SHIJIE ZHANG

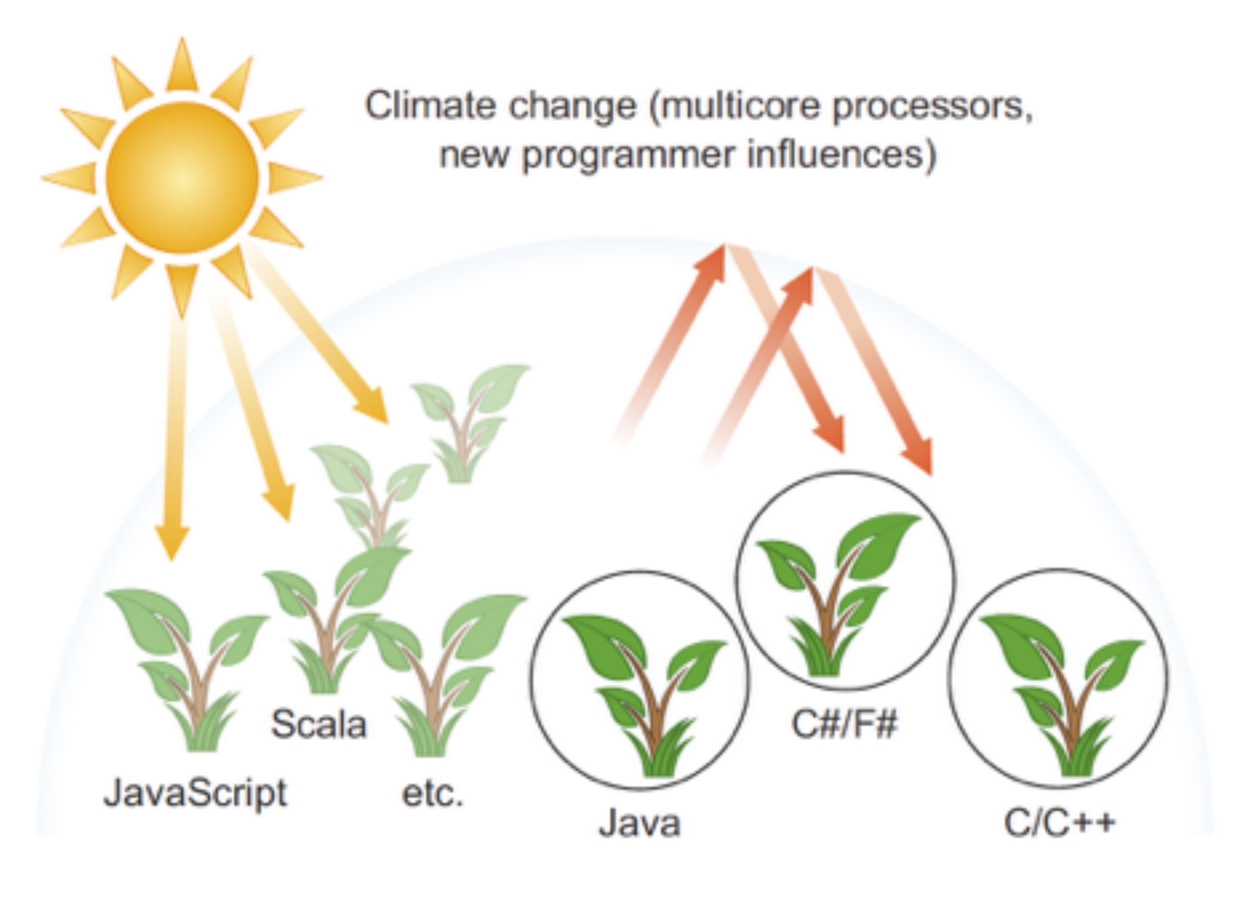
INTRO TO JAVA 8

OUTLINE

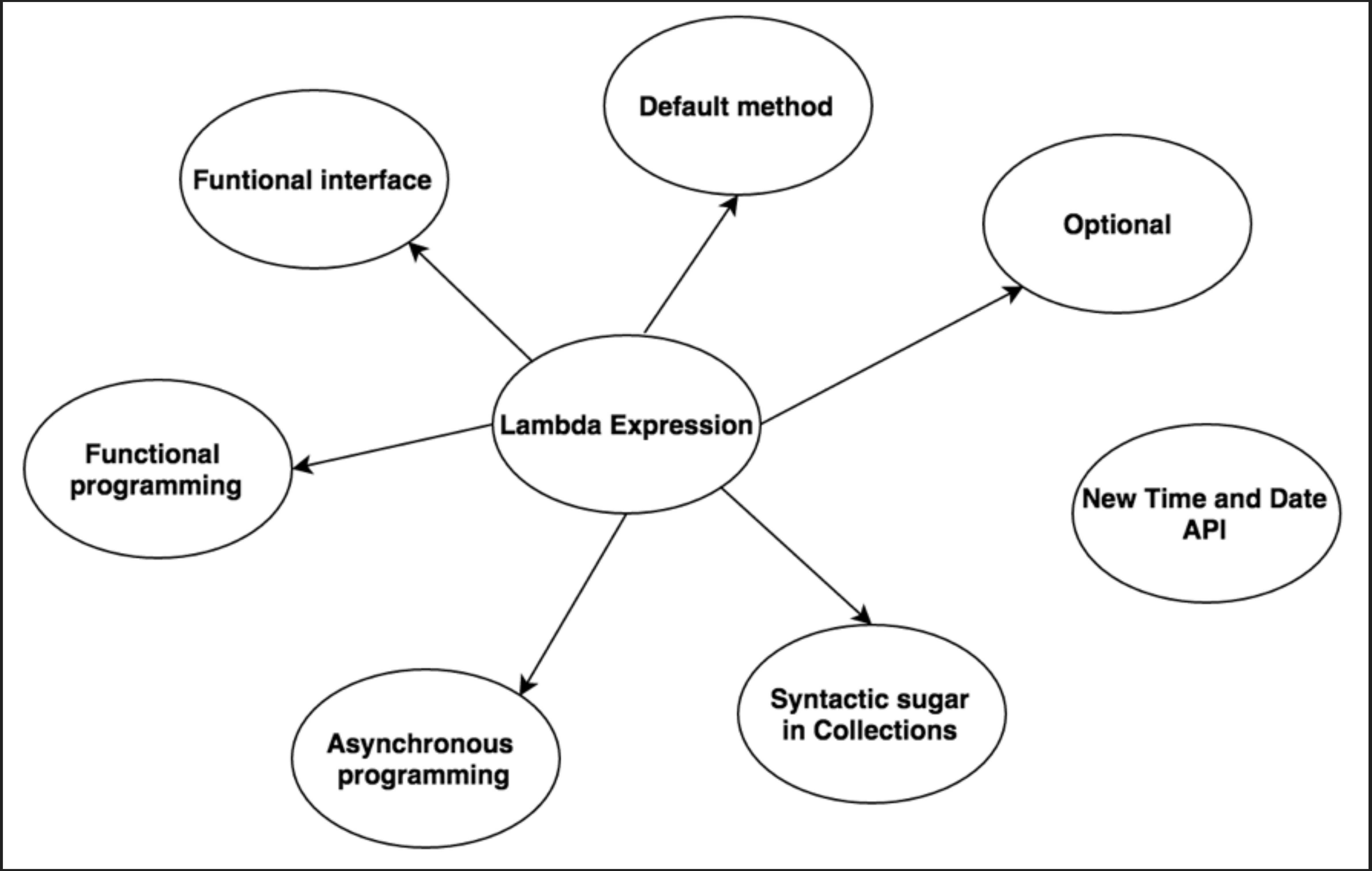
- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

