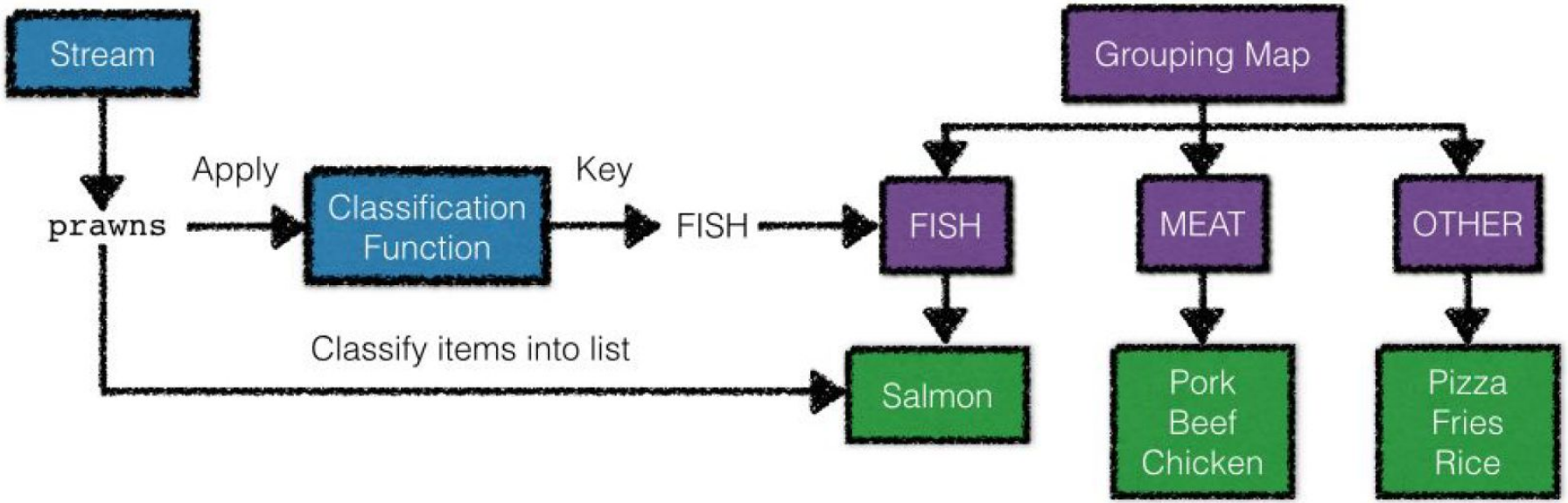


Collect


lambda
lambda
integer
dishes
filter
map
limit
collect
Intermediate Operations
Terminal Operation

Group

```
Map<Dish.Type, List<Dish>> dishesByType =
    menu.stream().collect(groupingBy(Dish::getType));
```



Streams usage

Streams and lambdas for all their good benefits can just as easily start to rot your code

They encourage the usage of anonymous functions and can add up the number of instructions on a single line

All in all it leads to cluttered, logic dense code that is hard to read and debug

Streams usage

Prefer using method references where the lambda expression is not trivial

Organize the stream pipeline such that each step is on a different line

Lambda expressions need to be short, if you find yourself putting braces then probably you need a method or to rethink your implementation

Prefer using the functional interfaces already defined in the JDK, by doing this you adhere to the same abstractions and the same concepts

Streams usage

Learn about methods available in the JDK that do more in a single call and use them in streams in order to simplify and clarify the logic expressed through the pipeline: `putIfAbsent` or `getOrDefault`

