Java Core OCP Part I

https://github.com/thimotyb/java8.git

Collections, Generics, Concurrency

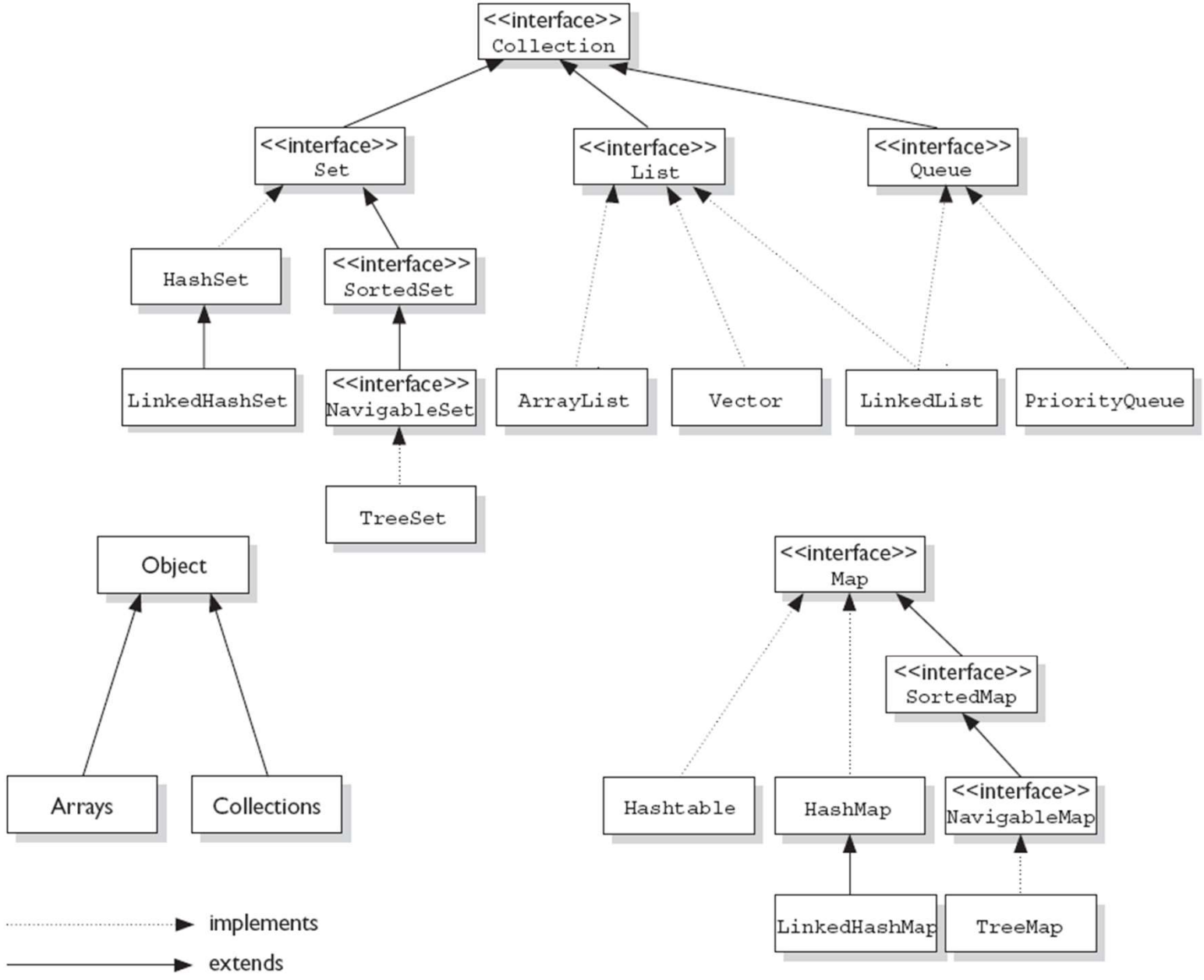
Collections - Overview

Interfaces

Collection	Set	SortedSet
List	Map	SortedMap
Queue	NavigableSet	NavigableMap

Classes (Sorted/Unsorted, Ordered/Unordered)

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				


<<interface>>
Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
HashSet
<<interface>>
SortedSet
LinkedHashSet
<<interface>>
NavigableSet
ArrayList
Vector
LinkedList
PriorityQueue
TreeSet
Object
Arrays
Collections
<<interface>>
Map
<<interface>>
SortedMap
Hashtable
HashMap
<<interface>>
NavigableMap
LinkedHashMap
TreeMap
implements
extends

Collections - List interface

- A **List** has an index
- **ArrayList**: fast iteration and fast random access. Ordered not Sorted. Use for fast iteration with no need to insert/delete.
- **Vector**: Like ArrayList, but synchronized for thread safety. Use only if necessary because of performance hit.
- **LinkedList**: ordered by index with elements doubly linked. Use for fast insertion/deletion. Slow in iteration.

Collections - Set interface

- A **Set** does not allow duplicates
- **HashSet**: unsorted, unordered Set. Uses the object hashCode, needs an efficient hash implementation.
- **LinkedHashSet**: ordered, unsorted Set. Iterates in the order of insertion.
- **TreeSet**: ordered, sorted Set. Elements are in ascending order, or you can implement your own Comparator.

Collections - Map Interface

- Map uses key, value pairs with unique keys.
- HashMap is an unsorted, unordered Map. Allows 1 null key. Allows null values.
- Hashtable: prehistoric. It is a synchronized HashMap. Does not allow null key or values.
- LinkedHashMap: Maintains insertion order.
- TreeMap: Sorted Map. Keys can implement Comparator.

Collections - Queue Interface

- A Queue is a list in FIFO order.
- Contains specific Queue methods.

Using Boxing (in Collections)

- Exercise: What happens here?

```
Integer i1 = 1000;
Integer i2 = 1000;
if(i1 != i2) System.out.println("different objects");
if(i1.equals(i2)) System.out.println("meaningfully equal");


Integer i3 = 10;
Integer i4 = 10;
if(i3 == i4) System.out.println("same object");
if(i3.equals(i4)) System.out.println("meaningfully equal");
```

Collections - Diamond Syntax

- Collection type can be inferred with Diamond
- This is OK:

```
ArrayList<String> stuff = new ArrayList<>()
List<Dog> myDogs = new ArrayList<> () ;
Map<String, Dog> dogMap = new HashMap<>();
```

This is KO:

```
ArrayList<> stuff = new ArrayList<String>()
```

Collections - Sorting with Comparator

- Comparable: Used by Collections.sort() and java.util.Arrays.sort() to sort Lists and arrays
- Comparator: Used to sort any class in any collection
- Must implement compare():
  - return negative if thisObject < anotherObject
  - Zero if thisObject == anotherObject
  - Positive if thisObject > anotherObject
- A sorted collection can be searched with binarySearch(). A collection must be sorted first to be binary searchable.

EXERCISE: Create a DVD Object with Title, Genre, Price. Put some of them in an ArrayList. Create a Comparator which orders by Genre, Ascending. Use the Comparator to Sort the List. Perform a Binary Search.

Collections and Generics - Polymorphism

- Consider this:
  - List<Integer> myList = new ArrayList<Integer>();
- This is legal as ArrayList implements List

- Consider this:
  - class Parent { }
  - class Child extends Parent { }
  - List<Parent> myList = new ArrayList<Child>();

EXERCISE: Try if this works

Generics - Polymorphism

- Polymorphism in Collections with Generics does not work as in Arrays
- Exercise: Try Rewriting AnimalDoctor with a Typed ArrayList
  - Can you pass an ArrayList<Dog> into an ArrayList<Animal> argument parameter?
  - Can you add a Dog into an ArrayList<Animal>?
- Generics have Erasure
  - The Typing in generics does not exist anymore at runtime
  - At runtime ALL collection code (legacy and typed) looks exactly like the pregeneric version
  - This is done to support Legacy use of Collections (without Generics)

Generics - Polymorphism

- The reason it is dangerous to pass a collection of a subtype into a method that takes a collection of a supertype is because you might add something wrong.

```
public void foo() {
        Cat[] cats = {new Cat(), new Cat()};
        addAnimal(cats); // no problem, send the Cat[] to the method
}
public void addAnimal(Animal [] animals) {
        animals [0] = new Dog(); // Eeek.We just put a Dog  in a Cat array!
}
```

- It compiles for Arrays (doesn't work at Runtime); it does not compile for ArrayList and Generics, because Arrays have ArrayStoreException, while Collections have their type Erased at runtime

Generics - Using ? Wildcard

- The problem is because the Method is adding potentially wrong elements
- We can specify that the method will be read-only by using the ? Wildcard
  - public void readAnimal(List<? extends Animal> animals)
  - This can access elements from the collection, but cannot add.
  - ? extends refers to a superclass or an interface, without distinction
- You can use ? wildcard and ADD to the collection by using ? super
  - public void addAnimal(List<? super Dog> animals) {
    - animals.add(new Dog()); // you can add a Dog with super

EXERCISE: See Example 2

Generics - Declarations

- We can use generic types in our own classes and methods
- This is the same mechanism used to define Collections with Generics

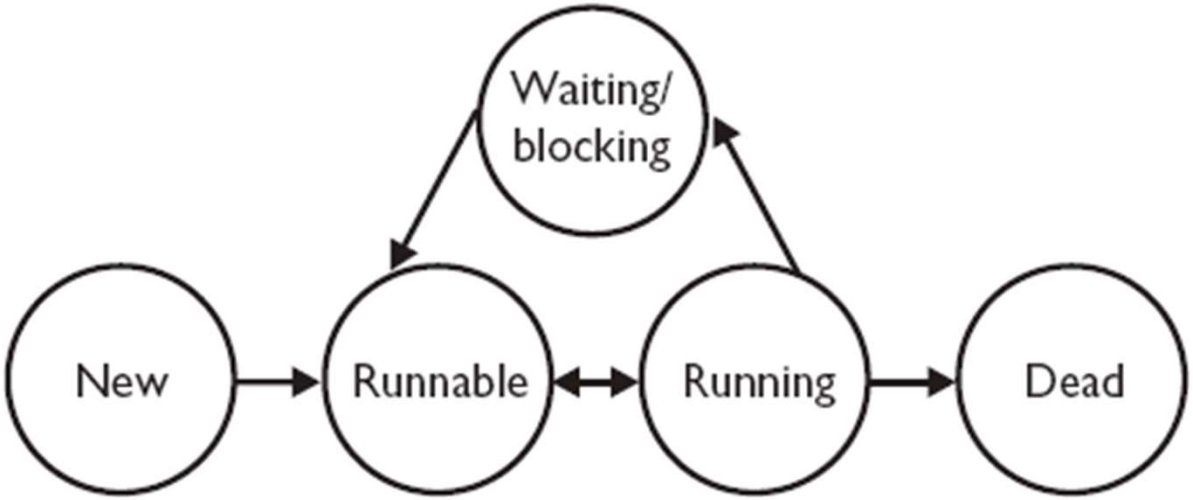
```
public class TestGenerics<T> {      // as the class type
  T anInstance;                     // as an instance variable type
  T [] anArrayOfTs;                 // as an array type

  TestGenerics(T anInstance) {      // as an argument type
     this.anInstance = anInstance;
  }
  T getT() {                        // as a return type
     return anInstance;
  }
}
```

```
public <T> void makeArrayList(T t)

public <T extends Number> void makeArrayList(T t)
```

Threads

- The Thread class is used to start and managing Threads
  - run(); start(); sleep(); yield()
- You can use the Runnable interface to start a Thread (Example 1)
- Order execution is not guaranteed (Example 2) across thread (it is within the same thread)
- Each thread starts and runs to completion (of the run method)
- Once a thread has been started, it can never be started again. (call start() again)

Thread States



New: start() not called
Runnable: start() called and eligible to run
Running: run() is running
Waiting/Blocking/Sleeping: wait() called, or blocked for I/O or lock, or sleep() called
Dead: run() has completed

Threads - sleep(), yield(), join()

- Thread.sleep(millis). Another thread might Interrupt.
  - After sleep, thread gets back to Runnable, not to Running (sleep time might not be exact)
- Threads run with priorities (1-10). yield() puts back the thread to **Runnable (not Waiting)** to allow other threads with the same priority to get into Running.
- Non-static join() allows one thread to wait until the end of another thread.

```
Thread t = new Thread();
t.start();
t.join(); // this thread (main) waits until t completes
```

- EXERCISE: Create some Threads and experiment with sleep() and join()

Threads - Synchronization

- Synchronization is needed when different threads access the same method/object/field in a common class, to avoid race conditions (see Race Example)
- To avoid race conditions, operations which need consistency need to be made atomic (e.g. checking balance and withdrawal are atomic)
- You can't guarantee that a single thread will stay running throughout the entire atomic operation.
- However, you can guarantee that even if the thread running the atomic operation moves in and out of the running state, no other running thread will be able to act on the same data.

Threads - Recipe for Synchronization

- *Mark the shared variables across Threads as private.*
- *Synchronize the code that modifies the variables. (synchronize)*

EXERCISE: Correct Race condition adding synchronize to makeWithdrawal and run again. Compare the outputs.

- Synchronization works with Locks. Every object has a Lock which is used in synchronized blocks.
- When we enter a synchronized non-static method, we automatically acquire the lock associated with the current instance of the class whose code we're executing (the *this* instance).

Threads - Synchronization with object lock

- Only methods (or blocks of code) can be synchronized, not variables or classes.
- Each object has just one lock.
- Not all methods in a class need to be synchronized. A class can have both synchronized and non-synchronized methods.
- If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call.
- If a class has both synchronized and non-synchronized methods, multiple threads can still access the other methods
- If a thread goes to sleep, it holds any locks it has
- A thread can acquire more than one lock (on different objects)

Threads - Exercise

In this exercise we will attempt to synchronize a block of code. Within that block of code we will get the lock on an object, so that other threads cannot modify it while the block of code is executing. We will be creating three threads that will all attempt to manipulate the same object. Each thread will output a single letter 100 times, and then increment that letter by one. The object we will be using is StringBuffer.

We could synchronize on a String object, but strings cannot be modified once they are created, so we would not be able to increment the letter without generating a new String object. The final output should have 100 As, 100 Bs, and 100 Cs all in unbroken lines.

Threads - Exercise

1. Create a class and extend the Thread class.

2. Override the run() method of Thread. This is where the synchronized block of code will go.

3. For our three thread objects to share the same object, we will need to create a constructor that accepts a StringBuffer object in the argument.

4. The synchronized block of code will obtain a lock on the StringBuffer object from step 3.

5. Within the block, output the StringBuffer 100 times and then increment the letter in the StringBuffer.

6. Finally, in the main() method, create a single StringBuffer object using the letter A, then create three instances of our class and start all three of them.