CLIMATE IS CHANGING

- Java was in dominant positions due to its simplicity, portability, safety and free to use.
- JVM-based dynamic language comes up, known for their simplicity and portability. (Groovy, Clojure, Scala)
- Big data is on the rise. Programmers need to deal with large collections.
- Multicore processor is becoming more and more popular. Programmers need to an easier way to do parallel programming.
- Java is kind of verbose.


Climate change (multicore processors, new programmer influences)
Scala
JavaScript
etc.
Java
C#/F#
C/C++

WHAT WILL JAVA 8 BRING US



OUTLINE

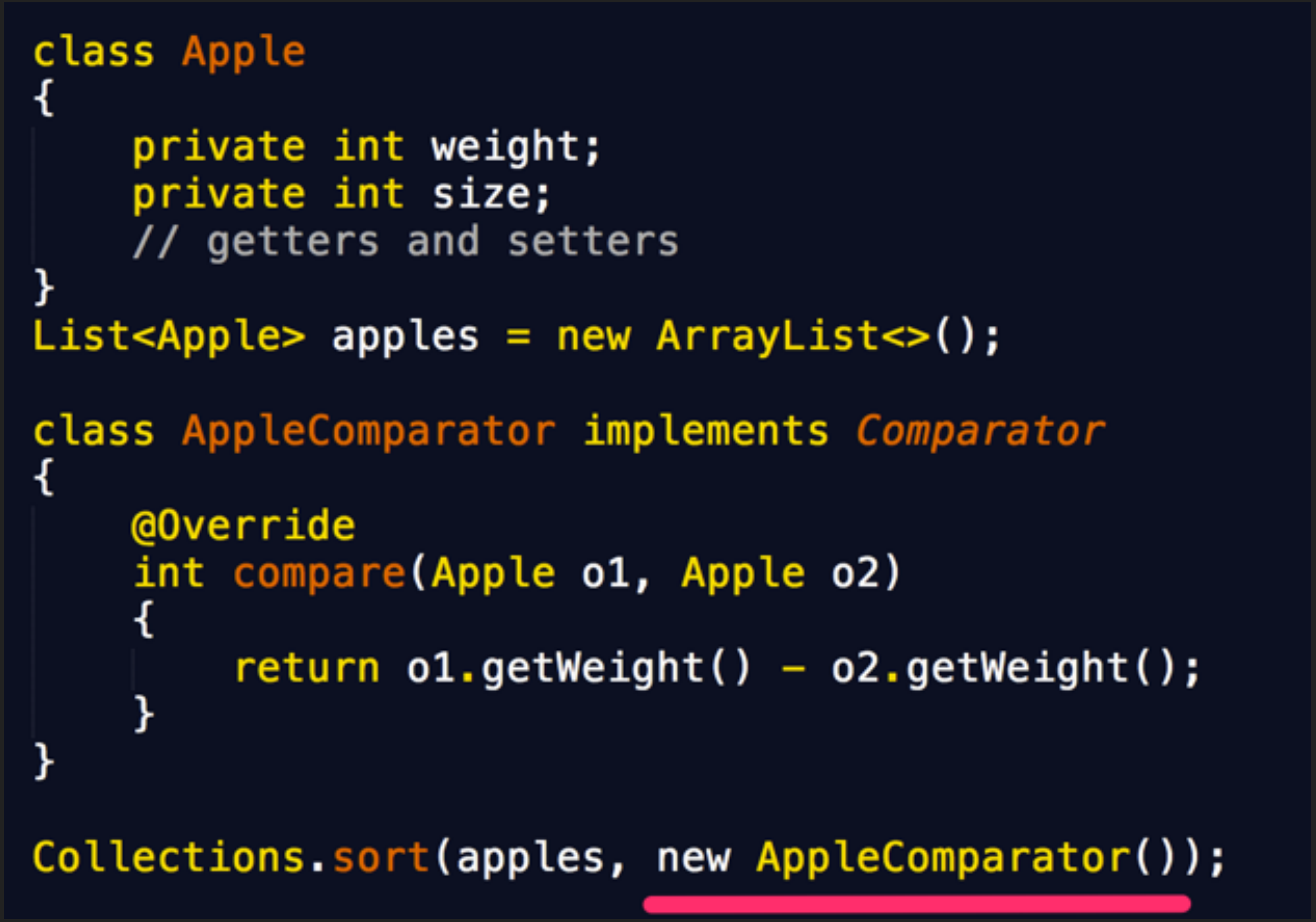
- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

OUTLINE

- What will Java 8 give us
- Behavior parameterization
  - More comprehensive functional interface
  - Lambda expression
  - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
  - Intuition
  - Intermediate and terminal operations
  - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

More comprehensive functional interface

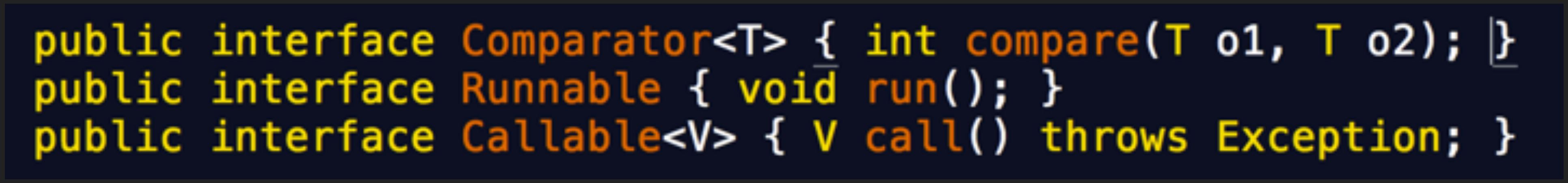
- Comparator is an interface. More exactly, a functional interface.



```
class Apple
{
    private int weight;
    private int size;
    // getters and setters
}
List<Apple> apples = new ArrayList<>();

class AppleComparator implements Comparator
{
    @Override
    int compare(Apple o1, Apple o2)
    {
        return o1.getWeight() - o2.getWeight();
    }
}

Collections.sort(apples, new AppleComparator());
```