Learn about the new concepts such as Optional

Streams usage

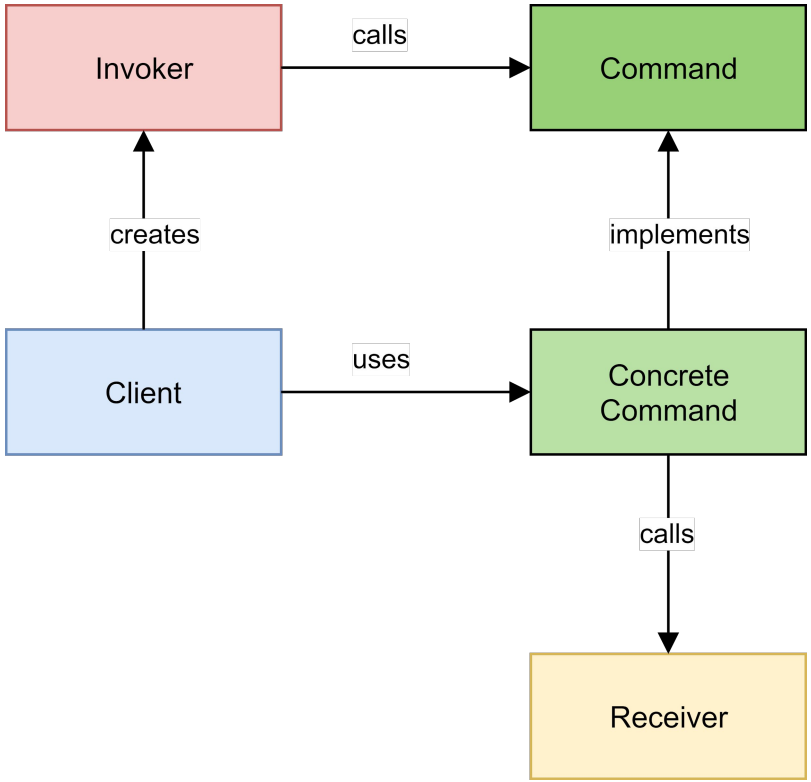
Don’t forget about streams specializations: IntStream, LongStream and DoubleStream, use them when dealing with primitive types

Order of the intermediate operations matters:

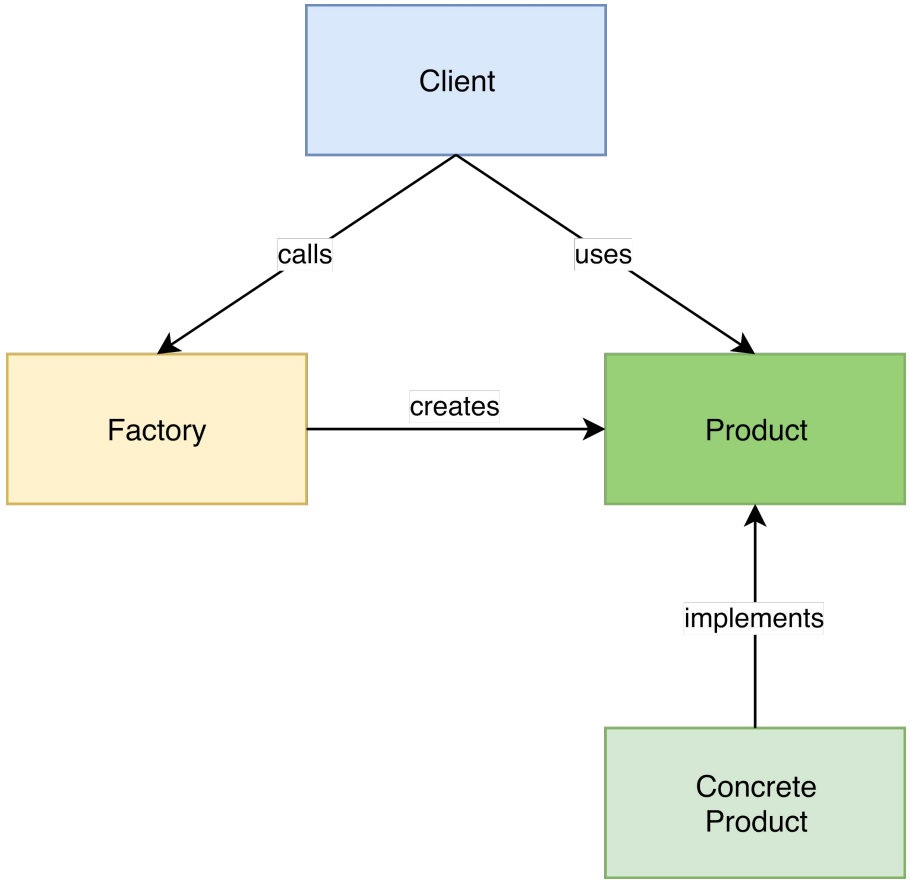
- expect `stream -> map -> filter` to perform worse than `stream -> filter -> map`