Threads - Deadlock

- Deadlock occurs when two threads are blocked, with each waiting for the other's lock.
- Example 7: There is a very small chance that it deadlocks
- Deadlocks can be usually resolved by changing lock order.
- See Example 7 and propose an order change to the lock to avoid the deadlock.

Threads - Interaction

- Threads can interact with one another to communicate about their locking status. The **Object class** has three methods, wait(), notify(), and notifyAll() that help threads communicate
- wait(), notify(), and notifyAll() must be called from within a synchronized context. A thread can't invoke a wait or notify method on an object unless it owns that object's lock.
- In the same way that every object has a lock, every object can have a list of threads that are waiting for a notification from the object. A thread gets on this waiting list by executing wait() on the target object. From that moment, it doesn't execute any further instructions until the notify() method of the target object is called. If many threads are waiting on the same object, only one will be chosen (in no guaranteed order)

Concurrency

- Creating concurrent programming with barebone Threads can be challenging and error prone: java.util.concurrent provides a tool-kit for robust creation of concurrent applications
- util.concurrent provides a more flexible way to use Threads mechanism:
  - Atomic Variables
  - Locks (Lock, ReadWriteLock, ReentrantLock)
  - Concurrent Collections
  - Executors and ThreadPools
  - Parallel Fork/Join Framework

Concurrency - Atomic Variables

- When threads work on a common field variable, accessing the variable value and incrementing (modifying) it are not atomic operations, they should be manually synchronized (see Example 1)
- Or the variable could be wrapped as Atomic
  - for an int, use AtomicInteger and use its methods (e.g. getAndIncrement()) to modify it
- EXERCISE: replace the shared integer with AtomicInteger in Example 1
- There are Atomic wrappers and operations for Integer, Long, Double
- General wrapper for Objects: AtomicReference updates atomically a referenced object

Concurrency - Locks

- the Locks package allows to create more sophisticated Locks:
  - Obtain a lock in one method and release it in another
  - Multiple wait/notify pool. Threads can select which pool to wait on with Condition
  - Ability to try and acquire a lock and specify an alternate action if unsuccessful

```
Object obj = new Object();
synchronized(obj) {    // traditional locking, blocks until acquired
                       // work
}                      // releases lock automatically
```

No Alternate

```
Lock lock = new ReentrantLock();
lock.lock();          // blocks until acquired
try {
    // do work here
} finally {           // to ensure we unlock
  lock.unlock();      // must manually release
}
```

With try

```
Lock lock = new ReentrantLock();
boolean locked = lock.tryLock(); // try without waiting
if (locked) {
  try {
      // work
  } finally {                    // to ensure we unlock
    lock.unlock();
  }
}
```

Concurrency - Concurrent Collections

- The java.util.concurrent package provides several types of collection that are thread-safe, without using coarse-grained synchronization (see Example 2)
- It is recommended to use these Concurrent Collections when you develop collections that are maintained by multiple threads
- One of the basic mechanisms is CopyOnWrite: every time the collection is altered, a new immutable copy is created (this guarantees thread safety)
  - They can be scanned only via iterator (for-each), as the iterator is bound to the original copy
  - They can be used only when the Collection undergoes only minimal changes, otherwise the performance hit is unacceptable

Concurrency - Concurrent Collections

- ConcurrentHashMap; ConcurrentSkipListSet
  - Provide a safe-thread mechanism which is not CopyOnWrite to solve the performance problem.
  - Iterator is weakly consistent (it may point to the original list or to a later version of the list)
- Blocking Queues
  - A useful system to transfer data between threads in a safe manner
  - Based on producer-consumer scenario, using the concept of a Queue
  - Allows to consume objects from a destination thread with order guarantee (see Example)

Concurrency - Executors and Thread Pools

- Executors (and the ThreadPools used by them) help meet two of the same needs that Threads do:
  - Creating and scheduling some Java code for execution and
  - Optimizing the execution of that code for the hardware resources you have available (using all CPUs, for example). Executor helps to adjust the optimal number of threads we want to run on the current OS to get optimum performance.
  - Introduces the concept of Task, which can be run on one (or more) Threads

```
Runnable r = new MyRunnableTask();
ExecutorService ex = Executors.newCachedThreadPool(); // subtype of Executor
ex.execute(r);
```

FixedThreadPool executor is more common - you emipirically find a fixed optimal number of threads to use to execute the task.

Concurrency - Executors with Callable

- Unlike the Runnable interface, a Callable may return a result upon completing execution and may throw a checked exception.
- An ExecutorService can be passed a Callable instead of a Runnable.
- The primary benefit of using a callable is the ability to return a result.
- Because an ExecutorService may execute the Callable asynchronously (just like a Runnable), Future is used to obtain the status and result of a Callable.
- Future.get() waits for task to get completed from the Executor, greatly simplifying synchronization when consuming the results (Example 3)

Concurrency - Fork/Join Framework

- With Fork/Join, instead of running several parallel small tasks, we parallelize a single big task (e.g. Map/Reduce)
- The Fork-Join ExecutorService implementation is java .util. concurrent . ForkJoinPool.
- You will typically submit a single task to a ForkJoinPool and await its completion.
- The ForkJoinPool and the task itself work together to divide and conquer the problem. Any problem that can be recursively divided can be solved using Fork/Join.
- With the Fork-Join Framework, a java.util.concurrent.ForkJoinTask instance is created to represent the task that should be accomplished.
- RecursiveTask is used if no result has to be returned.

Concurrency - Fork/Join Framework

```
class ForkJoinPaintTask {
  compute() {
    if(isFenceSectionSmall()) {  // is it a manageable amount of work?
      paintFenceSection();       // do the task
    } else {                     // task too big, split it
      ForkJoinPaintTask leftHalf = getLeftHalfOfFence();
      leftHalf.fork();           // queue left half of task
      ForkJoinPaintTask rightHalf = getRightHalfOfFence();
      rightHalf.compute();       // work on right half of task
      leftHalf.join();           // wait for queued task to be complete
    }
  }
}
```

Java Core OCP Part 2

https://github.com/thimotyb/Java8InAction.git

Functional Programming and More Concurrency

Agenda

- Introduction to Functional Aspects in Java
- Lambda Expressions and Functional Interfaces
- Stream API
- Parallel Data Processing
- Updates to Interfaces; Default methods
- Functional Programming Techniques; Blending OOP and FP

Why Functional Programming?

- Behavioral Parametrization is a good example of how Functional Programming can be put to use
- It means taking a block of code and making it available without executing it
- E.g on a collection:
  - 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
- The idea is turning *functions* into a passable value to methods

An Apple a Day...

- Image you want to filter green apples in your collection:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if( "green".equals(apple.getColor() ) {
            result.add(apple);
        }
    }
    return result;
}
```

An accumulator list for apples.

Select only green apples.

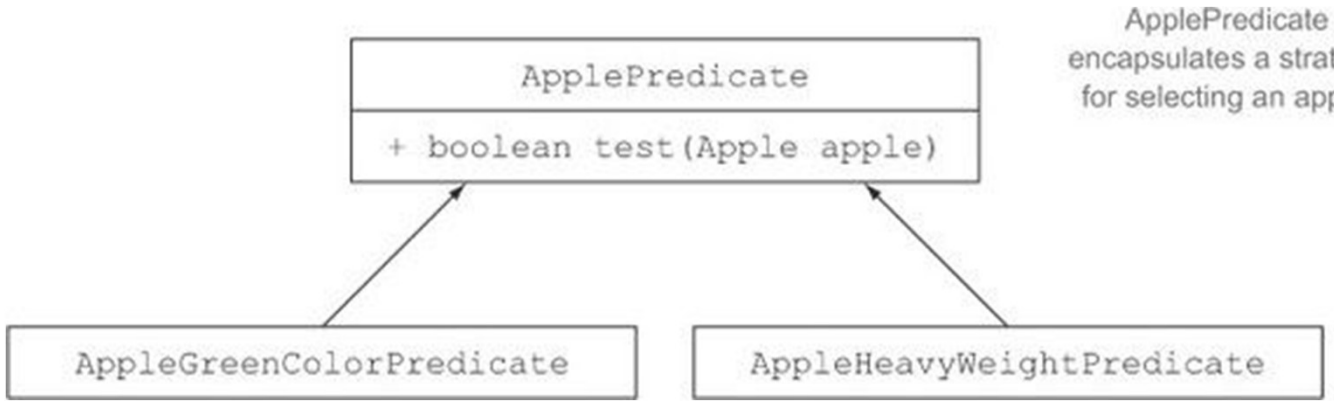
- Let's say you want to filter by parametric color and weight:

```
public static List<Apple> filterApples(List<Apple> inventory, String color,
                                       int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if ( (flag && apple.getColor().equals(color)) ||
             (!flag && apple.getWeight() > weight) ){
            result.add(apple);
        }
    }
    return result;
}
```

A really ugly way to select color or weight

Parametrize filter behavior using Strategy

- A design solution is to apply the Strategy pattern, creating an interface to encapsulate the filter behavior




```
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;
}
```

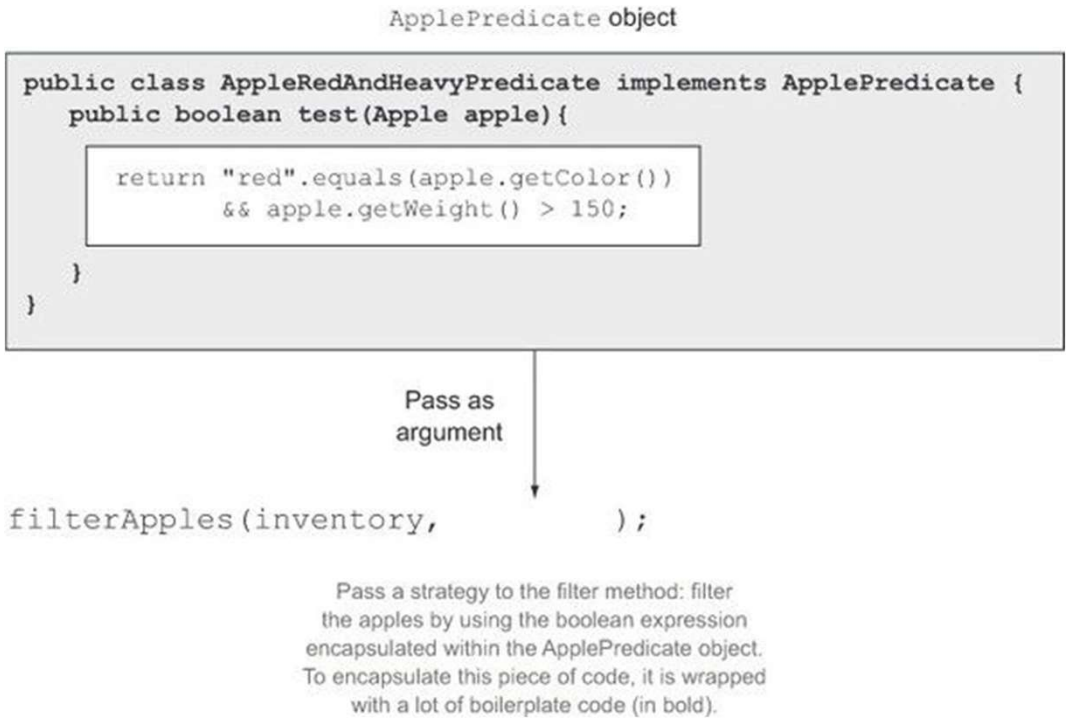
The predicate object encapsulates the condition to test on an apple.

```
public class AppleHeavyWeightPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate{
    public boolean test(Apple apple){
        return "green".equals(apple.getColor());
    }
}
```

Select only heavy apples.

Select only green apples.

Passing dynamic code as Strategy



- We can use anonymous classes to avoid creating a specific method for each behaviour

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    public boolean test(Apple apple){
        return "red".equals(apple.getColor());
    }
});
```

Parameterizing the behavior of the method `filterApples` directly inline!

Exercise: Write an Apple Formatter

Write a behaviorally generic print formatter for Apples

You can start from this code:

```
public static void prettyPrintApple(List<Apple> inventory, ???){

for(Apple apple: inventory) {

        String output = ???.???(apple);

        System.out.println(output);

} }
```

Extending the filter interfaces with generics

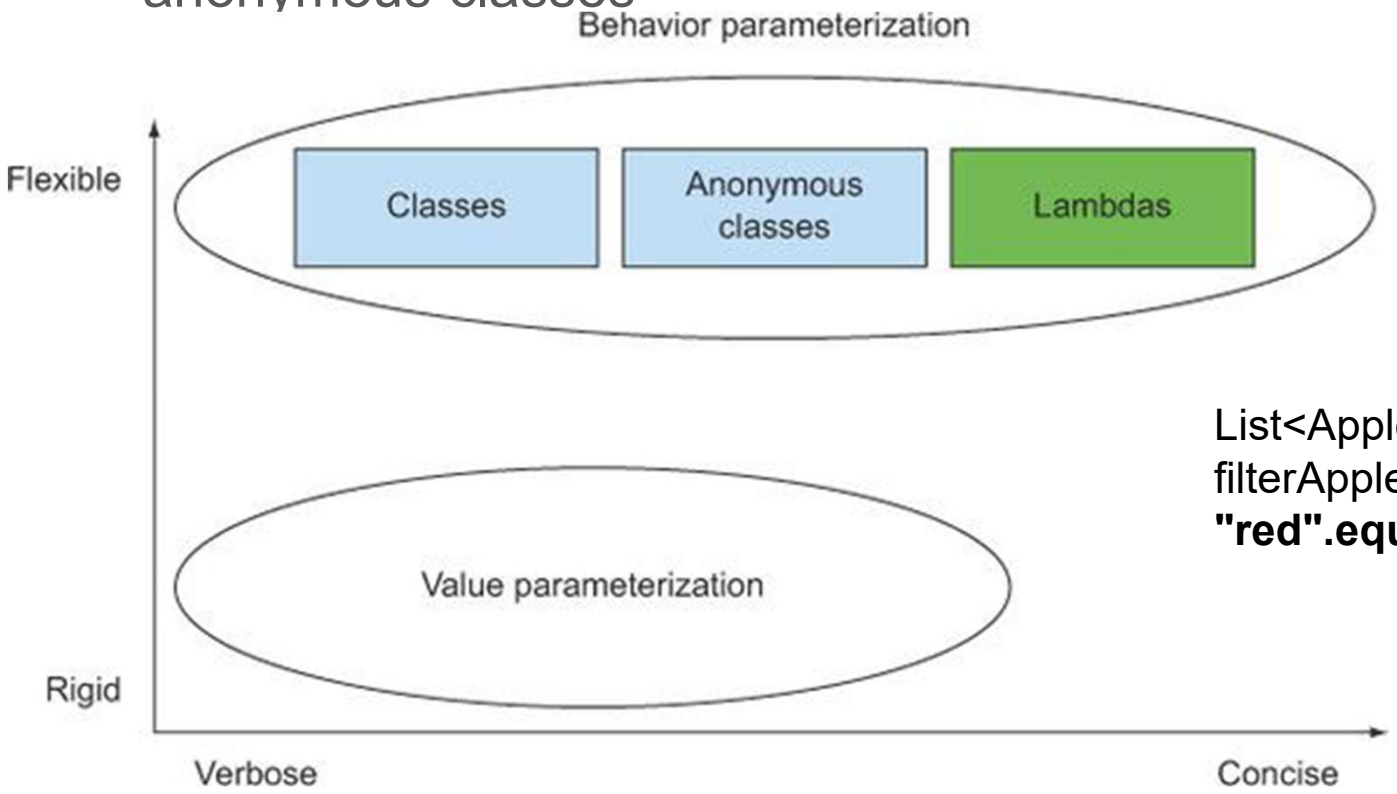
- We can further generalize from Apples by using generics to define the functional interface