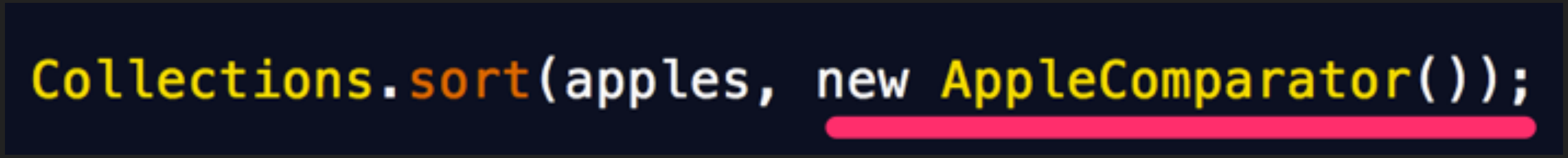
More comprehensive functional interface

- Functional interface: an interface has exactly one abstract method
- Several functional interface exists before Java 8



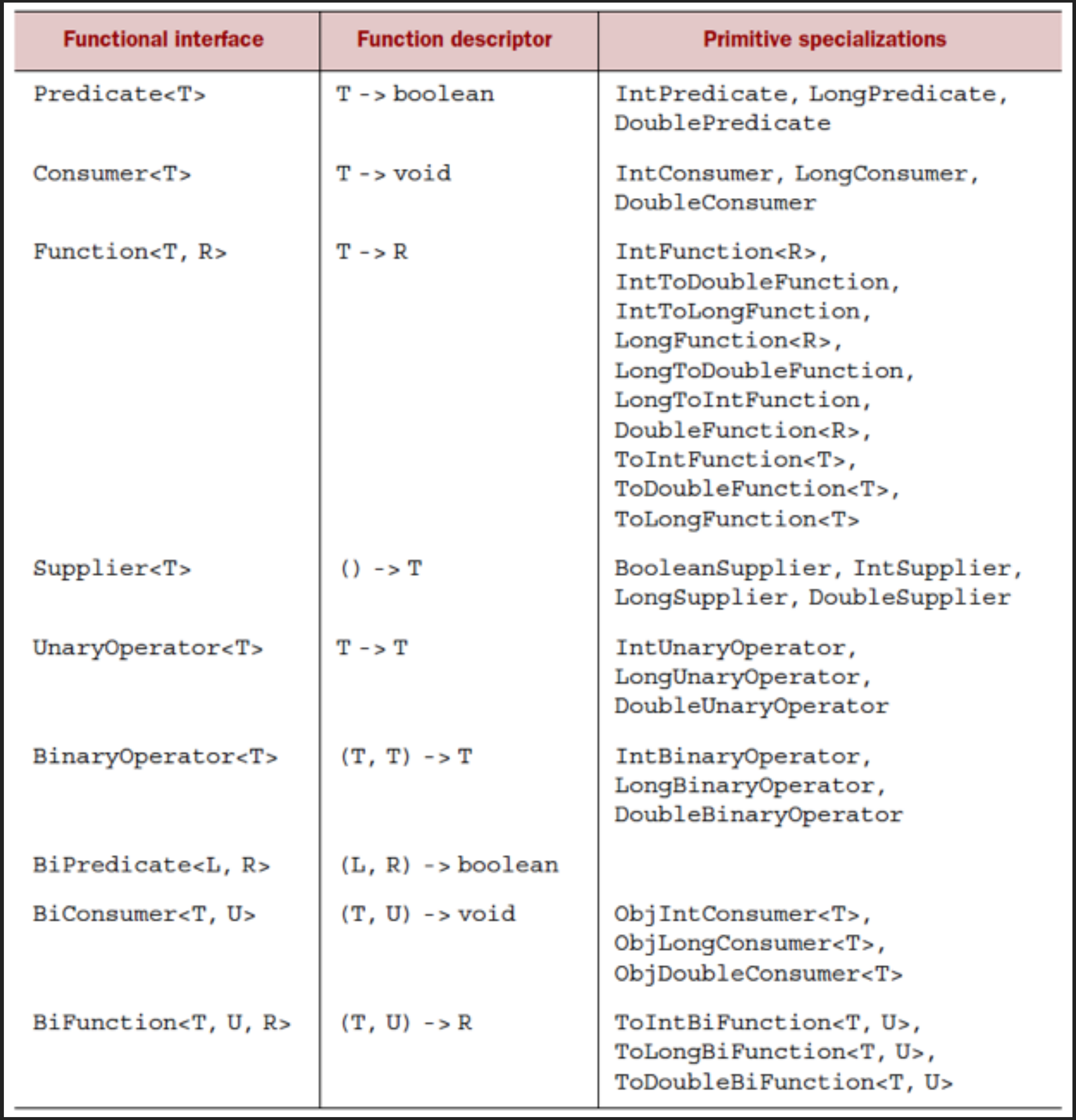
```
public interface Comparator<T> { int compare(T o1, T o2); }
public interface Runnable { void run(); }
public interface Callable<V> { V call() throws Exception; }
```

- Functional interface enables behavior parameterization



```
Collections.sort(apples, new AppleComparator());
```

BEHAVIOR PARAMETERIZATION - LAMBDAS



Functional interface	Function descriptor	Primitive specializations
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>, 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<L, R>	(L, R) -> 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>

Where to use functional interface

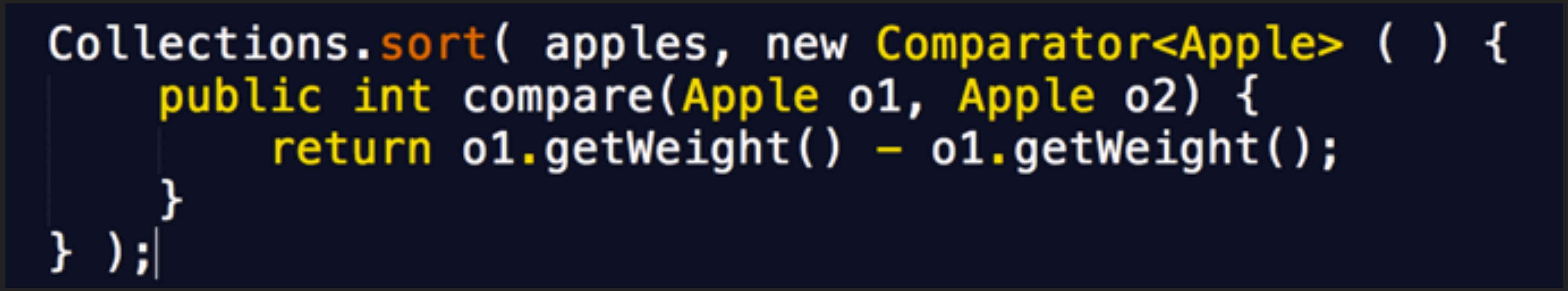
- Anywhere an object could be used
  - method arguments/parameters/return types
  - inside collections
  - Variables
  - …….