Better design patterns

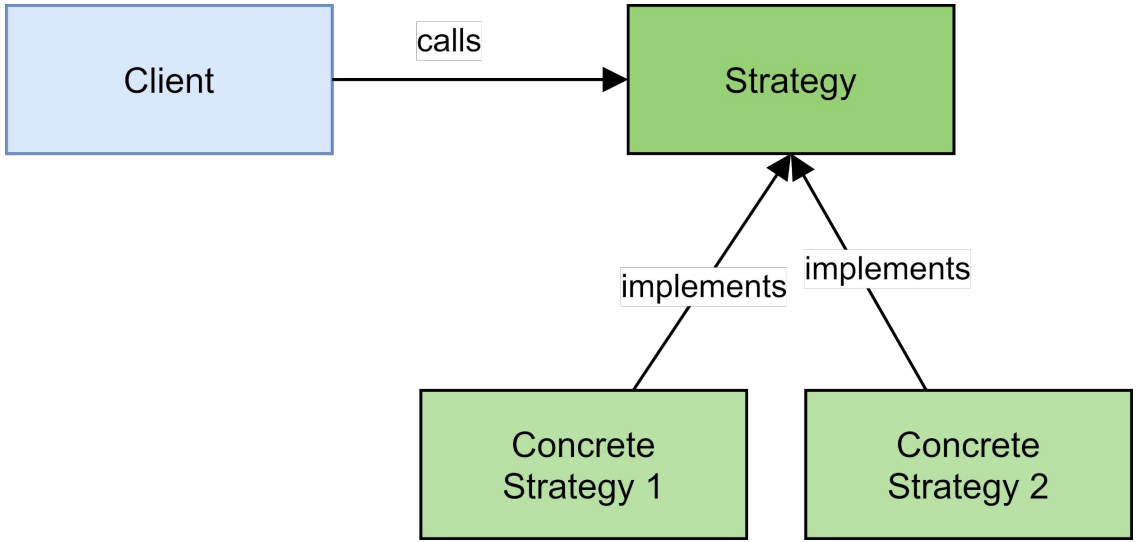
Command


Invoker
calls
Command
creates
implements
Client
uses
Concrete Command
calls
Receiver

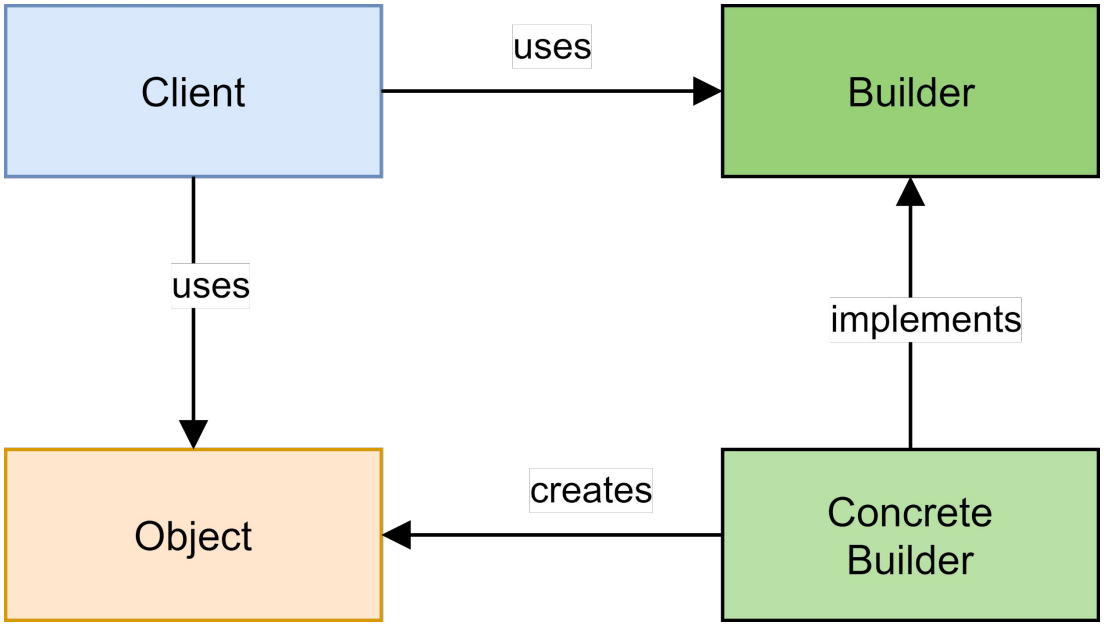
Factory


Client
calls
uses
Factory
creates
Product
implements
Concrete
Product

Strategy


Client
calls
Strategy
implements
implements
Concrete Strategy 1
Concrete Strategy 2

Builder


Client
uses
Builder
uses
implements
Object
creates
Concrete
Builder

Functional concepts

Any problems with our Java code?...