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

public static <T> List<T> filter(List<T> list, Predicate<T> p){
    List<T> result = new ArrayList<>();
    for(T e: list){
        if(p.test(e)){
            result.add(e);
        }
    }
    return result;
}
```

Introducing a type parameter T

Using lambdas instead of anon classes

- Lambda function capture this idea to avoid complexity and scope problems in anonymous classes




```
List<Apple> result =
filterApples(inventory, (Apple apple) ->
"red".equals(apple.getColor()));
```

Lambda expressions

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

Lke a method, a lambda has a list of parameters, a body, a return type, and a possible list of exceptions that can be thrown.

A lambda expression can be passed as argument to a method or stored in a variable. It is concise — You don’t need to write a lot of boilerplate like you do for anonymous classes.

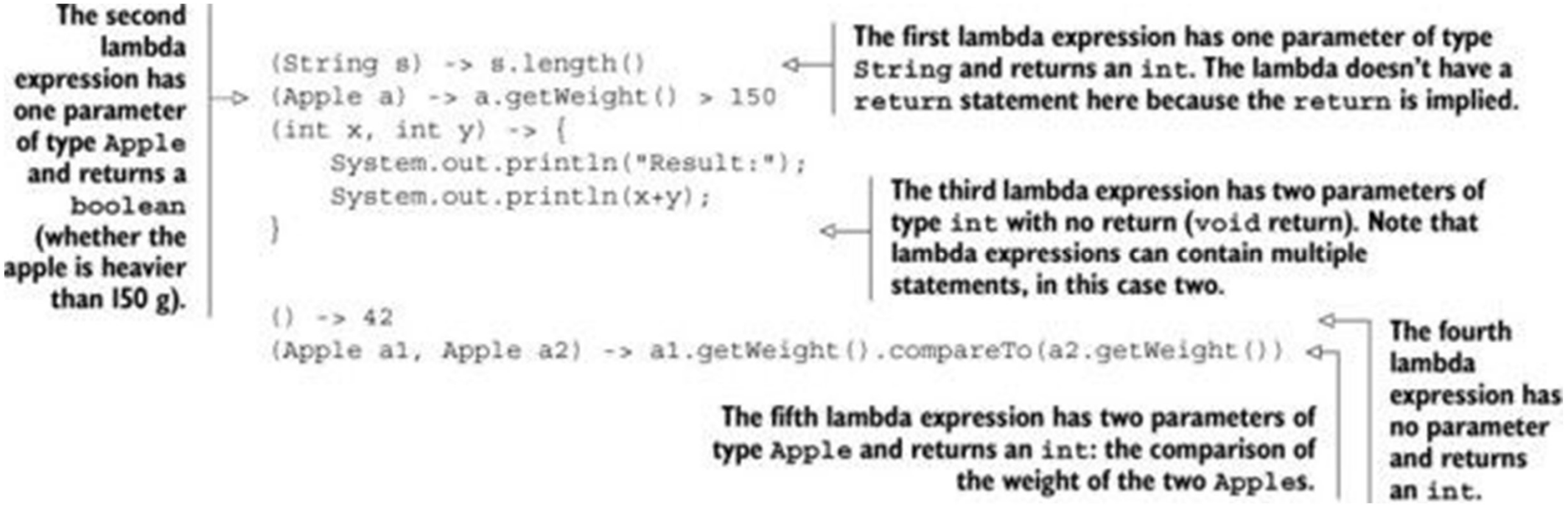
Anatomy of a Lambda Expression

A list of parameters— In this case it mirrors the parameters of the compare method of a Comparator—two Apples.

An arrow— The arrow -> separates the list of parameters from the body of the lambda.

The body of the lambda— Compare two Apples using their weights. The expression is considered the lambda's return value.

Examples

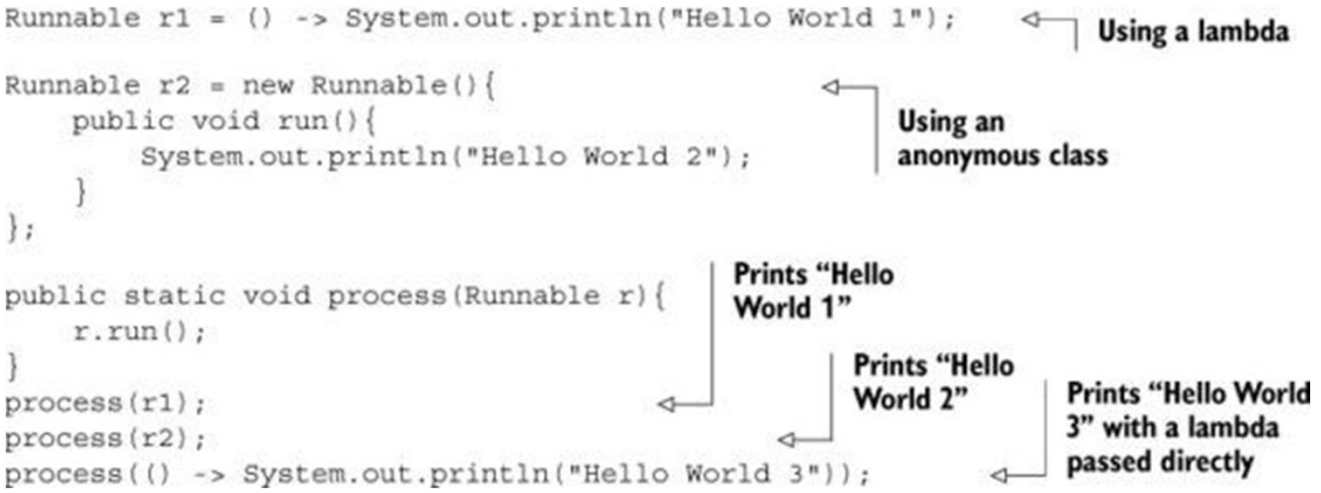

The second lambda expression has one parameter of type Apple and returns a boolean (whether the apple is heavier than 150 g).
(String s) -> s.length()
(Apple a) -> a.getWeight() > 150
(int x, int y) -> {
System.out.println("Result:");
System.out.println(x+y);
}
() -> 42
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
The first lambda expression has one parameter of type String and returns an int. The lambda doesn't have a return statement here because the return is implied.
The third lambda expression has two parameters of type int with no return (void return). Note that lambda expressions can contain multiple statements, in this case two.
The fourth lambda expression has no parameter and returns an int.
The fifth lambda expression has two parameters of type Apple and returns an int: the comparison of the weight of the two Apples.

Functional Interface

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

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

Some pre-Java 8 interfaces are functional, e.g. *Runnable*



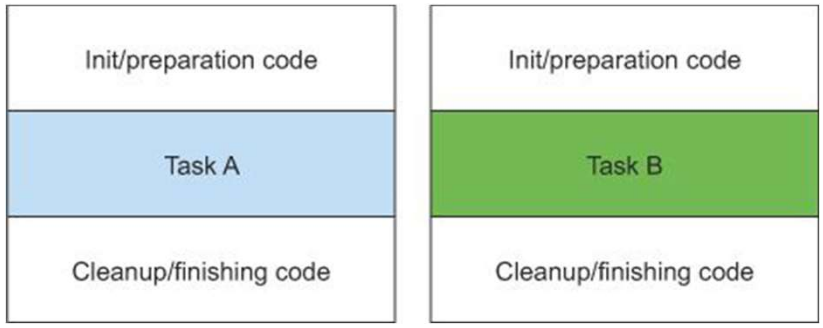
@FunctionalInterface

You can create specific interfaces with a single method

Use the @FunctionalInterface to clarify that the intent is creating a FI (not compulsory)

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

Example: Execute Around




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

This is the line that does useful work.

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

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

Processing the BufferedReader object

Reuse the processFile method and process files in different ways by passing different lambdas.

Common functional interfaces in Java 8

Functional Interfaces	# Parameters	Return Type	Single Abstract Method
Supplier<T>	0	T	get
Consumer<T>	1 (T)	void	accept
BiConsumer<T, U>	2 (T, U)	void	accept
Predicate<T>	1 (T)	boolean	test
BiPredicate<T, U>	2 (T, U)	boolean	test
Function<T, R>	1 (T)	R	apply
BiFunction<T, U, R>	2 (T, U)	R	apply
UnaryOperator<T>	1 (T)	T	apply
BinaryOperator<T>	2 (T, T)	T	apply

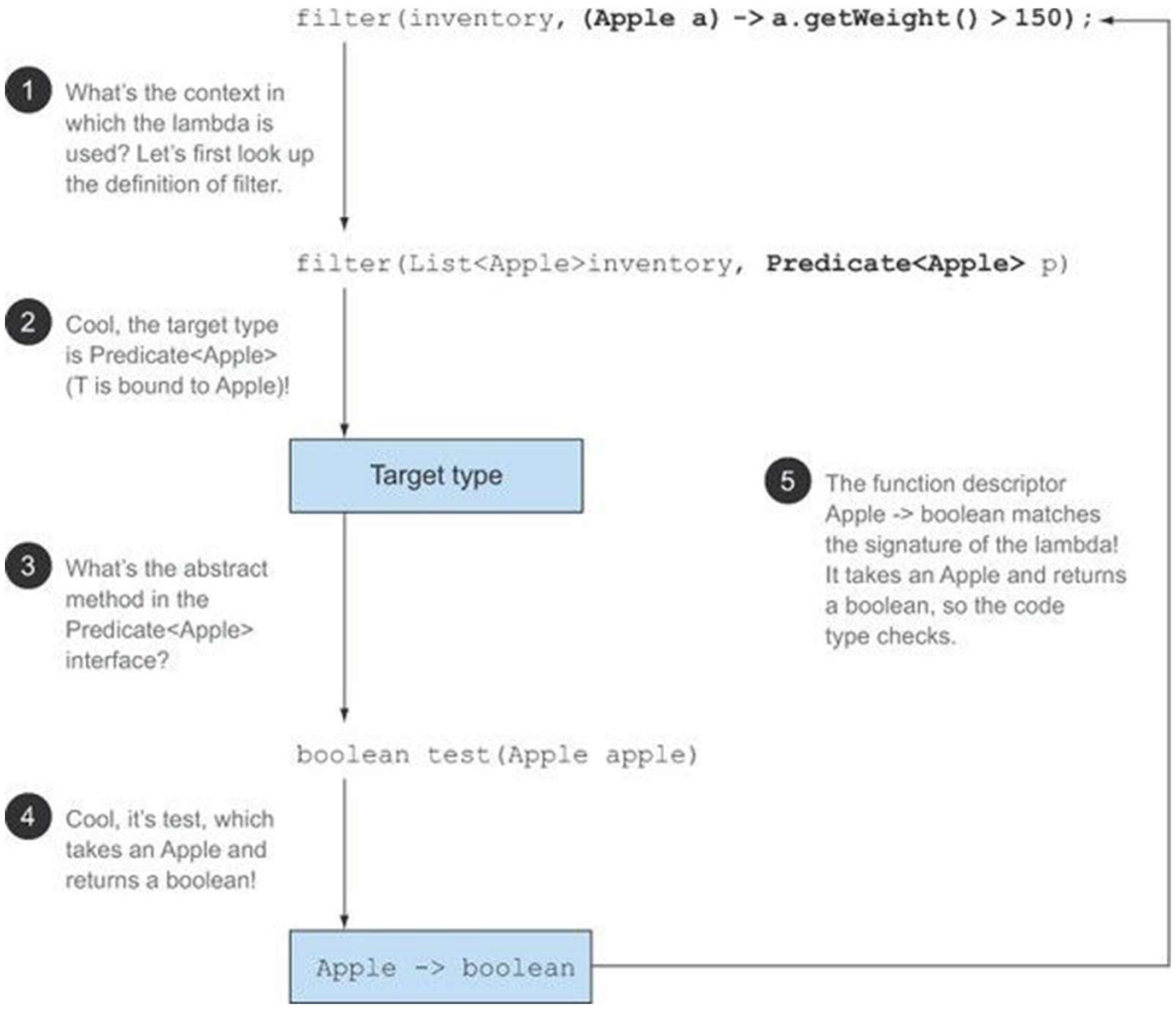
Example of Common FI: Function<T, R>

```
@FunctionalInterface
public interface Function<T, R>{
    R apply(T t);
}
public static <T, R> List<R> map(List<T> list,
                                 Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T s: list){
        result.add(f.apply(s));
    }
    return result;
}
// [7, 2, 6]
List<Integer> l = map(
                      Arrays.asList("lambdas","in","action"),
                      (String s) -> s.length()
                  );
```

The lambda is the implementation for the `apply` method of `Function`.

Type checking

The type of a lambda is deduced from the context in which the lambda is used. The type expected for the lambda expression inside the context is called the *target type*.



Lambda Target Typing

Because of the idea of target typing, the same lambda expression can be associated with different functional interfaces if they have a compatible abstract method signature.

The same lambda can therefore be used with multiple different functional interfaces:

```
Comparator<Apple> c1 = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

ToIntBiFunction<Apple, Apple> c2 = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

BiFunction<Apple, Apple, Integer> c3 = (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Special void-compatibility rule

If a lambda has a statement expression as its body, it’s compatible with a function descriptor that returns void (provided the parameter list is compatible too).

```
// 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 Java compiler deduces what functional interface to associate with a lambda expression from its surrounding context (the target type)

It can also deduce an appropriate signature for the lambda because the function descriptor is available through the target type.

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

No explicit type on the parameter a

```
Comparator<Apple> c =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Without type inference

```
Comparator<Apple> c =
  (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

With type inference

Using local variables

When lambda expressions contain references to free variables (variables that aren’t the parameters and defined in an outer scope), they’re called capturing lambdas.

Lambdas are allowed to capture (that is, to reference in their bodies) instance variables and static variables without restrictions. But local variables have to be explicitly declared final or are effectively final.

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
portNumber = 31337;
```

Error: local variables referenced from a lambda expression must be final or effectively final.

