Modern Java

Java 8, 9, 10,11, 12, 13, and 14: what’s happening?

Version	Release date	End of Free Public Updates[5][6]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	**January 2019 for Oracle (commercial)** December 2020 for Oracle (personal use) At least May 2026 for AdoptOpenJDK At least June 2023[7] for Amazon Corretto	December 2030
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	At least August 2024[7] for Amazon Corretto October 2024 for AdoptOpenJDK	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA

Legend: Old version | Older version, still maintained | **Latest version** | Future release

changes enable you to write programs more easily & with good performance

section-1

A well-known problem in software engineering is that no matter what you do, **user requirements will change**

Pattern :
Behavior parameterization

Behavior parameterization is a software development pattern that lets you handle frequent requirement changes.

Behavior parameterization

For example, if you process a collection, you may want to write a method that

- Can do “something” for every element of a list
- Can do “something else” when you finish processing the list
- Can do “yet something else” if you encounter an error

Coping with changing requirements

Farm-inventory application

First attempt:
filtering green apples

Second attempt:
parameterizing the color

Third attempt:
filtering with every attribute you can think of

Behavior parameterization

Fourth attempt:
filtering by abstract criteria

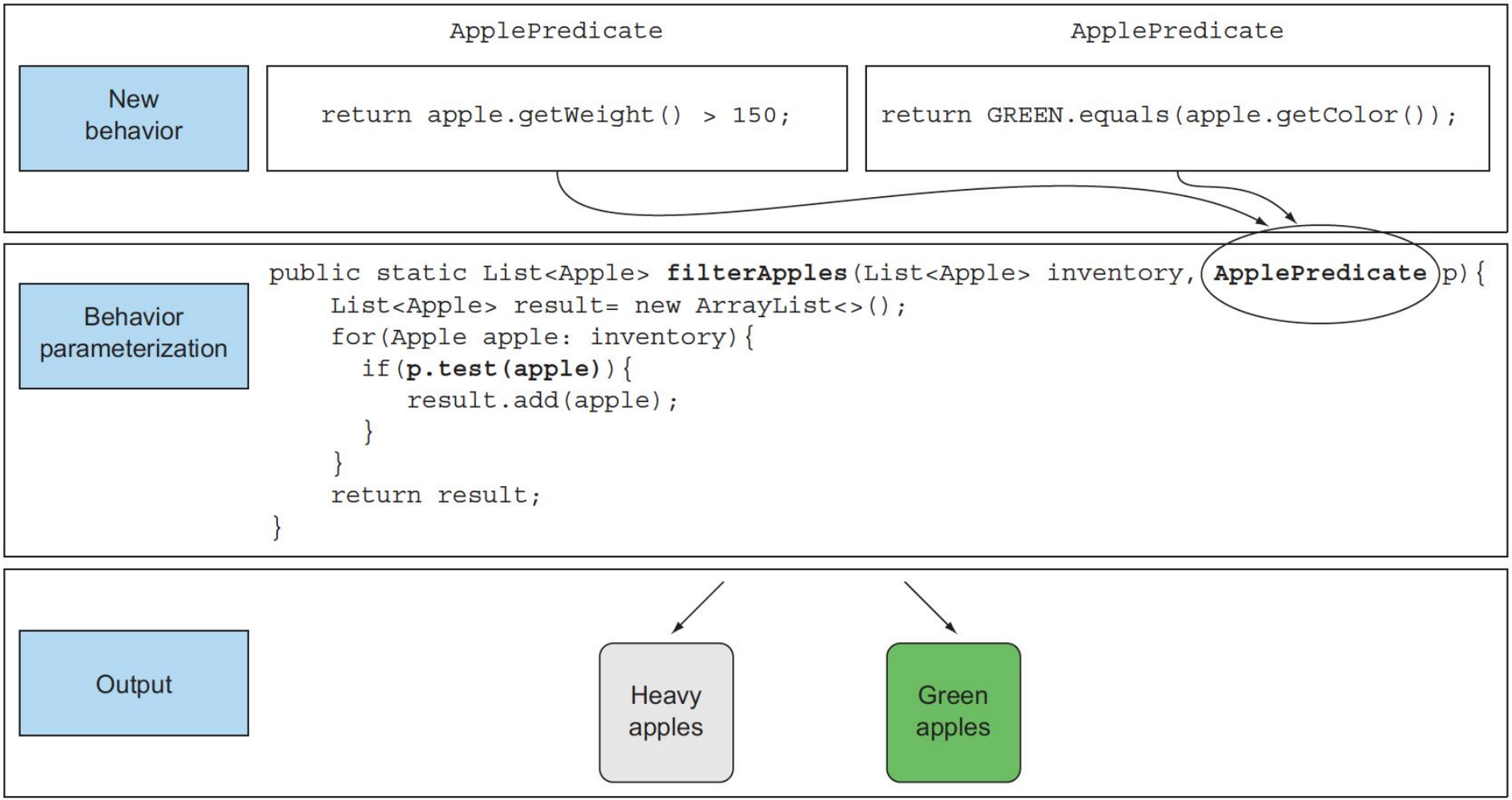
`ApplePredicate` object

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
   public boolean test(Apple apple){
      return RED.equals(apple.getColor())
             && apple.getWeight() > 150;
   }
}
```

Pass as argument