- Too imperative (low level logic, exposing details, noise)
- Mutating state everywhere (concurrency issues, flow hard to understand/test)
- Defensive programming (null checks, try-catch everywhere, synchronization pain)
- Debugging pain (‘Heisenbugs’ in production...)
- Testing pain (is higher test coverage enough of an assurance?)
- “No obvious defects” vs “obviously no defects”...

Functional Programming.. in Java?

- Why care about with FP? (“we have OOP”)
  - Trends (other languages, future Java; hardware)
  - Benefits
  - Problems to solve

- Is FP even possible in Java?
  - Is Java a functional language?
  - What is supported, what are the limits?
  - Is FP compatible/usable with OOP?
  - Why not just change the language?
  - Any other ways to make it more functional? (libraries)

FP - Where to start

- FP concerns: “to complicated, too much math / theory”
  - can we really learn/apply any of that? (in Java)
  - how much should we know to get started?...
  - learning curve - when will we see any benefits?...
  - will others developers (my team) still understand my code?...

- Can be done gradually:
  - Start with migration to a more declarative thinking (before FP)
  - Applying (some of) the general FP principles: possible now, would already brings benefits

Declarative vs Imperative

- Imperative - focus is on “how” things need to be done (step by step, in detail, lower level logic)
- Declarative - focus more on “what” needs to be done (higher level loginc)
- Functional is a subset of declarative (FP = declarative + high-order)

- Where to start?
  - Think more about “what” your code neeeds to do (not “how”)
  - Structure your code to reflect that (small pure functions with clear named, combined then in other short functions of higher level)
  - Know and use your tools: language syntax (forEach, streams); look for more declarative methods in existing API (like: Map.putIfAbsent(), getIfPresent(), merge(), replaceAll)
  - Make your methods more generic/declarative (higher-order functions, generics..)

FP concepts - Pure functions

```
public class DeclarativeVsImperative {

   static void findNemo_Imperative(List<String> names) {
       boolean found = false;
       for (String name : names) {
           if (name.equals("Nemo")) {
               found = true;
               break;
           }
       }
       if (found) System.out.println("Found Nemo");
       else System.out.println("Sorry, Nemo not found");
   }

   static void findNemo_Declarative(List<String> names) {
       if (names.contains("Nemo"))
          System.out.println("Found Nemo");
       else
          System.out.println("Sorry, Nemo not found");
   }
}
```

What is FP ?

- Key concepts:
  - Functions as first class objects
  - Pure functions
  - Higher order functions

- Rules:
  - No state
  - No side effects
  - Immutable variables
  - Favor recursion over looping

FP concepts - Functions as first class objects

- Means you can: create an instance of a function (of any type you need), store it in a variable, pass it around...

- In Java - functions not really first class objects
  - closes we get are Lambda Expressions
  - they still have an instances of a class behind scenes
  - rely on having/declaring some Functional interfaces matching the function type we need (and then on instantiating anonymous classes implementing those interfaces)
  - Still, they make life much easier / much nicer code

FP concepts - Pure functions

- A function is **pure** when:
  - The execution of the function has no side effects
  - The return of the function depends only on its input parameters

- Has no side effects:
  - Does not change any state: on containing class, on other classes, even on its input params; also does not change the outside world
  - A return type of “void” is a code smell (‘what does this it actually do?’..)

- **Substitution principle:**
  - A pure function can be replaced at any time with its implementation (inlined), with no changes to syntax of the program / end results.
  - *It makes reasoning about the code much simpler! (and also testing, debuging)*

FP concepts - Pure functions

```
public class ObjectWithPureFunction {
    public int sum(int a, int b) {
        return a + b;
    }
}

public class ObjectWithNonPureFunction {
    private int value = 0;

    public int add(int nextValue) {
        this.value += nextValue;
        return this.value;
    }
}
```

FP concepts - Higher-order functions

- A function is **a higher order** function when:
    - Takes one or more functions as parameters
    - Returns another function as the result

- Possible by using lambdas
- Few actual examples in the Java libraries: Comparator, Collections.sort ..