Closure

A *closure* is an instance of a function that can reference nonlocal variables of that function with no restrictions. When a closure is passed as argument to another function, it can also access and modify variables defined outside its scope.

Java 8 lambdas and anonymous classes do something similar to closures: they can be passed as argument to methods and can access variables outside their scope. But they have a restriction: *they can't modify the content of local variables of a method in which the lambda is defined*.

Those variables have to be implicitly final: this restriction exists because local variables live on the stack and are implicitly confined to the thread they're in. Allowing capture of mutable local variables opens new thread-unsafe possibilities, which are undesirable (instance variables are fine because they live on the heap, which is shared across threads).

Method references

Method references let you reuse existing method definitions and pass them just like lambdas, when lambdas actually call *one single method*.

Let's look at the Sorting Apples Example.

Using a method reference and java.util.Comparator.comparing:

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

Your first method reference!

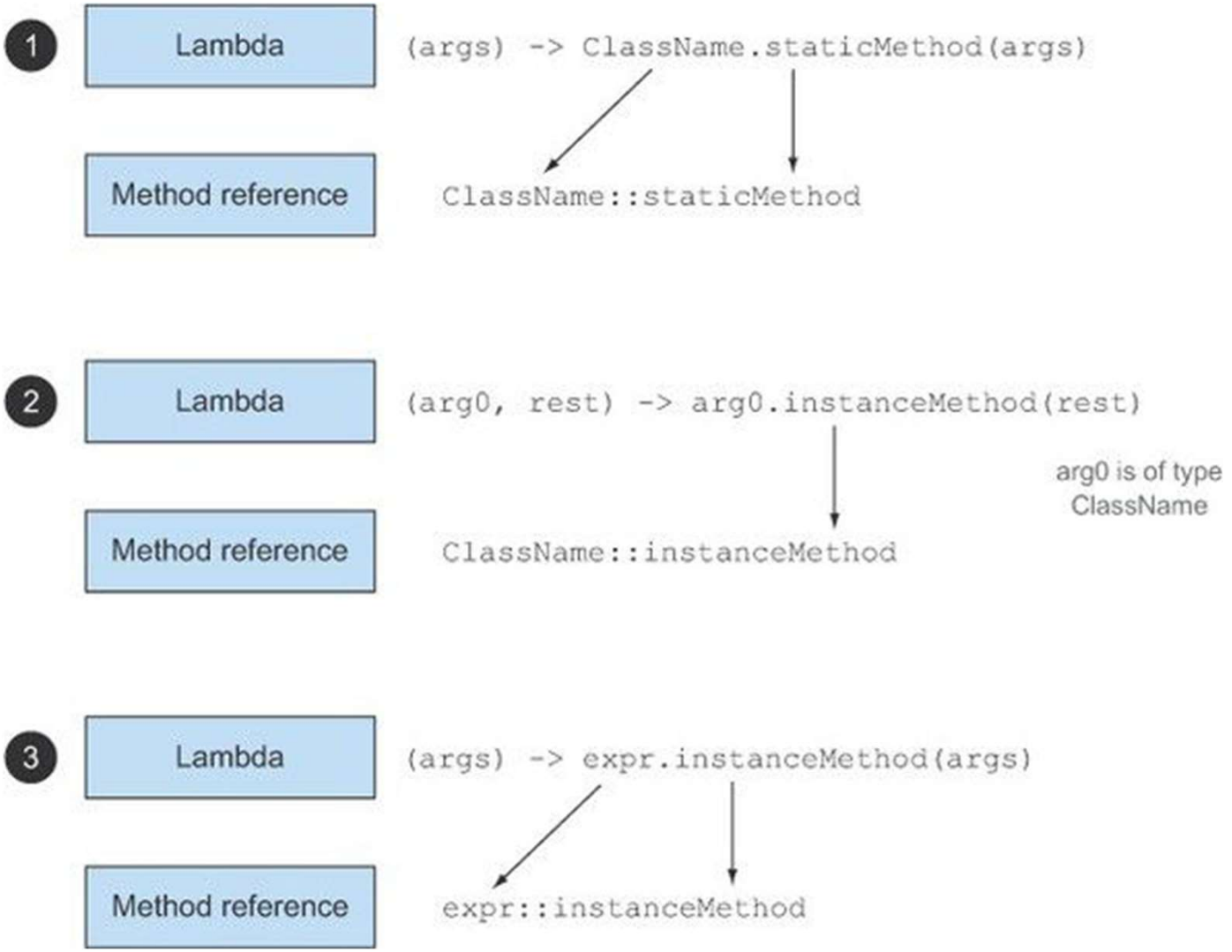
Apple::getWeight is shorthand for the lambda expression (Apple a) -> a.getWeight().

comparing is **<Apple, Integer> Comparator<Apple> java.util.Comparator.comparing(Function<? super Apple, ? extends Integer> keyExtractor)**

Kinds of Method References

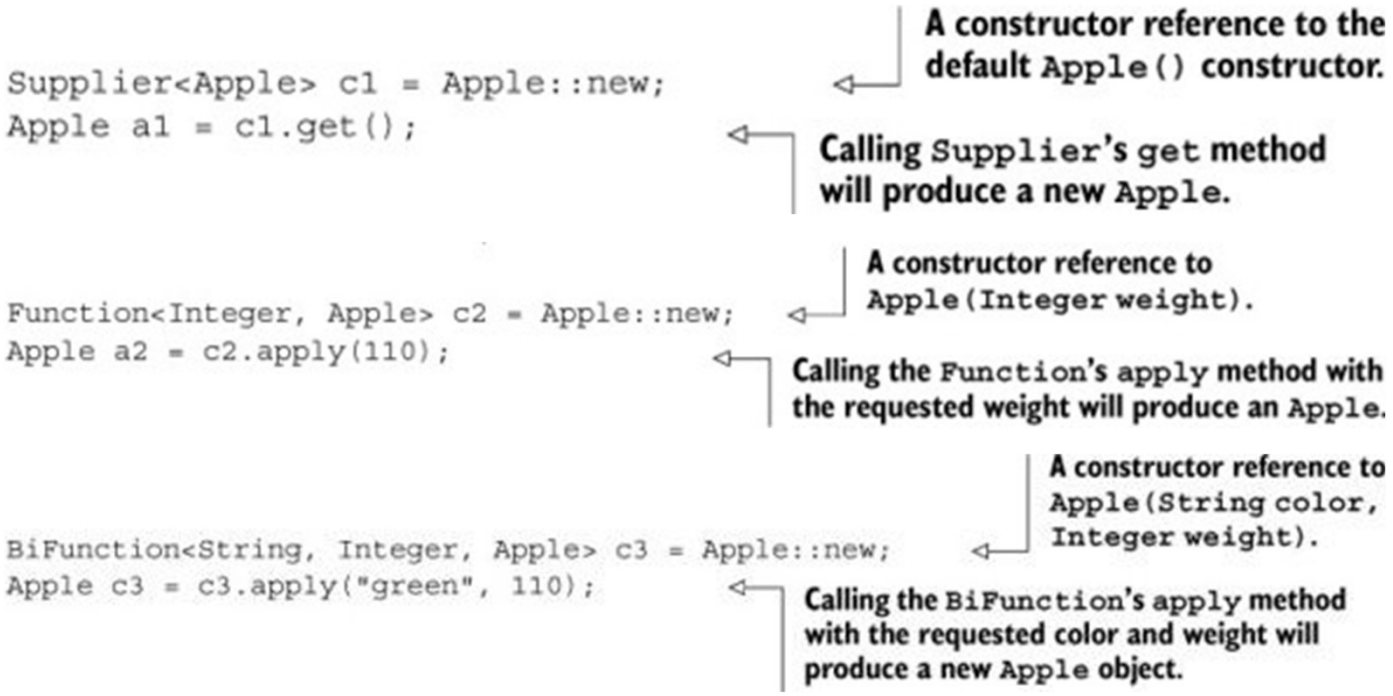
1. A method reference to a **static method** (for example, the method parseInt of Integer, written Integer::parseInt)
2. A method reference to an **instance method of an arbitrary type** (for example, the method length of a String, written String::length)
3. A method reference to an **instance method of an existing object** (for example, suppose you have a local variable anApple that holds an object of type Apple, which supports an instance method getWeight(); you can write anApple::getWeight)

Writing Method References for Lambdas



Constructor References

You can create a reference to an existing constructor using its name and the keyword new



This allows to create “dyamic object factories”

Streams

Streams are an update to the Java API that lets you manipulate collections of data in a declarative way. The focus is not on a collection as a “container”, but a collection as a “processor”. Streams can be processed in parallel transparently, without you having to write any multithreaded code

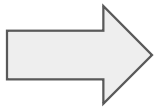
StreamBasic Example:

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

Filter the elements using an accumulator.

Sort the dishes with an anonymous class.

Process the sorted list to select the names of dishes.



```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
        menu.stream()
            .filter(d -> d.getCalories() < 400)
            .sorted(comparing(Dish::getCalories))
            .map(Dish::getName)
            .collect(toList());
```

Store all the names in a List.

Select dishes that are below 400 calories.

Sort them by calories.

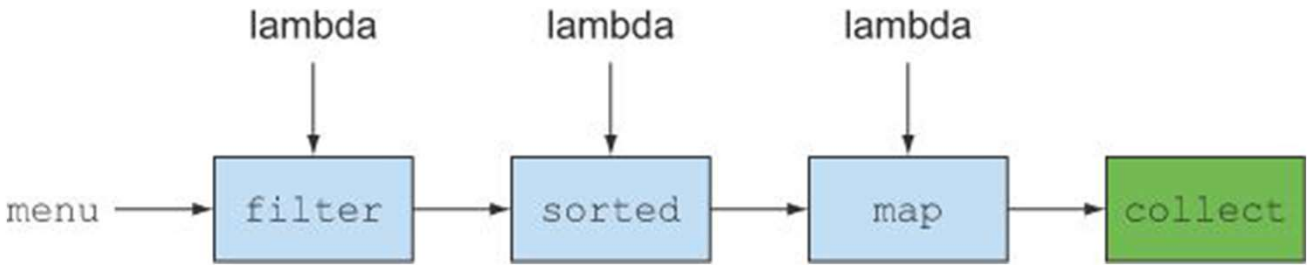
Extract the names of these dishes.

Benefits of Streams

The code is written in a declarative way: you specify what you want to achieve as opposed to specifying how to implement an operation

You chain together several building-block operations to express a complicated data processing pipeline

The processing can be inherently parallelized (given certain conditions)

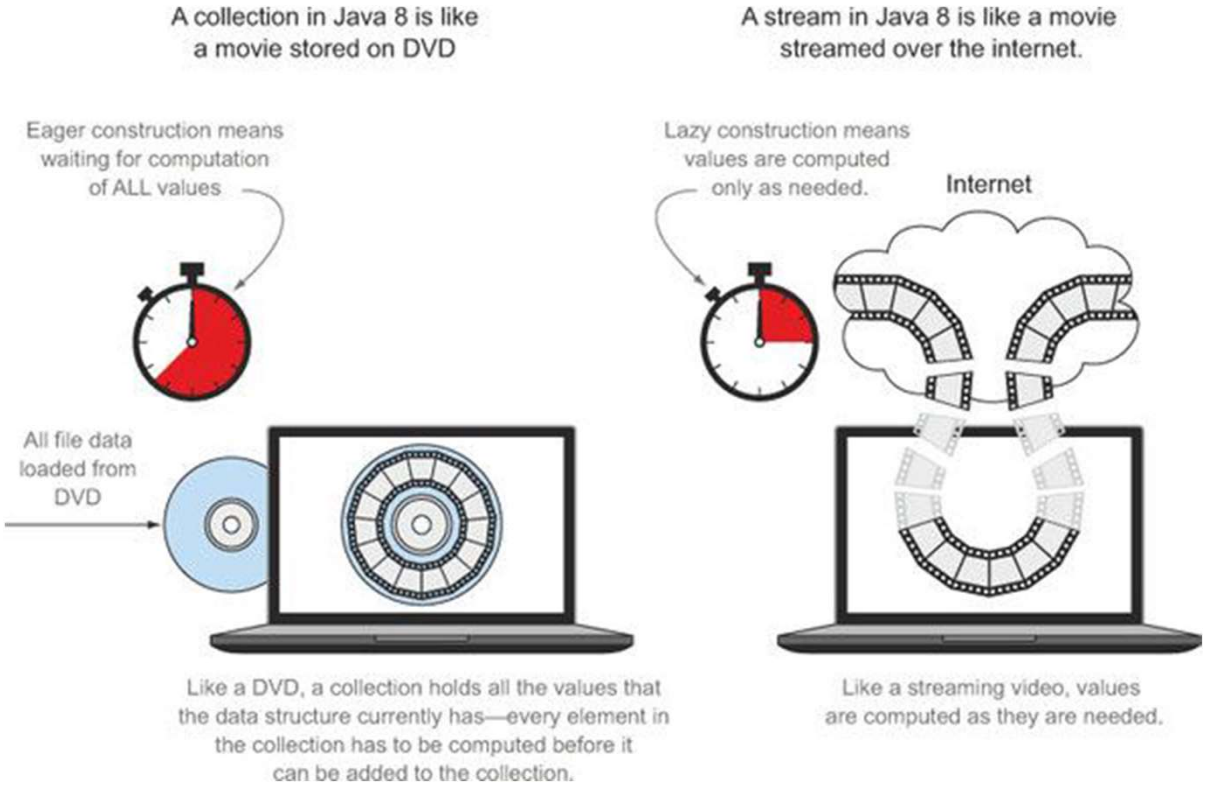


Recipe for building Streams

Streams vs Collections

In streams the focus is on “just in time” processing, not acting as a container

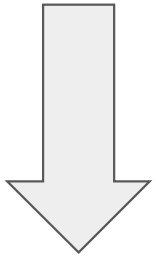
A stream is traversable *only once* (Streamvscollection example)



External vs Internal Iteration

```
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish d = iterator.next();
    names.add(d.getName());
}
```