BEHAVIOR PARAMETERIZATION - LAMBDAS

In the past, use anonymous class as instance for functional interface

Now, use lambda expression/method reference as instance for functional interface

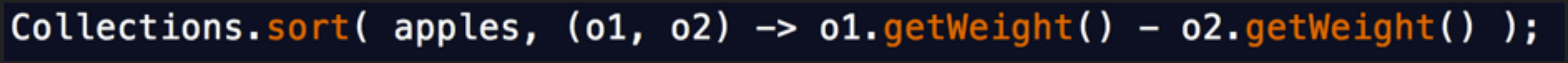
- Think about Comparator – create an anonymous class



```
Collections.sort( apples, new Comparator<Apple> ( ) {
    public int compare(Apple o1, Apple o2) {
        return o1.getWeight() – o1.getWeight();
    }
} );
```

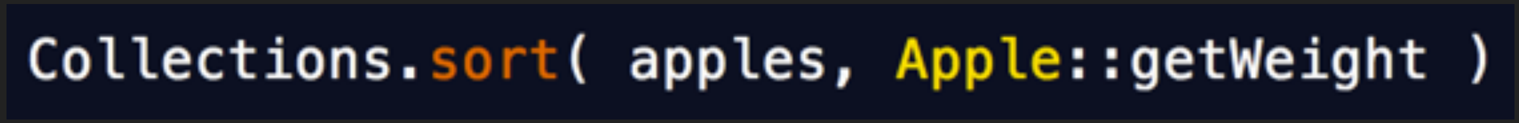
- All anonymous class could be replaced with lambda/method reference
  - Anonymous class ~ lambda expression ~ method reference

- lambda expressions



```
Collections.sort( apples, (o1, o2) -> o1.getWeight() – o2.getWeight() );
```

- method reference



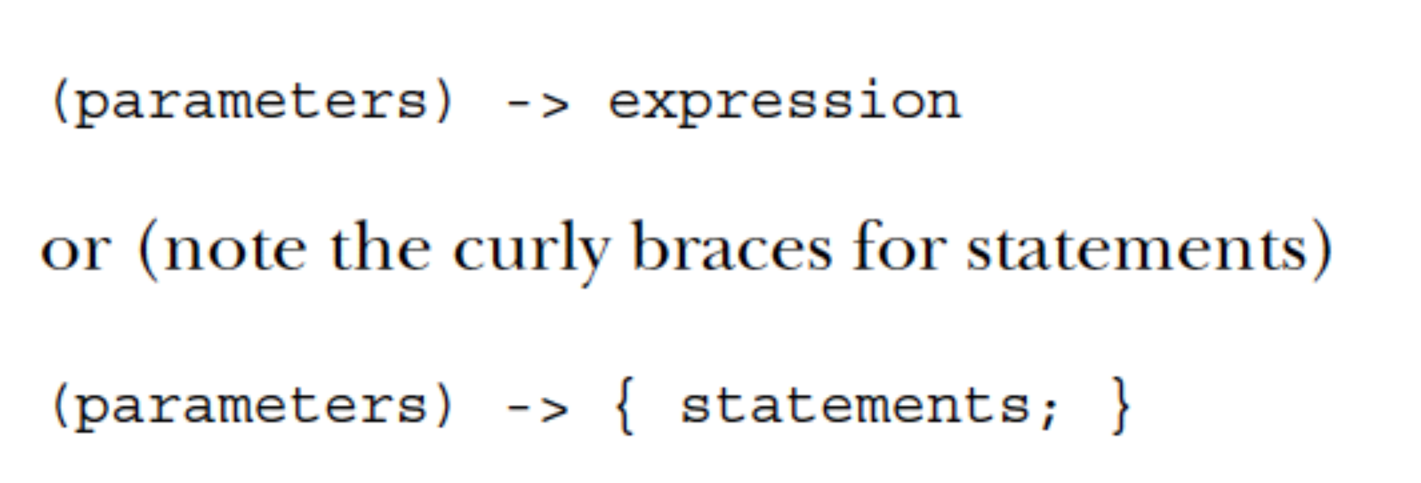
```
Collections.sort( apples, Apple::getWeight )
```

OUTLINE

- What will Java 8 give us
- Behavior parameterization
  - More comprehensive functional interface
  - Lambda expression
  - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
  - Intuition
  - Intermediate and terminal operations
  - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Lambdas definition

- Definition



```
(parameters) -> expression
```

or (note the curly braces for statements)

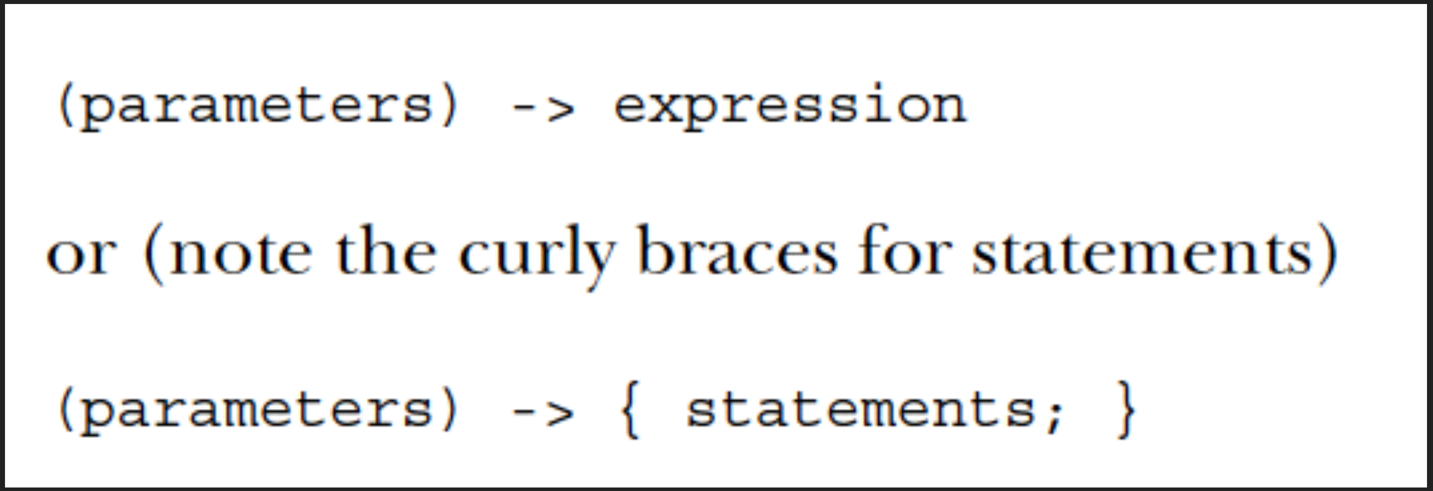
```
(parameters) -> { statements; }
```

- Examples:

1. ( ) -> { }
2. ( ) -> “Raoul”
3. ( ) -> { return “Mario”;}
4. (Integer i) -> return “Alan” + i;
5. (String s) -> { “Iron Man”; }

Lambdas definition

- Definition



```
(parameters) -> expression
```

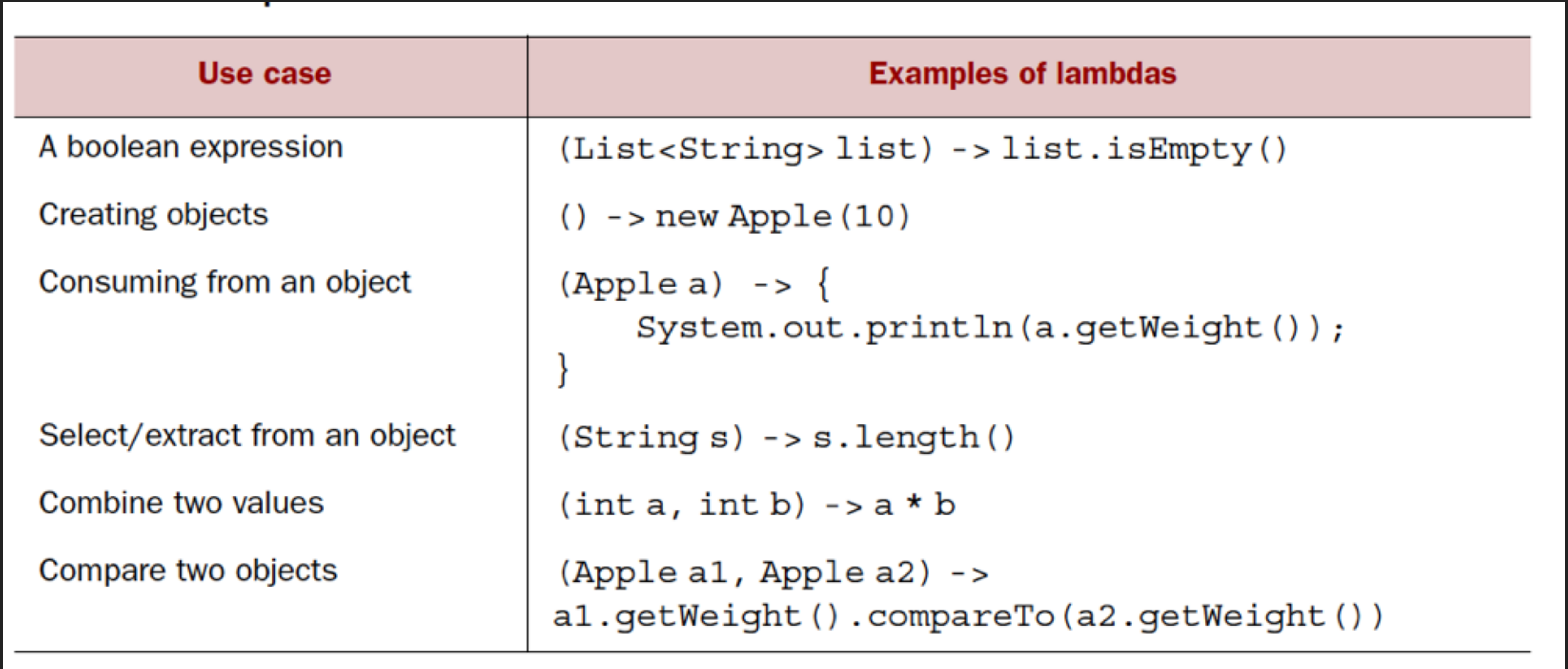
or (note the curly braces for statements)

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

- Examples:

1. ( ) -> { }
2. ( ) -> “Raoul”
3. ( ) -> { return “Mario”;}
4. (Integer i) -> return “Alan” + i;
5. (String s) -> { “Iron Man”; }