FP concepts - Higher-order functions

```
List<String> list = Arrays.asList("One", "Abc", "BCD");

Collections.sort(
       list,
       (a, b) -> a.compareTo(b));


Comparator<String> comparator = (a, b) -> a.compareTo(b);
Comparator<String> comparatorReversed = comparator.reversed();

Collections.sort(list, comparatorReversed);
```

FP Rules - No state

- Should have no state in our code
- Typically means: no external state
  - A function may use local variables containing temporary state, those are safe as long as inaccessible from outside (take care not to expose them in API)
  - It may not reference fields of the class it belongs to, or other classes

- No way to enforce this, you just need to properly design your code for this (“static” is not a solution, etc)

- Discussion: many useful programs need to store/use some state
  - Minimize it, contain it to only small places in code, access it with a clear/safe api
  - Push it to outside systems (filesystem, DB)
  - The core code of your app should have as little state as possible

FP Rules - No state

```
public class CalculatorWithNoState {
    public int sum(int a, int b) {
        return a + b;
    }
}

public class CalculatorWithState {
    private int initVal = 5;

    public int sum(int a) {
        return initVal + a;
    }
}
```

FP Rules - Immutability

- Because they make it easier to avoid side effects
- Variables, once initialised, should never change
  - The assignment statement - introduces the concept of time (the world before and after), is a direct cause of side-effects
  - Not only variables, but all data structures / composed objects / parameters

- Hard to do: requires an adjustment of thinking, for an OOP/iterative programmer
  - *Even if it's hard to believe, a lot of useful code can be written without any assignments!*

- Java doesn't make it easier:
  - "final" is not the default (for fields, params), need to write it everywhere
  - uses lots of mutable APIs - like for Collections (void clear(), boolean add())
    - Unmodifiable collections (="views") don't solve this
    - Libraries like Guava (ImmutableList…) are only a partial response, and arguably even worse (runtime exceptions, not compile time)

FP Rules - Immutability

```
public class Person {
   //immutable - use final!
   private final int cnp; //mandatory
   private final String name; //optional on build

   //mutable state (no final) - if really needed...
   private int age = 0;

   public Person(int cnp, int age) { this(cnp, "unknown", age); }

   public Person(int cnp, String name, int age) {
       this.cnp = cnp;
       this.name = name;
       this.age = age;
   }

   public int getCnp() { return cnp; }
   public String getName() { return name; }
   public int getAge() { return age; }

   public void setAge(int age) { this.age = age; }
}
```

FP Rules

- **No side effects:**
  - A function should not change any state (outside the function)
  - Includes variables of the parent class, state in parameters or state of external sysmstes (file system, DB, console)

- **Favor recursion over looping:**
  - Recursion uses call functions to achieve looping, so the code becomes more functional
  - Another alternative in Java: stream methods
  - *Discussion: Java does not have optimizes support for this (Tail-Call-Optimization, like Clojure, Scala,..), leading to poorer performance and the risk of blowing the stack for deep recursion*

Refactoring to streams

Functional null handling?

Functional handling of null

- Null pointer references = a ‘billion dollar mistake’ (Tony Hoare)
  - May be hard to avoid in lower level languages (assembler, C..)
  - Should be avoidable in higher-level ones

- Java - where does null appear from?
  - Not initialized fields, allocated but uninitialized arrays
  - Explicitly passed around:
    - returned from functions as a kind of default (Map.get()..)
    - Passed to function as parameters by some code (coding mistakes, but sometime event the api is defined like that…)

- Any way to avoid it?...

Functional handling of null

- To **model absence**, we should use **Optional**
  - javadoc: “A container object which may or may not contain a non-null value”
- Lots of methods to create, work with it in a safe(r) way:
  - of(), **ofNullable**(), empty(), isPresent(), orElse()
  - some unsafe ones: get() -> avoid using this!
  - many functional ones (higher-order): map, filter, ifPresent, flatMap, orElseGet…
- Usage:
  - recommended to be used as return type (when cannot compute a value)
  - should not be used for class fields or function input params
- Not fully safe - some methods throw (runtime) exceptions, you may still get a “null” instead of the Optional wrapper, etc..
  - avoid using .get() directly! Should always guard it with .isPresent() check, or even better, should use only map(), filter(), ifPresent(), orElse() methods!

Avoiding null with Optional

```
class Dog {
   private final String breed; //mandatory (not null/empty)
   private final Person owner; //optional (nullable)

   private Dog(String breed, Person owner) {  this.breed = breed;  this.owner = owner;  }

   String getBreed() { return breed; }  //never null, ok to return it directly

   Optional<Person> getOwner() { return Optional.ofNullable(owner); }

   public String toString() { return "Dog{" + "breed='" + breed + '\'' + ", owner=" + owner + '}'; }

   //some factory methods (to enforce validations on build)
   static Optional<Dog> build(String breed) { return build(breed, null); }

   static Optional<Dog> build(String breed, Person owner) {
       return (breed == null || breed.isEmpty()) ?
              Optional.empty() :
              Optional.of(new Dog(breed, owner));
   }
}
```

Functional exception handling?