Iterating explicitly



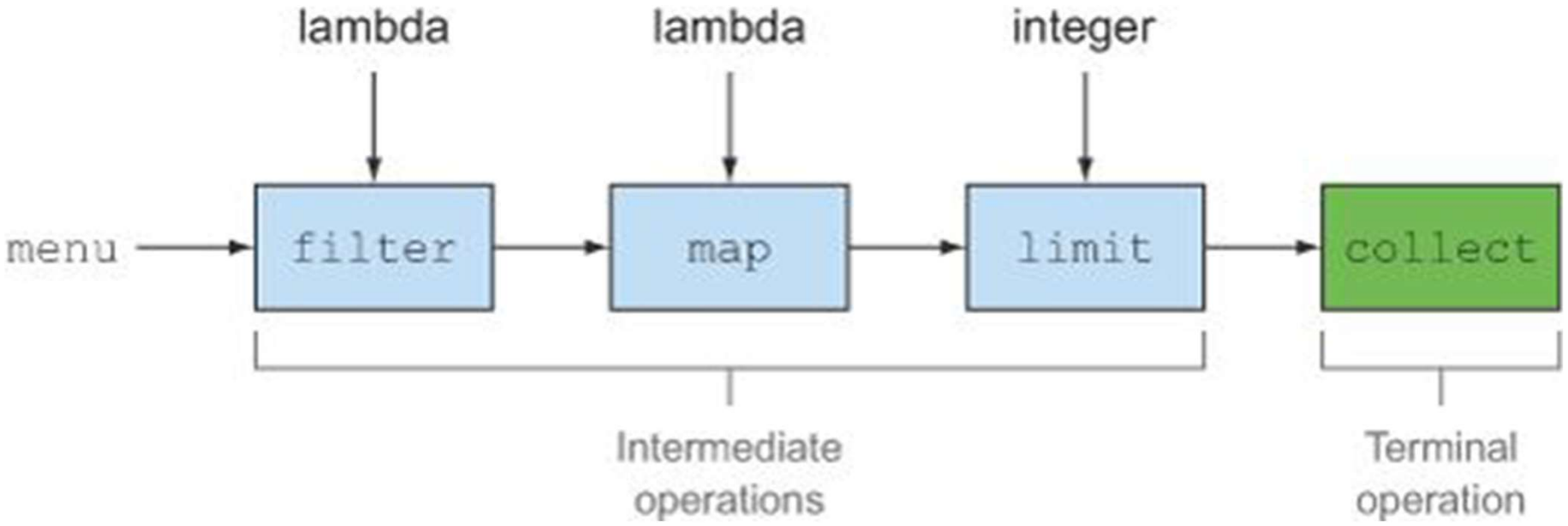
```
List<String> names = menu.stream()
                      .map(Dish::getName)
                      .collect(toList());
```

Start executing the pipeline of operations; no iteration!

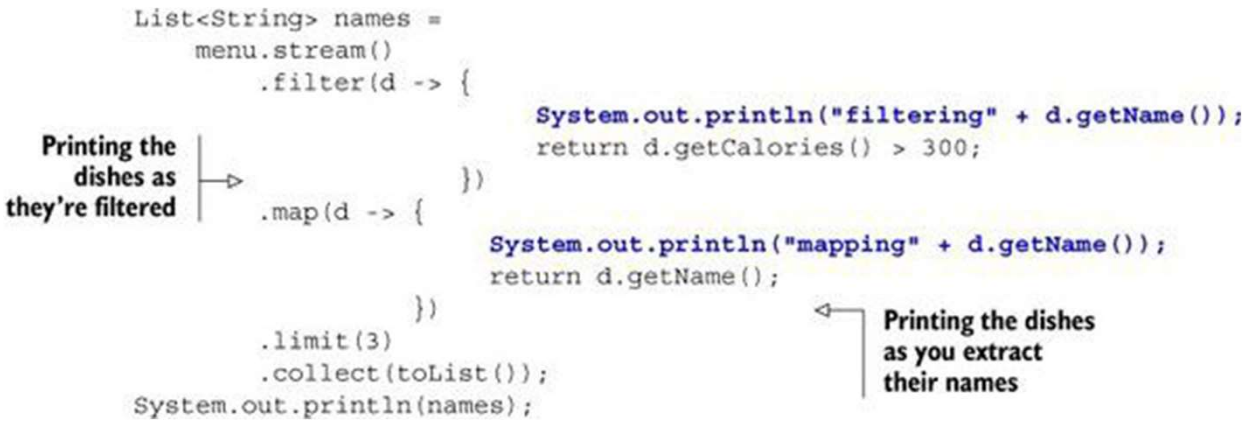
Parameterize `map` with the `getName` method to extract the name of a dish.

Stream Operations

Intermediate operations and terminal operations



Stream Structure



Laziness execution

Short-circuit operation

Loop fusion - what is the output of this code?

Terminal operations produce a result from a stream pipeline

Working with streams in general involves three items:

- A data source (such as a collection) to perform a query on
- A chain of intermediate operations that form a stream pipeline
- A terminal operation that executes the stream pipeline and produces a result

Intermediate Operations

Operation	Type	Return type	Argument of the operation	Function descriptor
`filter`	Intermediate	`Stream<T>`	`Predicate<T>`	`T -> boolean`
`map`	Intermediate	`Stream<R>`	`Function<T, R>`	`T -> R`
`limit`	Intermediate	`Stream<T>`		
`sorted`	Intermediate	`Stream<T>`	`Comparator<T>`	`(T, T) -> int`
`distinct`	Intermediate	`Stream<T>`		

Terminal Operations

Operation	Type	Purpose
`forEach`	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns `void`.
`count`	Terminal	Returns the number of elements in a stream. The operation returns a `long`.
`collect`	Terminal	Reduces the stream to create a collection such as a `List`, a `Map`, or even an `Integer`. See chapter 6 for more detail.

General Operations with Streams

- Filtering and Slicing (example) - filter, distinct, limit, skip
- Mapping and Flattening (example) - map, flatMap
- Finding and Matching (example) - anyMatch, allMatch, nonMatch, findAny, findFirst
- Reducing (example)

Example: given a list of numbers, return a list of the square of each number which is an even number.

Exercise

Given two lists of numbers, how would you return all pairs of numbers? For example, given a list [1, 2, 3] and a list [3, 4] you should return [(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)].

For simplicity, you can represent a pair as an array with two elements.

Return only pairs whose sum is divisible by 3; for example, (2, 4) and (3, 3) are valid

Parallelism, Stateless Operations, Bounded and Unbounded Streams

Using internal iteration, the internal implementation can perform the reduce operation in parallel

Iterative summation via external iteration, in contrast, involves shared updates to a sum variable, which doesn't parallelize gracefully. You have to use synchronization, and thread contention prevents any substantial parallelism

To do real parallelism, a new pattern is necessary: partition the input, sum the partitions, and combine the sums. To achieve this, the lambda passed to reduce *can't change state* (stateless operations) (for example, instance variables), and the *operation needs to be associative* so it can be executed in any order.

But operations like reduce, sum, and max need to have internal state to accumulate the result. In this case the internal state is small. In our example it consisted of an int or double. The internal state is of *bounded* size no matter how many elements are in the stream being processed. In other cases, the state is unbounded: all the unknown size has to be gathered to perform the operation (e.g. sort, distinct)

Overview of Stream Operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
distinct	Intermediate (stateful-unbounded)	Stream<T>		
skip	Intermediate (stateful-bounded)	Stream<T>	long	
limit	Intermediate (stateful-bounded)	Stream<T>	long	
map	Intermediate	Stream<R>	Function<T, R>	T -> R
flatMap	Intermediate	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	Intermediate (stateful-unbounded)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	Terminal	boolean	Predicate<T>	T -> boolean
noneMatch	Terminal	boolean	Predicate<T>	T -> boolean
allMatch	Terminal	boolean	Predicate<T>	T -> boolean
findAny	Terminal	Optional<T>		
findFirst	Terminal	Optional<T>		
forEach	Terminal	void	Consumer<T>	T -> void
collect	terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Terminal	long		

The Trading Exercise

Using the Transaction and Trader objects given in the example:

1. Find all transactions in the year 2011 and sort them by value (small to high).
2. What are all the unique cities where the traders work?
3. Find all traders from Cambridge and sort them by name.
4. Return a string of all traders' names sorted alphabetically.
5. Are any traders based in Milan?
6. Print all transactions' values from the traders living in Cambridge.
7. What's the highest value of all the transactions?
8. Find the transaction with the smallest value.

```
Trader raoul = new Trader("Raoul", "Cambridge");
    Trader mario = new Trader("Mario","Milan");
    Trader alan = new Trader("Alan","Cambridge");
    Trader brian = new Trader("Brian","Cambridge");

                                List<Transaction> transactions =
Arrays.asList(
        new Transaction(brian, 2011, 300),
        new Transaction(raoul, 2012, 1000),
        new Transaction(raoul, 2011, 400),
        new Transaction(mario, 2012, 710),
        new Transaction(mario, 2012, 700),
        new Transaction(alan, 2012, 950)
    );
```

Creating Streams

Streams can be created in various way (see examples):

- From values - Stream.of()
- From arrays
- From files
- Stream.iterate
- Stream.generate
- Using a Supplier for a Stream

Collectors

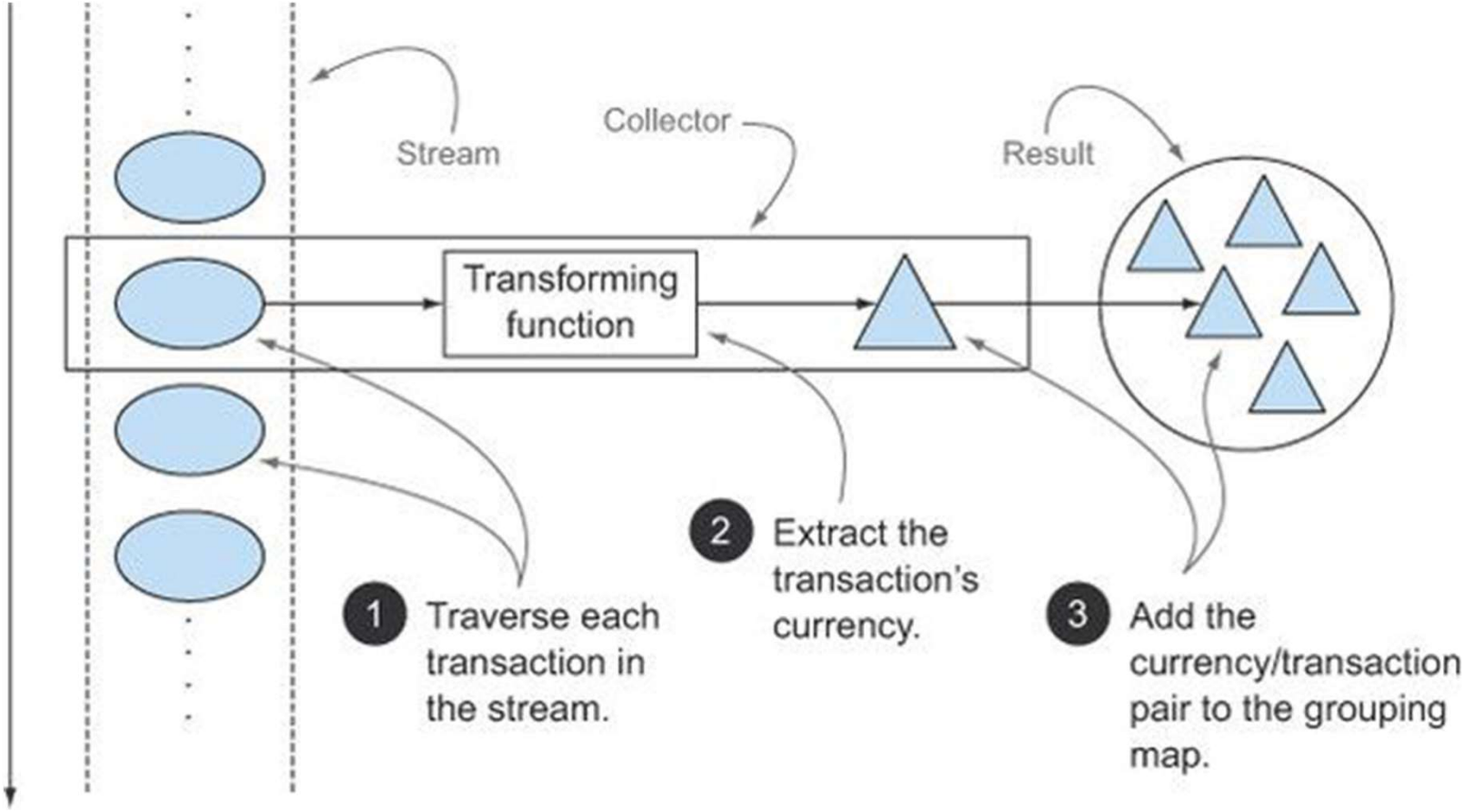
Collectors implement a general collect interface, which allows for better performance and parallelism in the terminal stage (reduce is immutable, collect is mutable). Collectors can, for example:

- Group a list of transactions by currency to obtain the sum of the values of all transactions with that currency (returning a Map<Currency, Integer>)
- Partition a list of transactions into two groups: expensive and not expensive (returning a Map<Boolean, List<Transaction>>)
- Create multilevel groupings such as grouping transactions by cities and then further categorizing by whether they’re expensive or not (returning a Map<String, Map<Boolean, List<Transaction>>>)