```
filterApples(inventory,          );
```

Pass a strategy to the filter method: filter the apples by using the boolean expression encapsulated within the ApplePredicate object. To encapsulate this piece of code, it is wrapped with a lot of boilerplate code (in bold).


ApplePredicate
ApplePredicate
New behavior
return apple.getWeight() > 150;
return GREEN.equals(apple.getColor());
Behavior parameterization
public static List<Apple> filterApples(List<Apple> inventory, ApplePredicate p){
List<Apple> result= new ArrayList<>();
for(Apple apple: inventory){
if(p.test(apple)){
result.add(apple);
}
}
return result;
}
Output
Heavy apples
Green apples

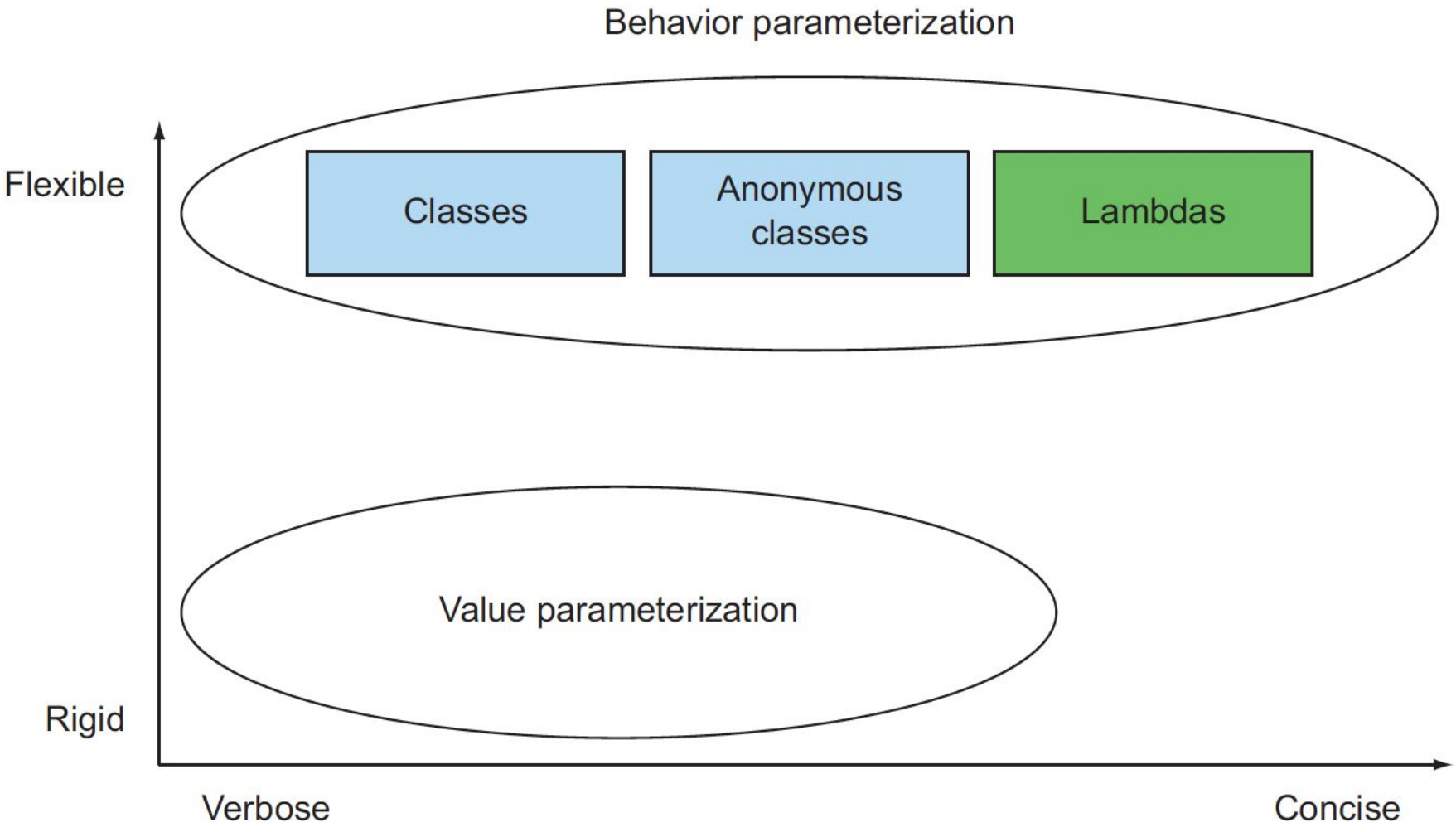
Quiz : Write a flexible prettyPrintApple method

Tackling verbosity

Fifth attempt:
Using an Anonymous class

Verbosity in general is bad

Sixth attempt:
using a lambda expression


Behavior parameterization
Flexible
Classes
Anonymous classes
Lambdas
Value parameterization
Rigid
Verbose
Concise

Seventh attempt:
abstracting over List type

Real-world examples

Behavior parameterization with Lambdas aka
FP
a.k.a
Strategy Design Pattern
a.k.a
Runtime Polymorphism

Is the ability for a method to take multiple different behaviors as parameters and use them internally to accomplish different behaviors

section-2

Java 8 :
Lambda expressions

Lambda Expression

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.

- *Anonymous*—We say *anonymous* because it doesn't have an explicit name like a method would normally have; less to write and think about!