Exceptions

- Why are they a problem?
  - Behave like unexpected exit points (non-local GOTOs)
  - Invisible in source code, or when async
  - Expensive to build/throw (synchronized building of stacktrace)
- Java anti-patterns for exception handling:
  - Log and throw, log and return, catch and ignore, declaring and/or catching top exception classes, throw from finally, etc..

- Strongly typed language - compiler can/should help us on exception flows too!
  - Checked exceptions are also not a solution (may be thrown, still interrupt the flow, overcrowd the api ..)

- Exceptions are a side-effect! -> substitution principle no longer holds!

Exceptions

- Solution: avoid exceptions as much as possible. Is it possible? how?..
- Principles:
  - all execution paths of code should be clearly visible in code;
  - there is **no exceptional flow** (at least not handled explicitly - OOM remains..)
  - methods should never throw (new) exceptions, but instead encapsulate all possible responses in the result (return type)

- This means we need **richer return types** - wrappers which can contain both the “good” reponse (happy path), but also all the “bad” responses as well
  - **Optional** is a partial solution - can return a response or no response, but doesn’t contain any info for the “bad” scenarions (like error message, cause…)
  - No other such types in Java:
    - We may create our own
    - We may use libraries wich define such types

Functional Java with VAVR

- **VAVR** (formerly Javaslang) - a functional library for Java 8+
  - *check out the very nice FP intro: http://www.vavr.io/vavr-docs/#_introduction*
  - strongly inspired by Scala (and other FP languages)

- Provides many **standard FP constructs**:
  - control structures: pattern matching, etc..
  - more functional interfaces (Function1..Function8)
  - persistent data structures - truly immutable, safe and fast
  - wrapper types: Tuples, **Try**, Either, Option (enhanced Optional)...

- It's a good intro and tool for **more advanced FP in Java** - expanding your thinking, bringing many of the FP benefits to your code, while still in Java
  - *it does prepares you for Scala, Kotlin, Haskel.. maybe also for future Java? :)*

- Production ready, used by many projects
  - *But get your team & boss onboard before deciding to use it :) …*

Exception handling with Try

- VAVR: **Try:**
  - Docs: *“a monadic container type which represents a computation that may either result in an exception, or return a successfully computed value”*
  - In English/Java: Try<T> - a wrapper type similar to Optional from Java, but which also may contain details of an Exception, for the failure case

- A more generic wrapper type also exists: Either<S,F> - keeps 2 values

- Yet more types: Tuple1..Tuple8 - wrappers which group a fixed number of values (of possible different types) which need to be passed around/processed like a unit (like: returning 2 values as a pair from a function, etc..)

Exception handling with Try

```
static Try<Integer> tryDivide(int dividend, int divisor) {
   return Try.of(() -> dividend / divisor);
}

static Try<Integer> parsePositiveInt(String s) {
   return Try.of(() -> Integer.parseInt(s))
           .filter(i -> i >= 0);
}

//...
Try<Integer> maybeResult = parsePositiveInt("bla");

maybeResult
       .onFailure(ex -> System.out.println("Failed to parse int, error:" + ex))
       .onSuccess(intVal -> System.out.println("Valid int value: " + intVal));

int resultOrDefault = maybeResult.map(i -> i * 2).getOrElse(-1);
```

Extra reading

- An easier path to functional programming in Java? https://developer.ibm.com/articles/j-java8idioms1/
- Robert C.Martin - FP: The failure of state - https://www.youtube.com/watch?v=7Zlp9rKHGD4
- Jenkov: Java FP - http://tutorials.jenkov.com/java-functional-programming/index.html
- FP patterns in Java 8 - https://dzone.com/articles/functional-programming-patterns-with-java-8
- Functional exception handling in Java (presentation only, also about Vavr) https://pivovarit.github.io/talks/functional-exception-handling/#/
- VAVR library - FP intro, user guide - http://www.vavr.io/vavr-docs/#_introduction
- Guide to Stream groupingBy collector - https://dzone.com/articles/the-ultimate-guide-to-the-java-stream-api-grouping