Lambda use cases - whenever you use anonymous class

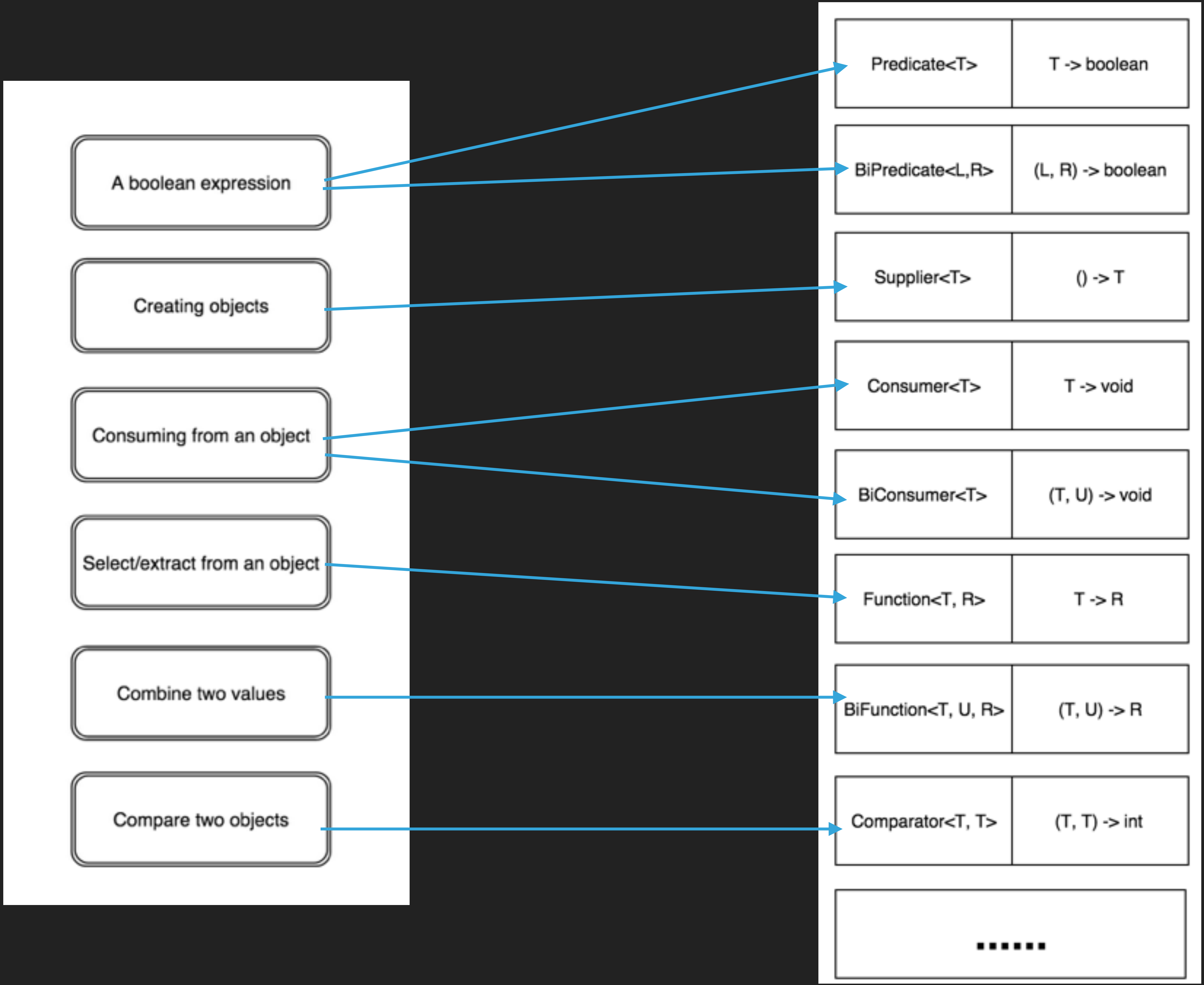


Use case	Examples of lambdas
A boolean expression	(List<String> list) -> list.isEmpty()
Creating objects	() -> new Apple(10)

Consuming from an object	(Apple a) -> {
	    System.out.println(a.getWeight());
	}

Select/extract from an object	(String s) -> s.length()
Combine two values	(int a, int b) -> a * b
Compare two objects	(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())

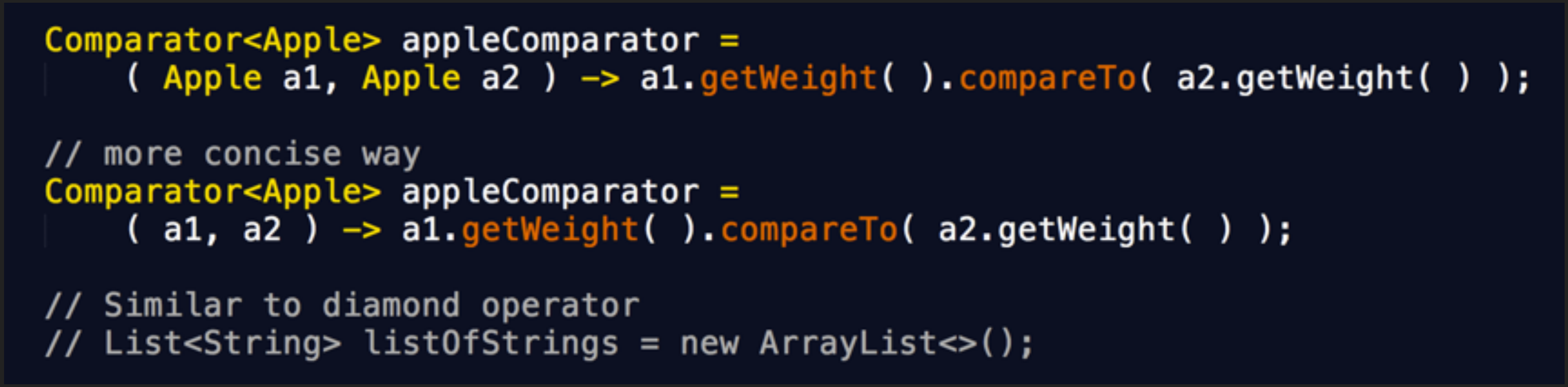
BEHAVIOR PARAMETERIZATION - LAMBDAS



Predicate<T>	T -> boolean
BiPredicate<L,R>	(L, R) -> boolean
Supplier<T>	() -> T
Consumer<T>	T -> void
BiConsumer<T>	(T, U) -> void
Function<T, R>	T -> R
BiFunction<T, U, R>	(T, U) -> R
Comparator<T, T>	(T, T) -> int
....	

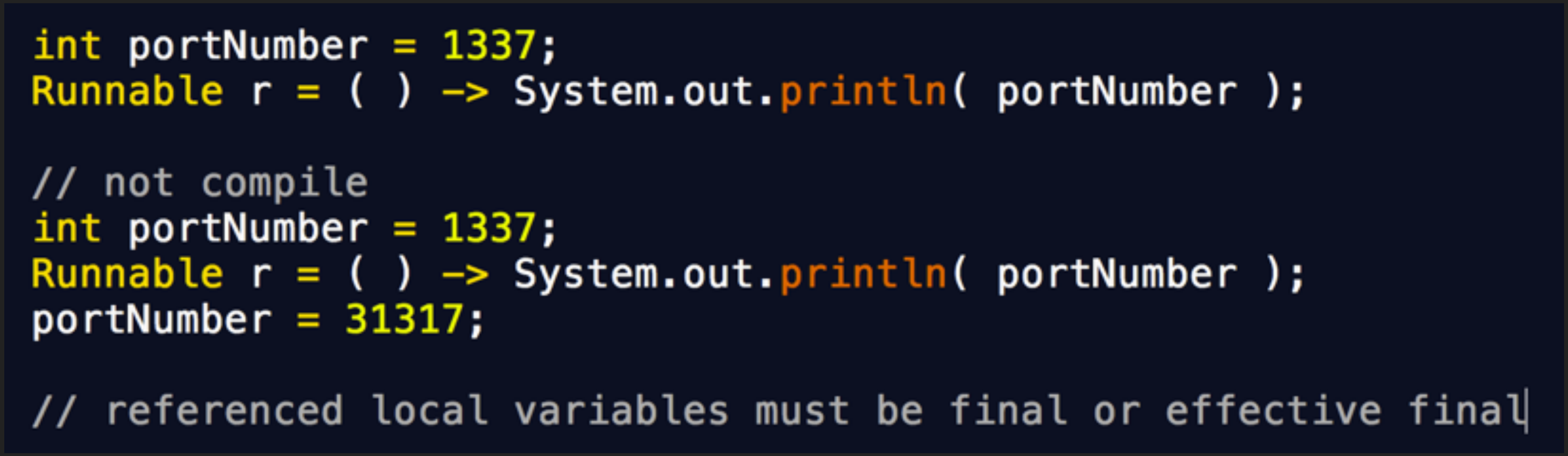
BEHAVIOR PARAMETERIZATION - LAMBDAS

Type reference



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

// more concise way
Comparator<Apple> appleComparator =
        ( a1, a2 ) -> a1.getWeight( ).compareTo( a2.getWeight( ) );