- *Function*—We say *function* because a lambda isn't associated with a particular class like a method is. But like a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.
- *Passed around*—A lambda expression can be passed as argument to a method or stored in a variable.
- *Concise*—You don't need to write a lot of boilerplate like you do for anonymous classes.

Why should you care about lambda expressions?

lambda expression,
let you pass code in a concise way.

Where exactly can you use lambdas?

You can use a lambda expression in the context of a **functional interface.**

Functional Interface is an interface that specifies exactly one abstract method.

Lambda Expression is
an instance of a concrete
implementation of the
functional interface

In other words, treat the whole
expression as an instance of a
functional interface

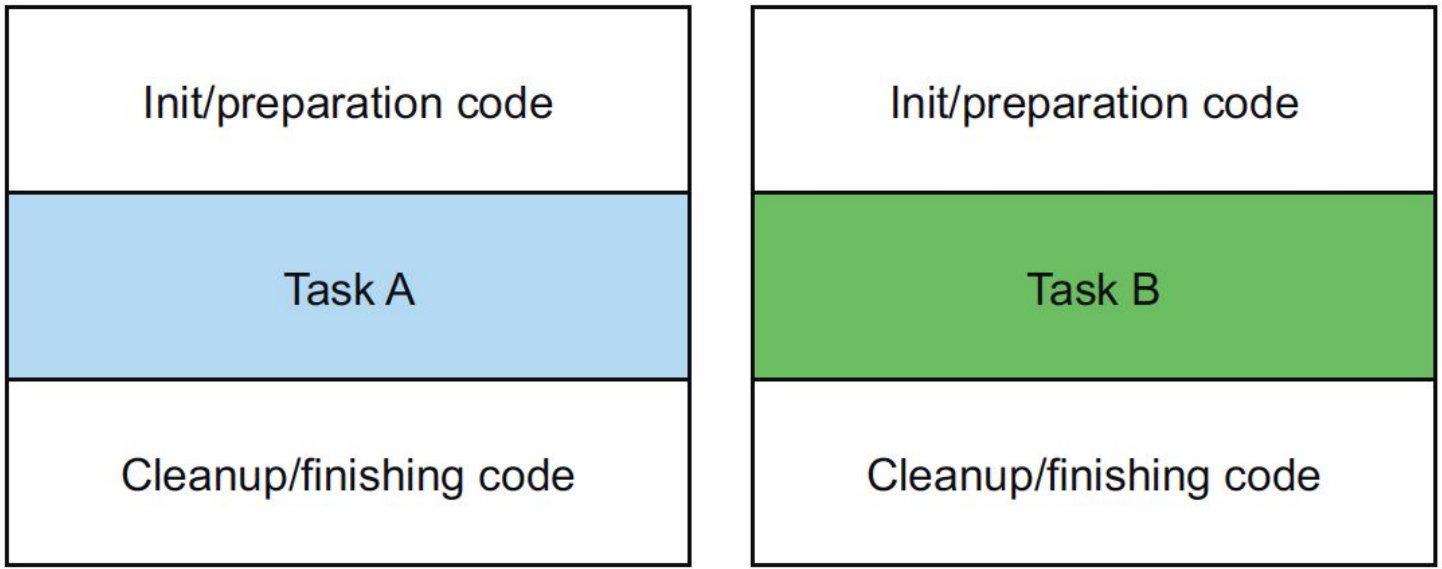
Function Descriptor

The signature of the abstract method of the functional interface describes the signature
of the lambda expression

Putting lambdas into practice: **the execute-around pattern**

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

This is the line that does useful work.



1. Step 1: Remember behavior parameterization
2. Step 2: Use a functional interface to pass behaviors
3. Step 3: Execute a behavior!
4. Step 4: Pass lambdas

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

1

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

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

2

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

3