Collectors - Grouping

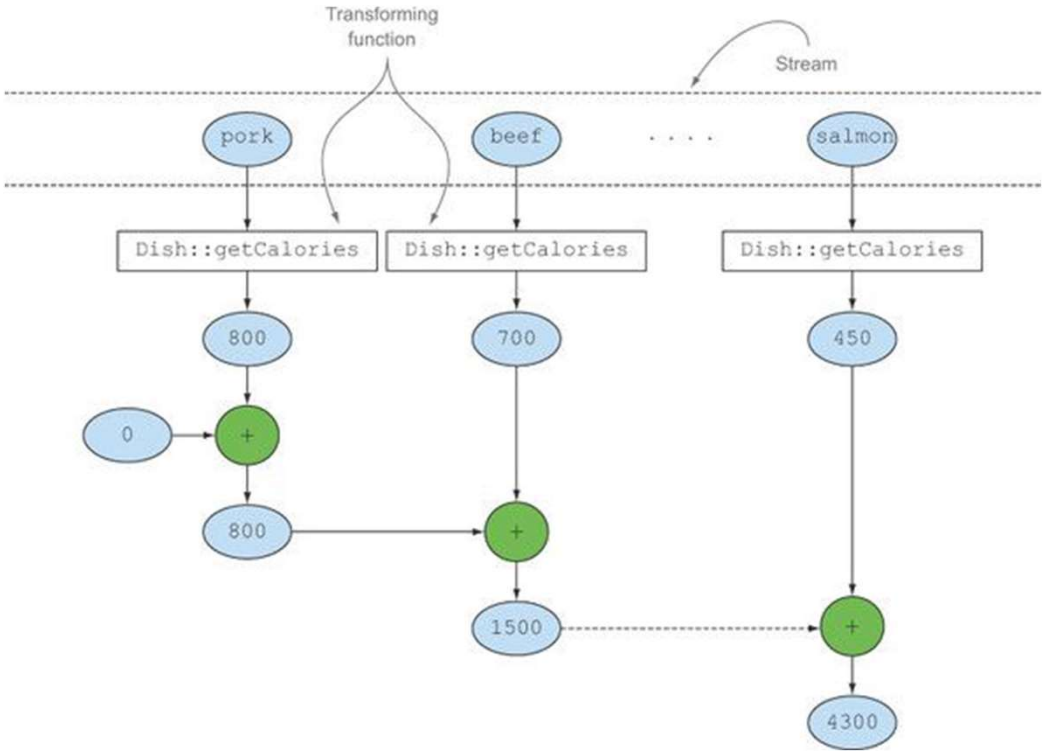
Map<Currency, List<Transaction>> transactionsByCurrencies =
transactions.stream().collect(groupingBy(Transaction::getCurrency));

Collectors as general reduction methods



Summarizing, Reducing, Joining, Grouping, Partitioning

- counting() -- Summarize example
- summingInt()
- averagingInt()
- summarizingInt()
- joining()
- maxBy(), minBy()
- reducing() -- Reducing example
- groupingBy() -- Grouping example
- collectingAndThen()
- partitioningBy() -- Partitioning example



Collector Interface

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

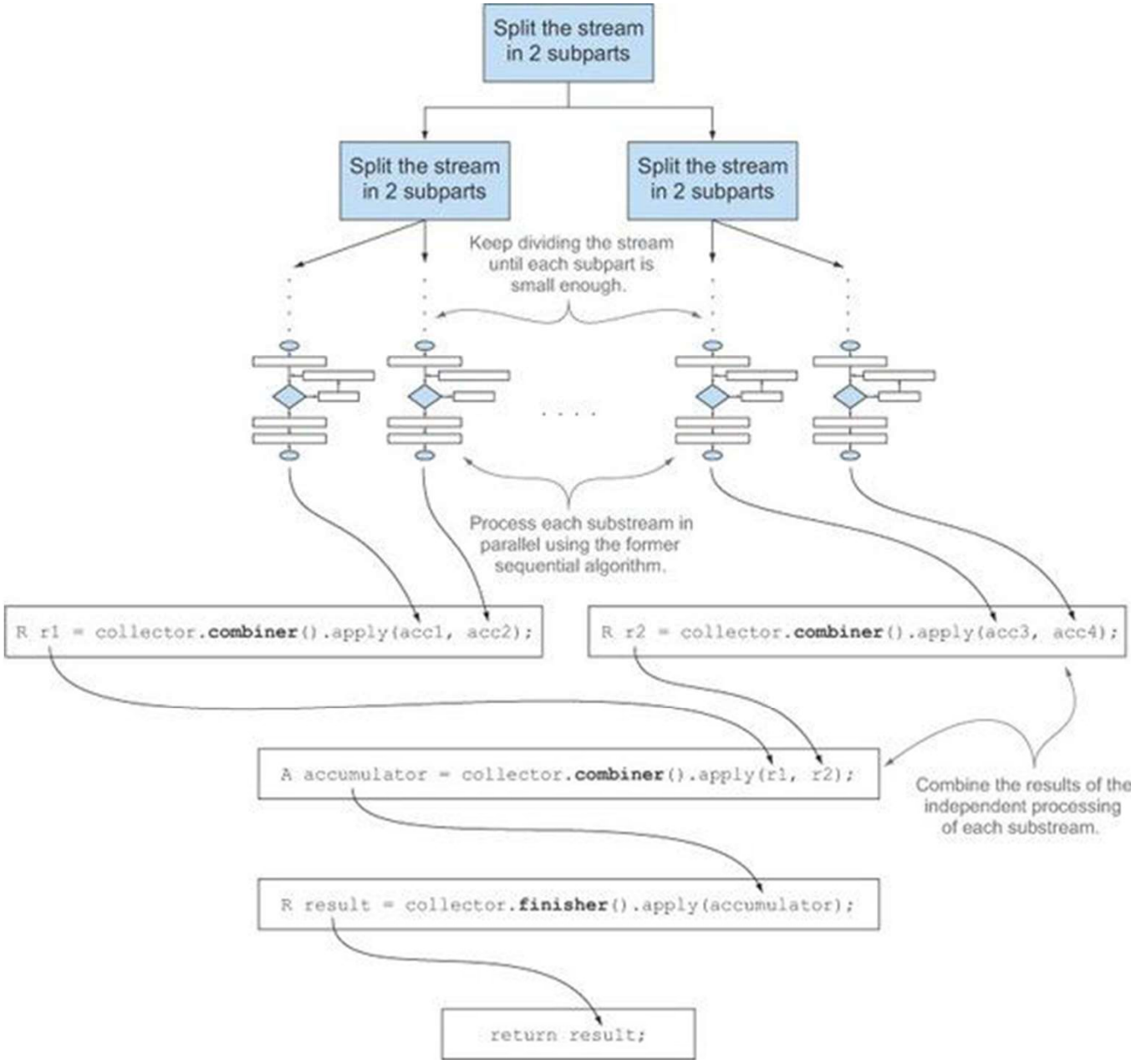
The Collector interface consists of a set of methods that provide a blueprint for how to implement specific reduction operations (Example: ToListCollector)

- T is the generic type of the items in the stream to be collected.
- A is the type of the accumulator, the object on which the partial result will be accumulated during the collection process.
- R is the type of the object (typically, but not always, the collection) resulting from the collect operation.

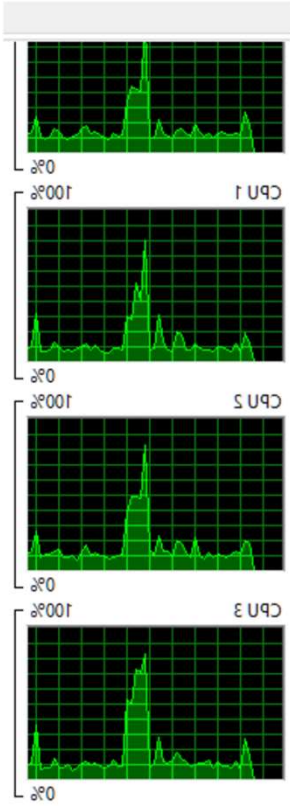
Components of the Collector Interface

- The supplier method has to return a Supplier of an empty result—a parameterless function that when invoked creates an instance of an empty accumulator used during the collection process.
- The accumulator method returns the function that performs the reduction operation. The function returns void because the accumulator is modified in place.
- The finisher method has to return a function that's invoked at the end of the accumulation, in order to transform the accumulator object into the final result.
- The combiner method defines how the accumulators resulting from the reduction of different subparts of the stream are combined when the subparts are processed in parallel (allows a parallel reduction of the stream).
- Characteristics, returns an immutable set providing hints about whether the stream can be reduced in parallel and which optimizations are valid when doing so. (UNORDERED CONCURRENT)

Collector Parallelization



Prime Numbers Example



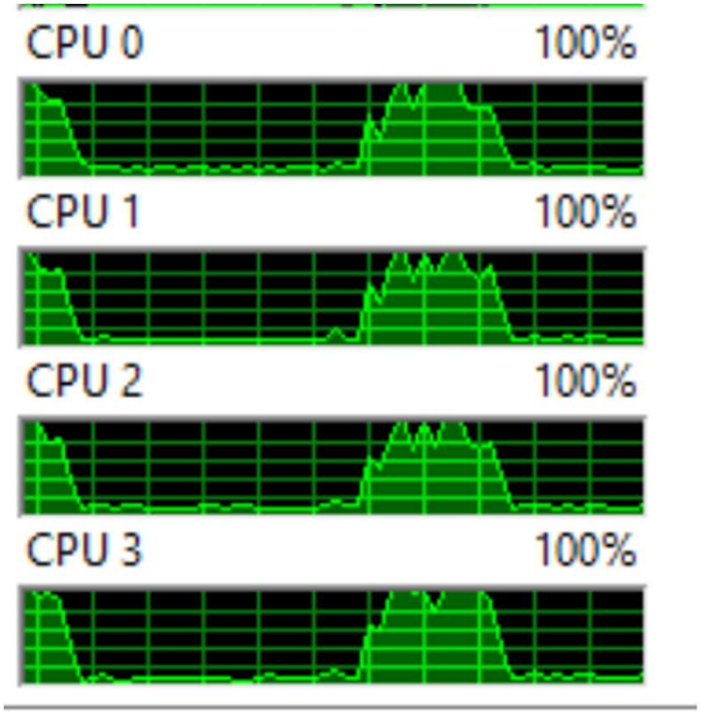
```
public Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector
                (int n) {
    IntStream.rangeClosed(2, n).boxed()
        .collect(
                    () -> new HashMap<Boolean, List<Integer>>() {{   <- Supplier
                       put(true, new ArrayList<Integer>());
                       put(false, new ArrayList<Integer>());
                    }},
                (acc, candidate) -> {                                <- Accumulator
                   acc.get( isPrime(acc.get(true), candidate) )
                      .add(candidate);
                },
                  (map1, map2) -> {                                  <- Combiner
                    map1.get(true).addAll(map2.get(true));
                    map1.get(false).addAll(map2.get(false));
                });
}
```

Parallel Streams

It’s possible to turn a collection into a parallel stream by invoking the method parallel() on the collection source. A parallel stream is a stream thatsplits its elements into multiple chunks, processing each chunk with a different thread. Thus, you can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy.

Not always parallelization means improvement. In the parallel example, the iterative version using a traditional for loop runs much faster than parallel() because it works at a much lower level and, more important, doesn’t need to perform any boxing or unboxing of the primitive values.

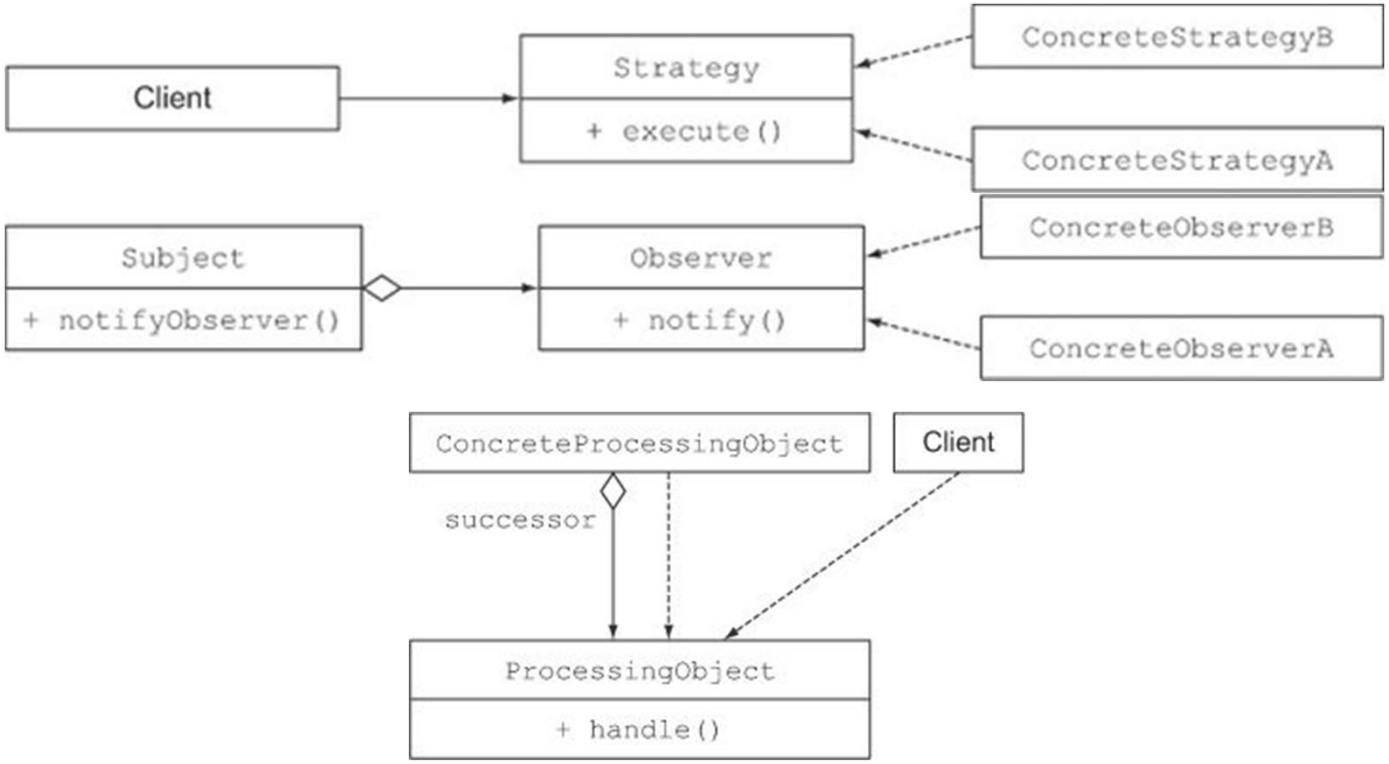
Using a native stream without unboxing yields comparable performance.



Rethinking OOP Patterns with FP

Many existing object-oriented design patterns can be made redundant or written in a more concise way using lambda expressions. For example these five design patterns:

- Strategy
- Template method
- Observer
- Chain of responsibility
- Factory



Rethinking OOP Patterns with FP

```
// with lambdas
Validator v3 = new Validator((String s) -> s.matches("\\d+"));
System.out.println(v3.validate("aaaa"));
Validator v4 = new Validator((String s) -> s.matches("[a-z]+"));
System.out.println(v4.validate("bbbb"));
```

```
feedLambda.registerObserver((String tweet) -> {
    if(tweet != null && tweet.contains("money")){
        System.out.println("Breaking news in NY! " + tweet); }
});
```

```
Function<String, String> pipeline = headerProcessing.andThen(spellCheckerProcessing);
```

```
final static private Map<String, Supplier<Product>> map = new HashMap<>();
static {
    map.put("loan", Loan::new);
    map.put("stock", Stock::new);
    map.put("bond", Bond::new);
}
```

Java Core OCP Part 3

https://github.com/thimotyb/java8.git

Interfaces, Exceptions, DateTime, Localization

Default Methods

Java interfaces group related methods together into a contract. Any class that implements an interface must provide an implementation for each method defined by the interface or inherit the implementation from a superclass. But this causes a problem when library designers need to update an interface to add a new method.