// Similar to diamond operator
// List<String> listOfStrings = new ArrayList<>();
```

Restriction on local variables



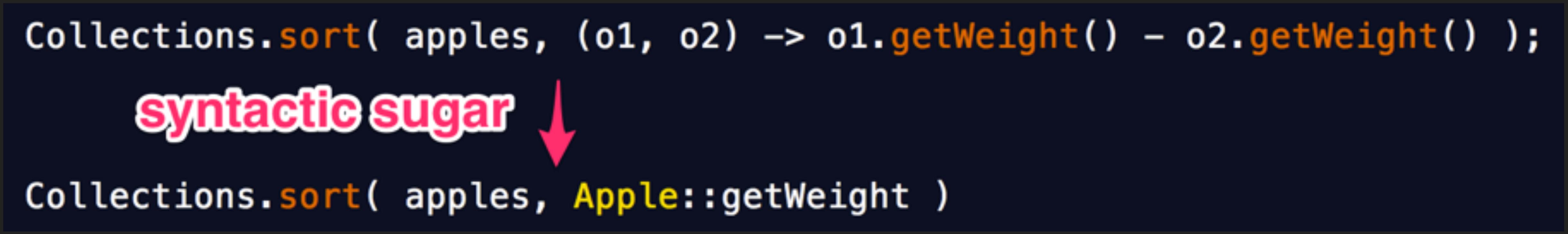
```
int portNumber = 1337;
Runnable r = ( ) -> System.out.println( portNumber );

// not compile
int portNumber = 1337;
Runnable r = ( ) -> System.out.println( portNumber );
portNumber = 31317;

// referenced local variables must be final or effective final
```

OUTLINE

- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Method reference definition

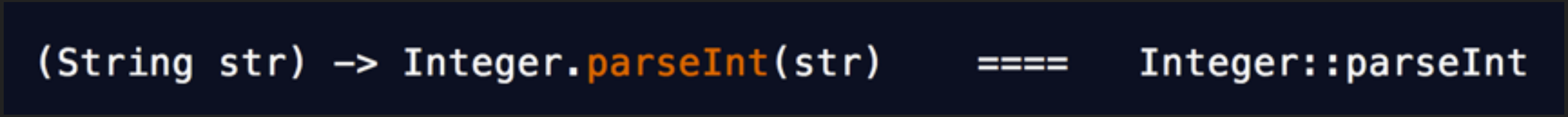
- Definition: syntactic sugar for lambda expressions



```
Collections.sort( apples, (o1, o2) -> o1.getWeight() - o2.getWeight() );

Collections.sort( apples, Apple::getWeight )
```

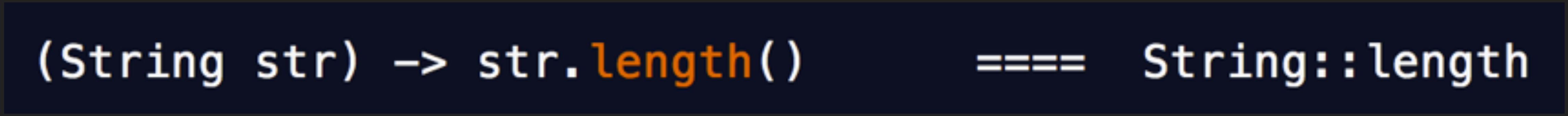
Rules for converting lambda to method reference

- A method reference to a static method



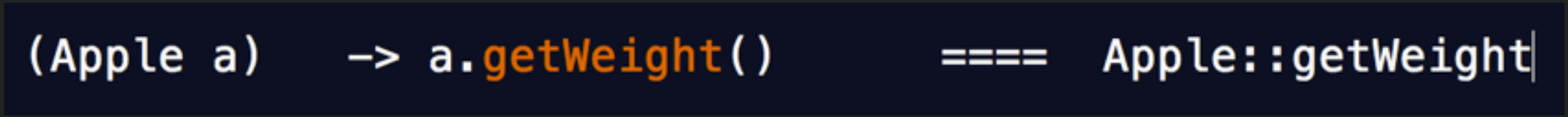
```
(String str) -> Integer.parseInt(str)    ====   Integer::parseInt
```

- A method reference to an instance method of an arbitrary type



```
(String str) -> str.length()        ====   String::length
```

- A method reference to an instance method of an existing object



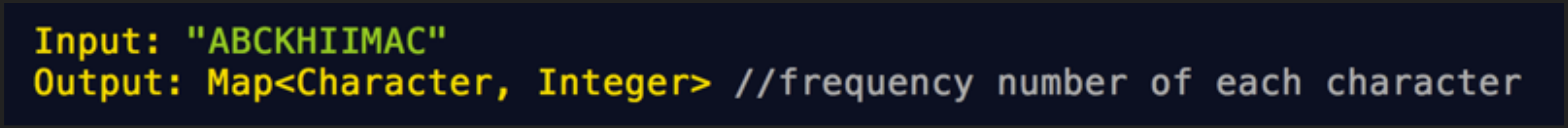
```
(Apple a)   -> a.getWeight()        ====   Apple::getWeight
```

OUTLINE

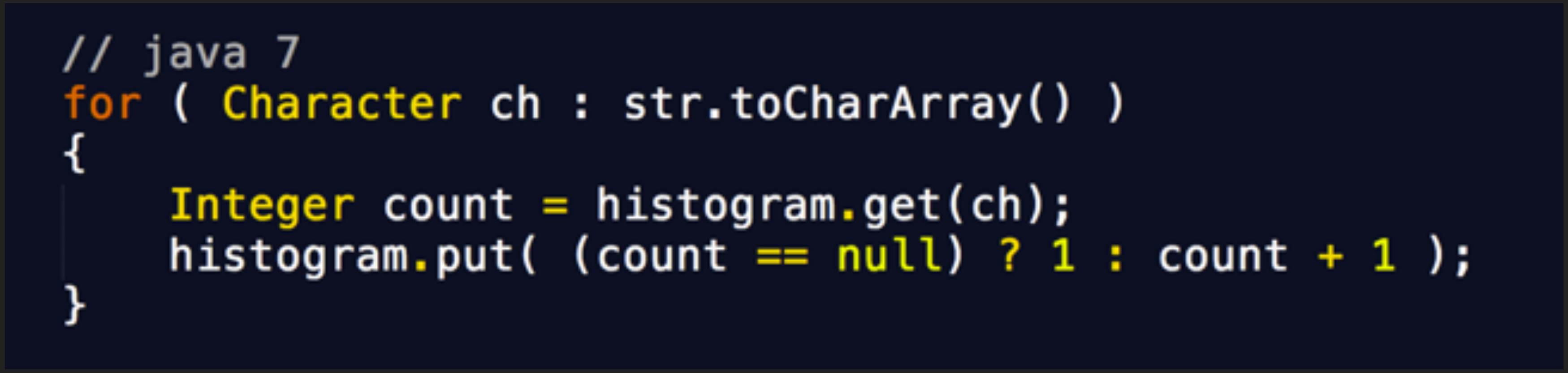
- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Map interface: getOrDefault method

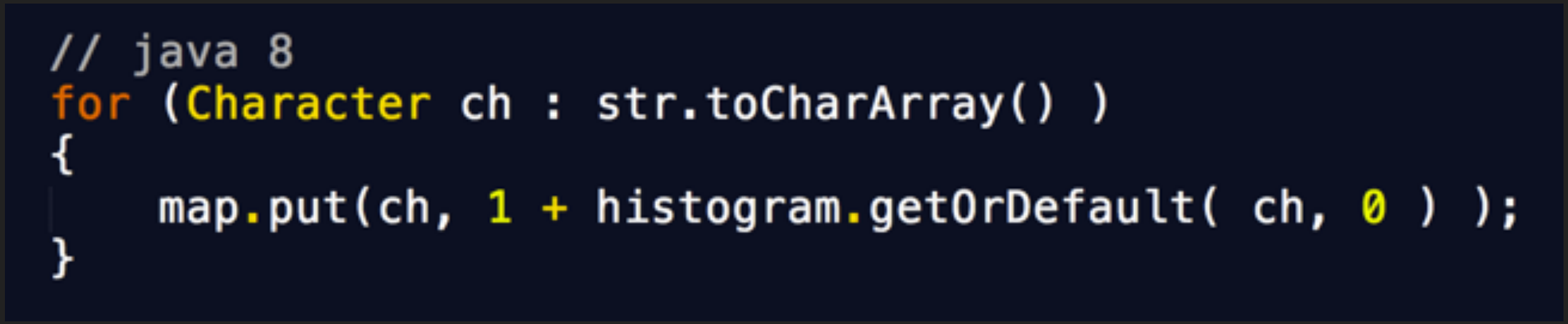
- Definition: getOrDefault(K key, V defaultValue)
- Scenario: get a value (may not exist) from a map, do some calculation and put it back