```
String oneLine = processFile((BufferedReader br) ->
                                  br.readLine());

String twoLines = processFile((BufferedReader br) ->
                                  br.readLine + br.readLine());
```

4

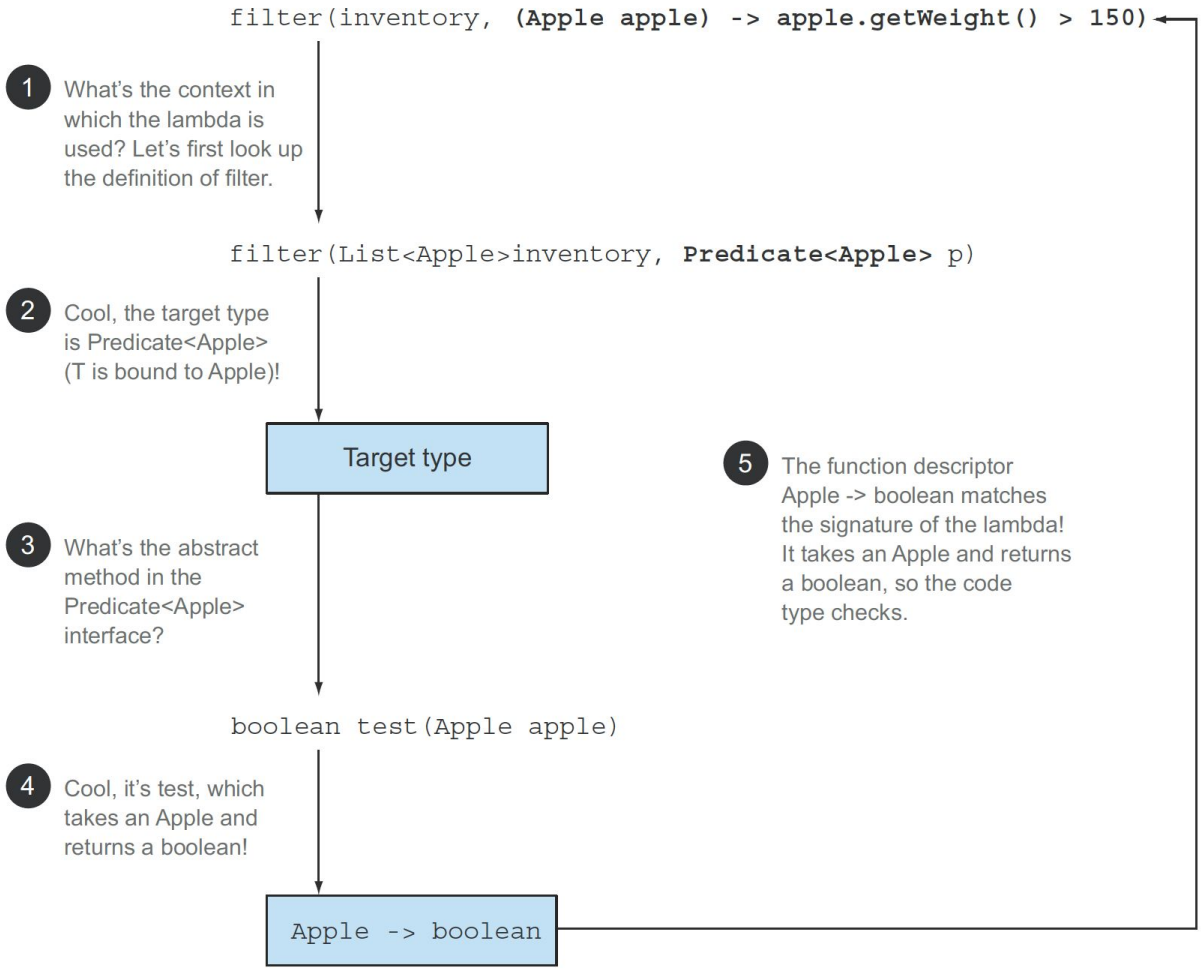
Using functional interfaces

java.util.function.*

Functional interface	Predicate<T>	Consumer<T>
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<T, U>	(T, U) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

Use case	Example of lambda	Matching functional interface
A boolean expression	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
Creating objects	`() -> new Apple(10)`	`Supplier<Apple>`
Consuming from an object	`(Apple a) -> System.out.println(a.getWeight())`	`Consumer<Apple>`
Select/extract from an object	`(String s) -> s.length()`	`Function<String, Integer>` or `ToIntFunction<String>`