Interfaces in Java 8 can now declare methods with implementation code; this can happen in two ways:

- First, Java 8 allows static methods inside interfaces.
- Second, Java 8 introduces a new feature called default methods that allows you to provide a default implementation for methods in an interface.

Resolution rules

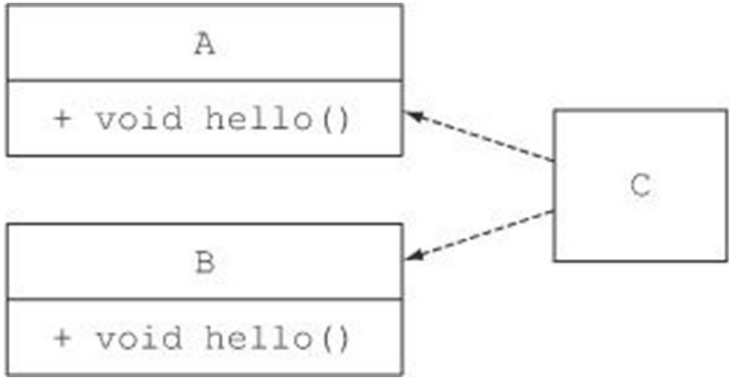
```
public interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}
public interface B extends A {
    default void hello() {
        System.out.println("Hello from B");
    }
}

public class C implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

What gets printed?

There are three rules to follow when a class inherits a method with the same signature from multiple places (such as another class or interface):

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

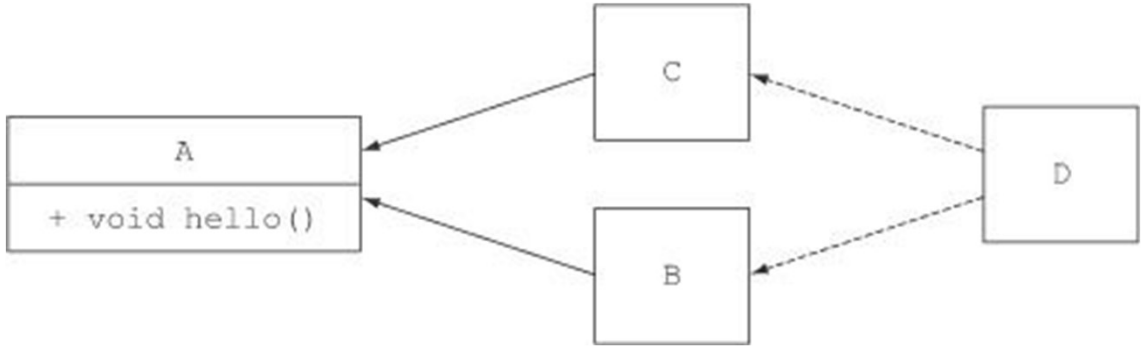
Interface Disambiguation




```
public class C implements B, A {
    void hello(){
        B.super.hello();
    }
}
```

Explicitly choosing to call the method from interface B

Diamonds aren’t a programmer’s best friends




```
public interface A{
    default void hello(){
        System.out.println("Hello from A");
    }
}

public interface B extends A { }

public interface C extends A { }

public class D implements B, C {
    public static void main(String... args) {
        new D().hello();
    }
}
```

What gets printed?

- What if B has a hello() as well?
- What if both B and C have a different hello()?

Interfaces vs Abstract Classes

So what’s the difference between an abstract class and an interface? They both can contain abstract methods and methods with a body.

First, a class can extend only from one abstract class, but a class can implement multiple interfaces.

Second, an abstract class can enforce a common state through instance variables (fields). An interface can’t have instance variables.

Optionals

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

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

Java 8 introduces a new class called java.util.Optional<T> that encapsulates an optional value.

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

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

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

A person might or might not own a car, so you declare this field `Optional`.

A car might or might not be insured, so you declare this field `Optional`.

An insurance company must have a name.

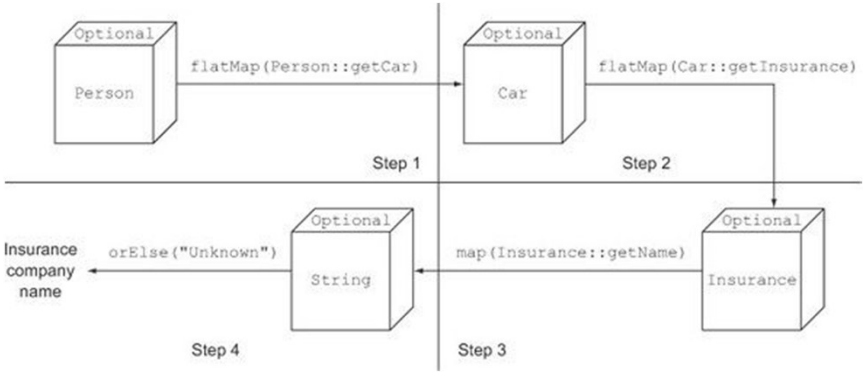
Common patterns with Optional

Optional<Car> optCar = Optional.of(car);

Optional<Car> optCar = Optional.ofNullable(car);

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

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

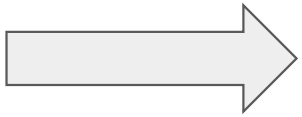



```
public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

Exceptions - Try-with-resources

- Resources can now implement the Autocloseable interface
- You can add more than one resource in the try() declaration
- Catch and finally are executed after closing the resources

```
open a resource
try {
work with the resource
}
finally {
close the resource
}
```



```
try (Resource res = ...) {
work with res
}
```

Exceptions - Implementing Autocloseable

- It is possible to create new resources using Autocloseable

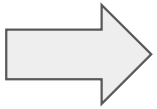
```
public class MyResource implements AutoCloseable {
        public void close()                {
                // take care of closing the resource
        }
}
```

Exercise: Create two autocloseable resources, using them in a try-with-resources, and apply also a catch and finally. Determine what is the order of the calls.

Exceptions - Multiple Catch

- Allow to overcome redundancy in catching exceptions
- You can’t use multiple names for the exceptions
- Order does not matter
- You have to ensure that only one exception is thrown at a time

```
try {
// access the database and write to a file }
catch (SQLException e) { handleErrorCase(e);
} catch (IOException e) { handleErrorCase(e) ;
}
```



```
try {
// access the database and write to a file
}  catch (SQLException | IOException e) {
        handleErrorCase(e);
}
```

Exceptions - Suppressed Exceptions

- In Java 6, an exception thrown in a finally clause discards the previous exception.
- The try-with-resources statement reverses this behavior. When an exception is thrown in a close method of one of the AutoCloseable objects, the original exception gets rethrown, and the exceptions from calling close are caught and attached as “suppressed” exceptions.

```
} catch (Exception e) {
                    System.err.println(e.getMessage());
                    for (Throwable t : e.getSuppressed())          {
                                System.err.println("suppressed:" + t);
                    }
}
```

- **Exercise**: create a main exception and at least a suppressed exception.

Date/Time API - Instant and Duration

- Java Time counts 86,400 seconds per day (no leap second)
- In Java, an Instant represents a point on the time line.
  - epoch is arbitrarily set at midnight of January 1, 1970 at the prime meridian that passes through the Greenwich Royal Observatory in London.
- Instant.now(); Duration.between();
- You can get the length of a Duration in conventional units by calling toNanos, toMillis, toSeconds, toMinutes, toHours, or toDays.

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

Date/Time API - Arithmetics on Instant and Duration

- Instant and Duration are *immutable:* operations return new instances

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster =
timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();
// Or timeElapsed.toNanos() * 10 < timeElapsed2.toNanos()
```

Date/Time API - Local Dates

- There are two kinds of human time in the new Java API, local date/time and zoned time. Local date/time has a date and/or time of day, but no associated time zone information.
- Do not use zoned time unless you really want to represent absolute time instances. Use Local date and time for schedule times.

```
LocalDate today = LocalDate.now(); // Today's date
LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);
alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);
// Uses the Month enumeration
```

Date/Time API - Date Adjusters

- For scheduling applications, you often need to compute dates such as “the first Tuesday of every month.”
  - DateAdjusters are static methods called with the .with() method

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

Date/Time API - Local Time

- A LocalTime represents a time of day

```
LocalTime rightNow = LocalTime.now();

LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(22, 30, 0)
```

Date/Time API - Zoned Time

- When you have a conference call at 10:00 in New York, but happen to be in Berlin, you expect to be alerted at the correct local time. In this case Time Zones have to be considered.
- Each time zone has an ID, such as America/New_York or Europe/Berlin. To find out all available time zones, call *ZoneId.getAvailableIds*.
- ZonedDateTime builds a specific Instant with Time Zone:

```
ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,
ZoneId.of("America/New_York"));
// 1969-07-16T09:32-04:00[America/New_York]
```

- What happens at the begin/end hour of DST? And when you perform operations that cross a DST Zone?

Date/Time API - Formatting and Parsing

- The DateTimeFormatter class provides three kinds of formatters to print a date/time value:
  - Predefined standard formatters
  - Locale-specific formatters (SHORT, MEDIUM, LONG, FULL)
  - Formatters with custom patterns

```
String formatted = DateTimeFormatter.ISO_DATE_TIME.format(apollo11launch);
// 1969-07-16T09:32:00-05:00[America/New_York]
```

```
DateTimeFormatter formatter =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
String formatted = formatter.format(apollo11launch);
// July 16, 1969 9:32:00 AM EDT
```

Date/Time API - Conversions with Legacy API

- A set of conversion static methods is available
  - java.util.Date: Date.from(instant); date.toInstant()
  - java.util.GregorianCalendar:
    - GregorianCalendar.from(zonedDateTime)
    - cal.toZonedDateTime()
  - LocalDate:
    - Date.valueOf(localDate)
    - date.toLocalDate()
  - etc.

Localization

- A locale specifies the language and location of a user, which allows formatters to take user preferences into account.
- A locale consists of up to five components:
    - A language, specified by two or three lowercase letter
    - Optionally, a script, specified by four letters with an initial uppercase, such as Latn (Latin), Cyrl (Cyrillic)
    - Optionally, a country or region, specified by two uppercase letters or three digits
    - Optionally, a *variant*.
    - Optionally, an extension. Extensions describe local preferences for calendars

Locale Formats

- The NumberFormat class in the java.text package provides three factory methods for formatters that can format and parse numbers
- Use the Currency class to control the currency used by the formatters. You can get a Currency object by passing a currency identifier to the static Currency.getInstance method.
- When formatting date and time, there are four locale-dependent issues:

    1. The names of months and weekdays should be presented in the local language.
    2. There will be local preferences for the order of year, month, and day.
    3. The Gregorian calendar might not be the local preference for expressing dates.
    4. The time zone of the location must be taken into account.

Message internazionalization / Resource Bundles

- When you internationalize a program, you often have messages with variable parts. The static format method of the MessageFormat class takes a template string with placeholders.
- The placeholders can change using ResourceBundles: When localizing an application, it is best to separate the program from the message strings, button labels, and other texts that need to be translated. In Java, you can place them into resource bundles

Java Core OCP Part 4

Annotations, NIO, Security

Annotations

- Annotations are tags that you insert into your source code so that some tool can process them. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.
- Annotations do not change the way your programs are compiled. The Java compiler generates the same virtual machine instructions with or without the annotations.
- To benefit from annotations, you need to select a processing tool and use annotations that your processing tool understands, before you can apply that tool to your code.

Using Annotations

- Annotations can have key/value pairs called elements.
- An item can have multiple annotations.
- Annotations are usually applied to method declarations. There are many other places where annotations can occur. They fall into two categories: declarations (classes, methods, parameters…) and type uses (generics, implemented interfaces, constructors, exceptions…)
- You cannot annotate class literals and imports.

Defining Annotations

```
@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface Test {

    long timeout();

    ...