Combine two values	`(int a, int b) -> a * b`	`IntBinaryOperator`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`	`Comparator<Apple>` or `BiFunction<Apple, Apple, Integer>` or `ToIntBiFunction<Apple, Apple>`

Type Checking

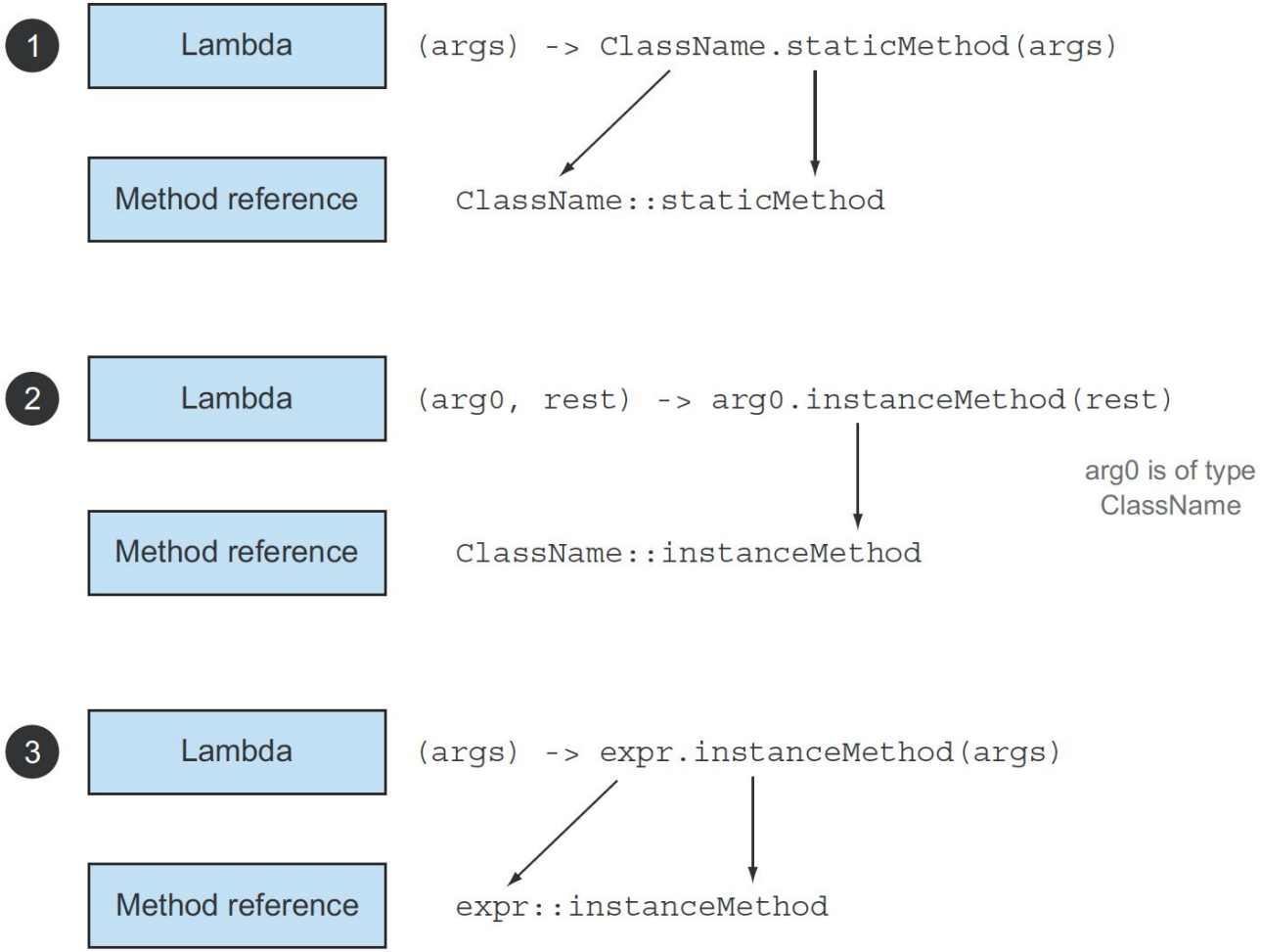

filter(inventory, (Apple apple) -> apple.getWeight() > 150)
1
What's the context in which the lambda is used? Let's first look up the definition of filter.
filter(List<Apple>inventory, Predicate<Apple> p)
2
Cool, the target type is Predicate<Apple> (T is bound to Apple)!
Target type
5
The function descriptor Apple -> boolean matches the signature of the lambda! It takes an Apple and returns a boolean, so the code type checks.
3
What's the abstract method in the Predicate<Apple> interface?
boolean test(Apple apple)
4
Cool, it's test, which takes an Apple and returns a boolean!
Apple -> boolean

Type Inference

Using Local Variables

Method References

Lambda	Method reference equivalent
`(Apple apple) -> apple.getWeight()`	`Apple::getWeight`
`() -> Thread.currentThread().dumpStack()`	`Thread.currentThread()::dumpStack`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> System.out.println(s)`	`System.out::println`
`(String s) -> this.isValidName(s)`	`this::isValidName`


1
Lambda
(args) -> ClassName.staticMethod(args)
Method reference
ClassName::staticMethod
2
Lambda
(arg0, rest) -> arg0.instanceMethod(rest)
arg0 is of type
ClassName
Method reference
ClassName::instanceMethod
3
Lambda
(args) -> expr.instanceMethod(args)
Method reference
expr::instanceMethod

Putting lambdas and method references into practice

Step 1: Pass code

Step 2: Use an anonymous class

Step 3: Use lambda expressions

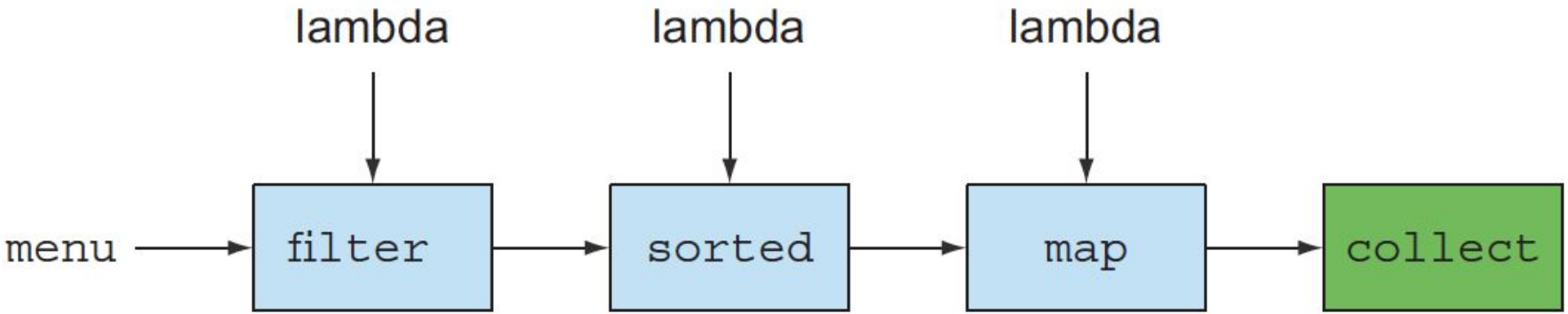
Step 4: Use method references

Useful methods to compose lambda expressions

Stream API

Stream API

Java API that let you manipulate collections of data in a declarative way


lambda
lambda
lambda
menu
filter
sorted
map
collect

Streams API in Java 8 lets you write code that’s

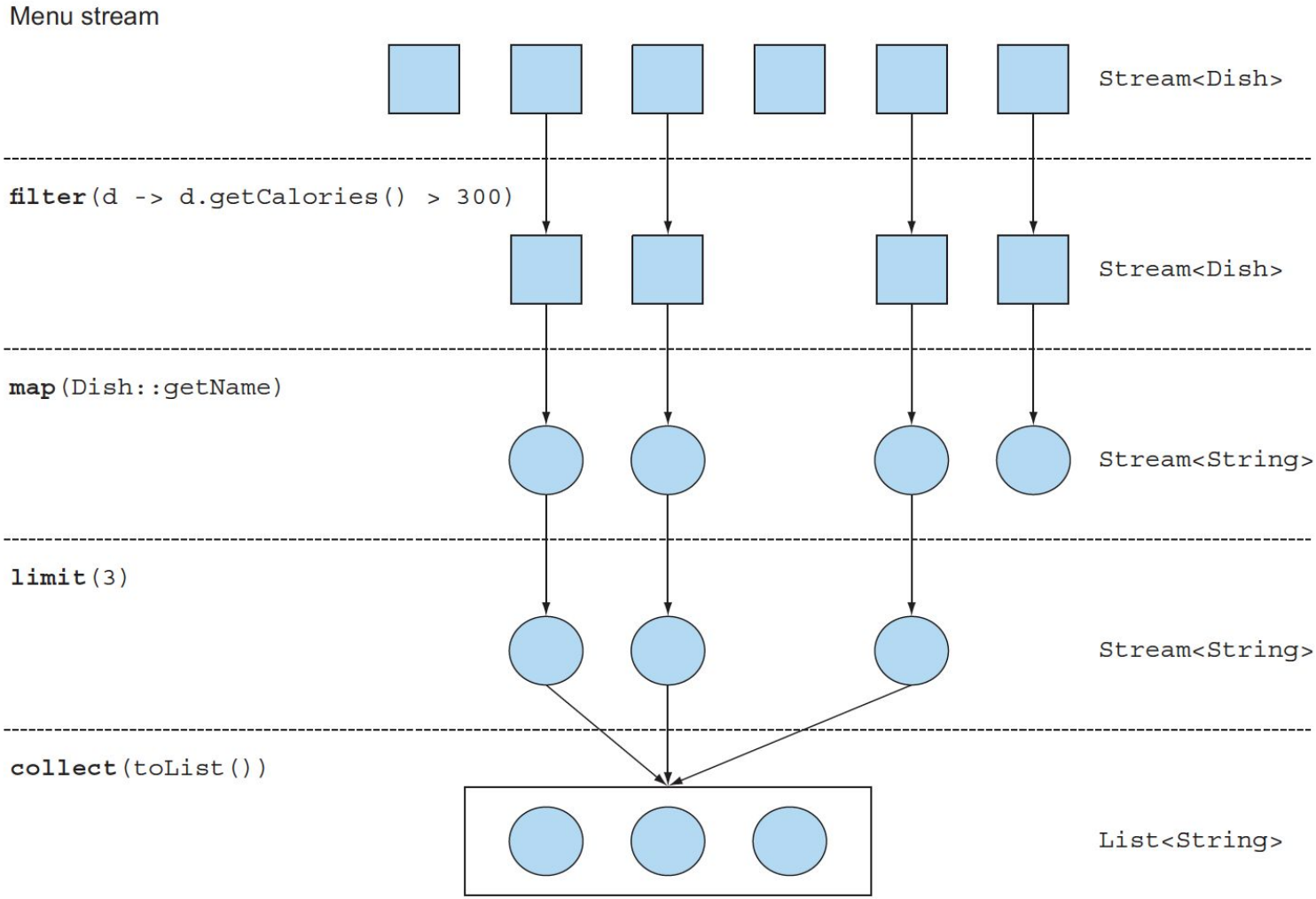
Declarative—More concise and readable
Composable—Greater flexibility
Parallelizable—Better performance

what exactly is a stream?

A short definition is
“a sequence of elements from a source that supports data-processing operations.”

stream operations have two important characteristics

- Pipelining
- Internal iteration


Menu stream
Stream<Dish>
filter(d -> d.getCalories() > 300)
Stream<Dish>
map(Dish::getName)
Stream<String>
limit(3)
Stream<String>
collect(toList())
List<String>