}
```

- Each annotation must be declared by an annotation interface, with the @interface syntax. The methods of the interface correspond to the elements of the annotation.
- The @interface declaration creates an actual Java interface. Tools that process annotations receive objects that implement the annotation interface.
- The element declarations in the annotation interface are actually method declarations. The methods of an annotation interface can have no parameters and no throws clauses, and they cannot be generic.
- The @Target and @Retention annotations are meta-annotations. They annotate the Test annotation, indicating the places where the annotation can occur and where it can be accessed.
- The value of the @Target meta-annotation is an array of ElementType objects, specifying the items to which the annotation can apply. You can specify any number of element types, enclosed in braces. The compiler checks that you use an annotation only where permitted.

Element Type	Annotation Applies To
`ANNOTATION_TYPE`	Annotation type declarations
`PACKAGE`	Packages
`TYPE`	Classes (including `enum`) and interfaces (including annotation types)
`METHOD`	Methods
`CONSTRUCTOR`	Constructors
`FIELD`	Instance variables (including `enum` constants)
`PARAMETER`	Method or constructor parameters
`LOCAL_VARIABLE`	Local variables
`TYPE_PARAMETER`	Type parameters
`TYPE_USE`	Uses of a type

Annotation @Retention

- The @Retention meta-annotation specifies where the annotation can be accessed. There are three choices.
    - RetentionPolicy.SOURCE: The annotation is available to source processors, but it is not included in class files.
    - RetentionPolicy.CLASS: The annotation is included in class files, but the virtual machine does not load them. This is the default.
    - RetentionPolicy.RUNTIME: The annotation is available at runtime and can be accessed through the reflection API.
- The Java API defines a number of annotation interfaces in the java.lang, java.lang.annotation, and javax.annotation packages. Four of them are meta-annotations that describe the behavior of annotation interfaces. The others are regular annotations that you use to annotate items in your source code

Annotations for Compilation

- The @Deprecated annotation can be attached to any items whose use is no longer encouraged.
- The @Override makes the compiler check that the annotated method really overrides a method from the superclass.
- The@SuppressWarnings annotation tells the compiler to suppress warnings of a particular type
- The @SafeVarargs annotation asserts that a method does not corrupt its varargs parameter
- The @Generated annotation is intended for use by code generator tools. Any generated source code can be annotated to differentiate it from programmer-provided code.
- @FunctionalInterface is used to annotate conversion targets for lambda expressions.

Annotations for Managing Resources

- The @PostConstruct and @PreDestroy annotations are used in environments that control the lifecycle of objects, such as web containers and application servers.
- The @Resource annotation is intended for resource injection. For example, consider a web application that accesses a database.

Meta-Annotations

- @Target
- @Retention
- The @Documented meta-annotation gives a hint to documentation tools such as Javadoc.
- The @Inherited meta-annotation applies only to annotations for classes. When a class has an inherited annotation, then all of its subclasses automatically have the same annotation. This makes it easy to create annotations that work similar to marker interfaces (such as the Serializable interface).
- The @Repeatable meta-annotation makes it possible to apply the same annotation multiple times. For historical reasons, the implementor of a repeatable annotation needs to provide a container annotation that holds the repeated annotations in an array.

Example: Processing Annotations at Runtime

- Let us consider a simple example of processing an annotation at runtime using the reflection API.
- @ToString Example, create this annotation to use on classes and fields. Possibility to choose if include the name of the field in the toString.
- The processing is based on the methods of the AnnotatedElement interface
- The processor of the annotation is the static method in the ToStrings class.

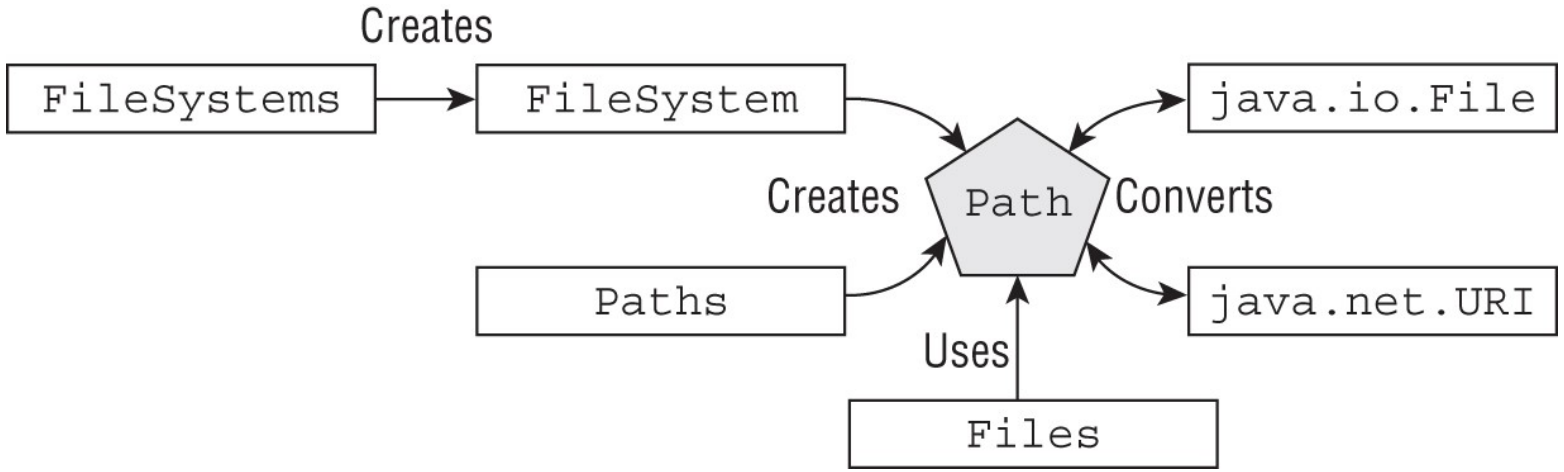
```
T getAnnotation(Class<T>)

T getDeclaredAnnotation(Class<T>)

T[] getAnnotationsByType(Class<T>)

T[] getDeclaredAnnotationsByType(Class<T>)

Annotation[] getAnnotations()

Annotation[] getDeclaredAnnotations()
```

Source-Level Annotation Processing

- Another use for annotation is the automatic processing of source files to produce more source code, configuration files, scripts, or whatever else one might want to generate.
- Annotation processing is integrated into the Java compiler. During compilation, you can invoke annotation processors by running
    - javac -processor ProcessorClassName1,ProcessorClassName2,... sourceFiles
- An annotation processor implements the Processor interface, generally by extending the AbstractProcessor class. You need to specify which annotations your processor supports.

NIO.2 – File Handling

- The cornerstone of NIO.2 is the java.nio.file.Path interface. A Path instance represents a hierarchical path on the storage system to a file or directory. You can think of a Path as the NIO.2 replacement for the java.io.File class.
- The Path interface contains support for symbolic links
- Constructors are needed to get instances of the Path interface



NIO.2 – Networking support and Non-blocking IO

- NIO.2 provides additional Channel interfaces to create network-based classes
- `DatagramChannel`
    - The DatagramChannel class of Java's NIO module provides a selectable channel for the datagram-oriented sockets. In other words, it allows creating a datagram channel to send and receive the datagrams (UDP packets). (Channels are one-way abstractions to Buffers or Sockets)
    - https://www.baeldung.com/java-nio-datagramchannel
- `AsynchronousSocketChannel`
    - `https://blogs.oracle.com/javamagazine/post/java-nio-nio2-buffers-channels-async-future-callback`

Asynchronous operations with NIO.2

NIO.2 added asynchronous capabilities for both socket-based and file-based I/O, allowing you to take full advantage of your hardware's capabilities. This is an important feature for any language that wishes to remain relevant in the server-side and systems programming spaces.

Asynchronous I/O is simply a type of input/output processing that allows other activity to take place before the reading and writing has finished

The NIO.2 API provides many new asynchronous channels that you can work with.

- AsynchronousFileChannel for file-based I/O
- AsynchronousSocketChannel for socket-based I/O; this supports timeouts
- AsynchronousServerSocketChannel for asynchronous sockets accepting connections
- AsynchronousDatagramChannel for "fire and forget" I/O; it does not check for a valid connection

Async with Future or Callback

There are two main styles that you can adopt when using the NIO.2 asynchronous I/O APIs: the Future style and the Callback style.

Future style. The Future style uses a Future object from java.util.concurrent. The Future object represents the result of your asynchronous operation and is either still pending or will be fully materialized once the operation has been completed. Typically you would use the get() method (with or without a timeout) to retrieve the results when the asynchronous I/O activity has been completed. The following example reads 100 bytes from a file and then gets the result (which will be the number of bytes actually read).

Callback style, which uses the CompletionHandler interface. Some developers are more comfortable using the Callback style because it is similar to event-handling code. The java.nio.channels.CompletionHandler<V, A> interface (where V is the result type and A is the attached object you are getting the result from) has two methods that must be given implementations. This means, of course, that you can’t use a lambda expression to represent one. Instead, anonymous inner classes are typically used. These methods, completed(V, A) and failed(V, A), must be given an implementation that describes how the program should behave when the asynchronous I/O operation has been completed successfully or has failed for some reason. One (and only one) of these two methods will be invoked when the asynchronous I/O activity has been completed.

Designing a Secure Object with sensitive Information

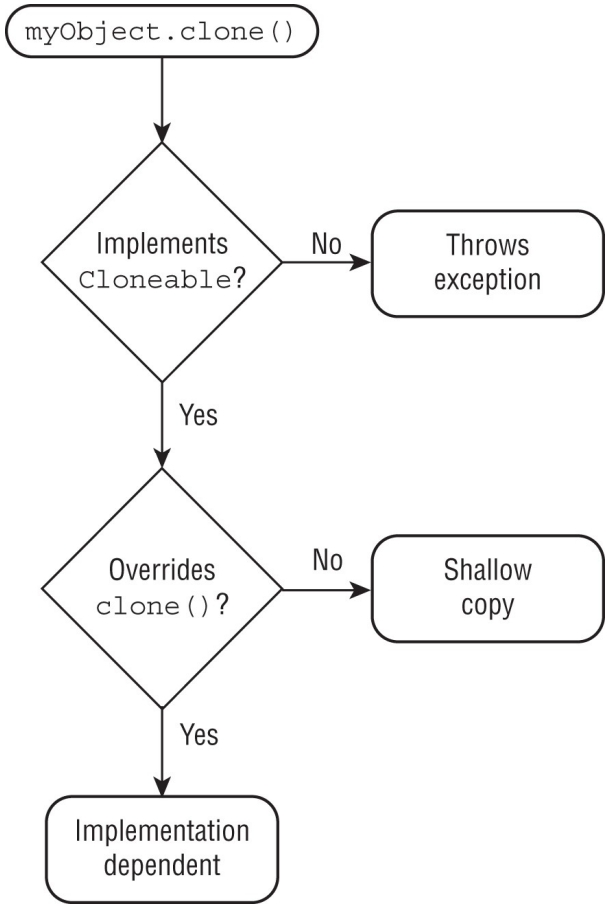
- The main issues are: access control, extensibility, validation, and creating immutable objects
  - limit accessibility by making instance variables private or package-private, whenever possible
  - If your application is using modules, you can do even better by only exporting the security packages to the specific modules that should have access
  - Restrict extensibility by marking a class *final (see ComboLocks example)*
  - Create immutable objects, so that the object state cannot be changed after it is created (see Immutable example)

Immutable classes

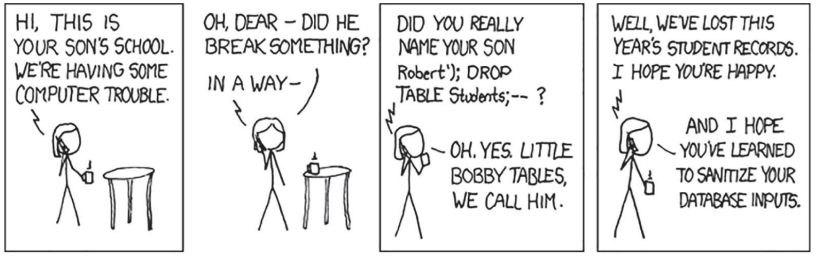
- Immutable objects are helpful when writing secure code because you don't have to worry about the values changing. They also simplify code when dealing with concurrency.
- A common strategy for making a class immutable.
    - Mark the class as final.
    - Mark all the instance variables private.
    - Don't define any setter methods and make fields final.
    - Don't allow referenced mutable objects to be modified. Do not return internal object without defensive copying.
    - Use a constructor to set all properties of the object, making a defensive copy if needed. (clone so that other code cannot tamper the original reference afterwards)

Cloning Objects

- Java has a Cloneable interface that you can implement if you want classes to be able to call the clone() method on your object. This helps with making defensive copies.
- By default, the clone() method makes a **shallow** copy of the data, which means only the top-level object references and primitives are copied. No new objects from within the cloned object are created. For example, if the object contains a reference to an ArrayList, a shallow copy contains a reference to that same ArrayList. Changes to the ArrayList in one object will be visible in the other since it is the same object.
- Write an implementation that does a *deep copy and clones the objects inside*. A deep copy does make a new ArrayList object. Changes to the cloned object do not affect the original



Injection and Input Validation



- Validate Inputs, e.g. with WhiteListing
- Use PreparedStatements to avoid SQL Injection

```
public int getOpening(Connection conn, String day)
      throws SQLException {
   String sql = "SELECT opens FROM hours WHERE day = '" + day +"'";
   try (var ps = conn.prepareStatement(sql);
      var rs = ps.executeQuery()) {
      if (rs.next())
         return rs.getInt("opens");
   }
   return -1;
}
```

Avoid command injection if «..» is inputed as dir:

```
Console console = System.console();
String dirName = console.readLine();
if (dirName.equals("mammal") || dirName.equals("birds")) {
   Path path = Paths.get("c:/data/diets/" + dirName);
   try (Stream<Path> stream = Files.walk(path)) {
      stream.filter(p -> p.toString().endsWith(".txt"))
         .forEach(System.out::println);
   }
}
```

Protecting In-Memory Information

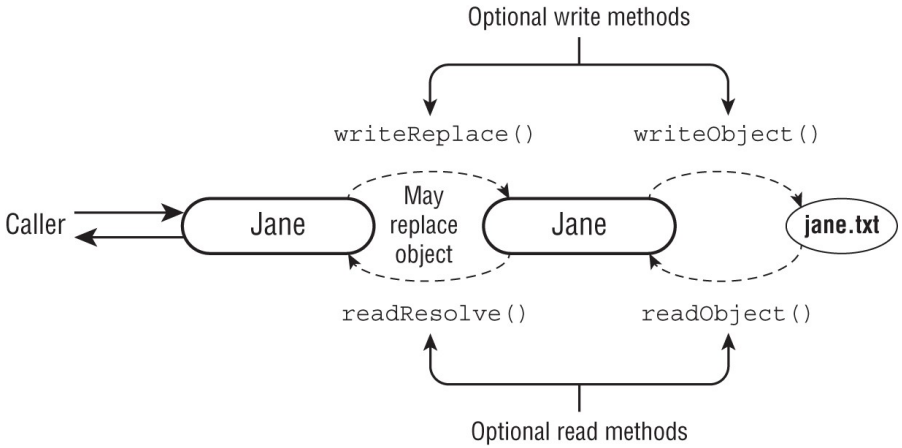
- Do not include sensitive fields in the toString() method to avoid having them ending up in logs, console output, ecc.
- For in-memory passwords or other sensitive information, use char[] instead of a String. This is safer for two reasons:
    - It is not stored as a String, so Java won't place it in the String pool, where it could exist in memory long after the code that used it is run. (Or dumped in case of a provoked crash)
    - You can null out the value of the array element rather than waiting for the garbage collector to do it.
    - When the sensitive data cannot be overwritten, it is good practice to set confidential data to null when you're done using it. If the data can be garbage collected, you don't have to worry about it being exposed later.

Limiting File Access

- Using Policy Files
  - https://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html
  - https://docs.oracle.com/javase/tutorial/security/userperm/policy.html

Serializing and Deserializing Objects

- Marking a sensitive field as *transient* prevents it from being serialized. As an alternative use serialPersistentFields with an array of fields to serialize.
- Security may demand custom serialization: e.g. encrypt or hash sensitive fields before serialization
- It is possible to use pre-post serialization method to protect references to instances and make sure the reference returned is the one we want (readResolve/writeReplace)



Preventing Denial of Service Attacks

- A denial of service (DoS) attack is when a hacker makes one or more requests with the intent of disrupting legitimate requests. Most denial of service attacks require multiple requests to bring down their targets.
- Some common sources of denial of service are:
    - Leaking resources (not closing files, connections, sockets…) → Use try-with-resources
    - Reading very large resources → Check the size of a resource before opening it
    - Including potentially large resources (e.g. “Billion laughs attack”: https://en.wikipedia.org/wiki/Billion_laughs_attack
    - Overflowing → Check for boundaries, int and double precision
    - Wasting Data Structures → Do not let external inputs to grow Maps or Lists, limit the size of Maps or Lists