```
Input: "ABCKHIIMAC"
Output: Map<Character, Integer> //frequency number of each character
```



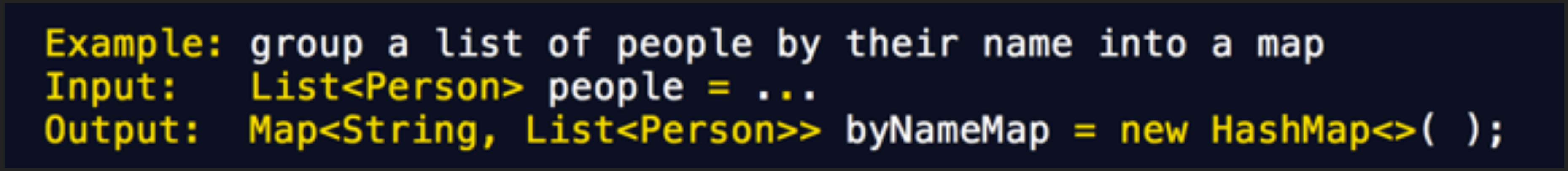
```
// java 7
for ( Character ch : str.toCharArray() )
{
    Integer count = histogram.get(ch);
    histogram.put( (count == null) ? 1 : count + 1 );
}
```



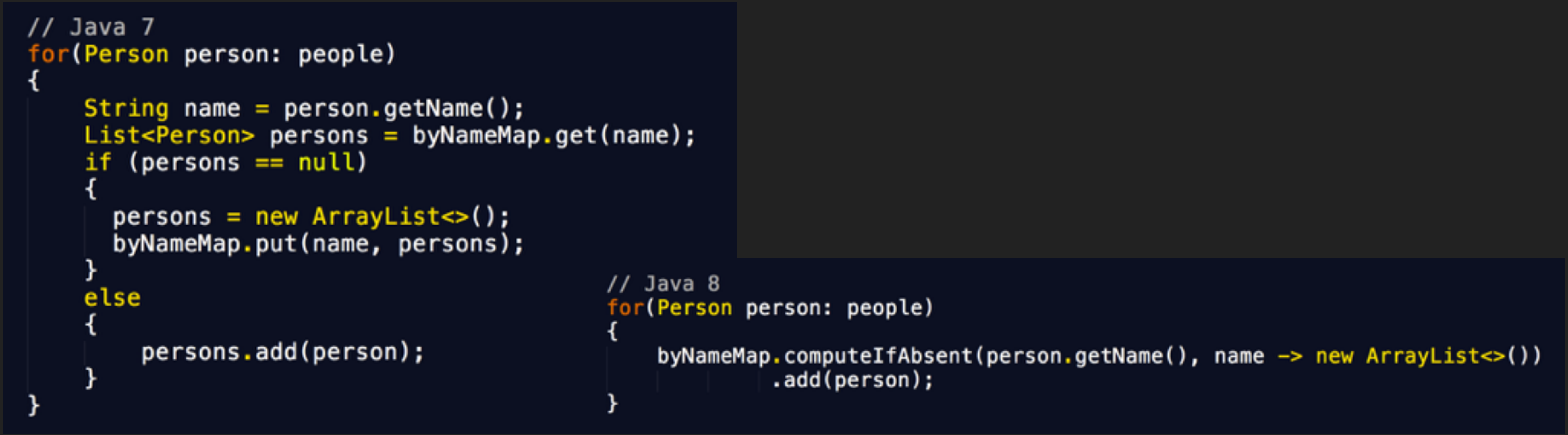
```
// java 8
for (Character ch : str.toCharArray() )
{
    map.put(ch, 1 + histogram.getOrDefault( ch, 0 ) );
}
```

Map interface: computeIfAbsent method

- Definition: computeIfAbsent(K key, Function mapping)
- Scenario: if the key does not exist, compute a value for it.



```
Example: group a list of people by their name into a map
Input:   List<Person> people = ...
Output:  Map<String, List<Person>> byNameMap = new HashMap<>( );
```

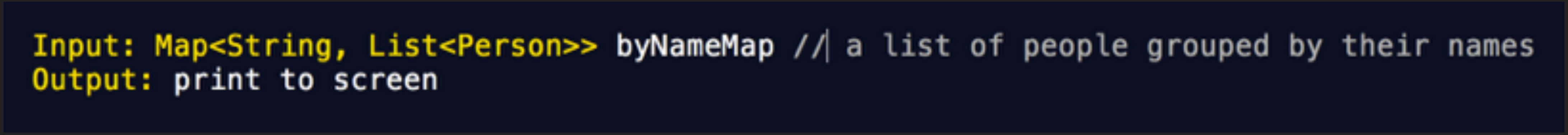


```
// Java 7
for(Person person: people)
{
    String name = person.getName();
    List<Person> persons = byNameMap.get(name);
    if (persons == null)
    {
      persons = new ArrayList<>();
      byNameMap.put(name, persons);
    }
    else
    {
        persons.add(person);
    }
}
```

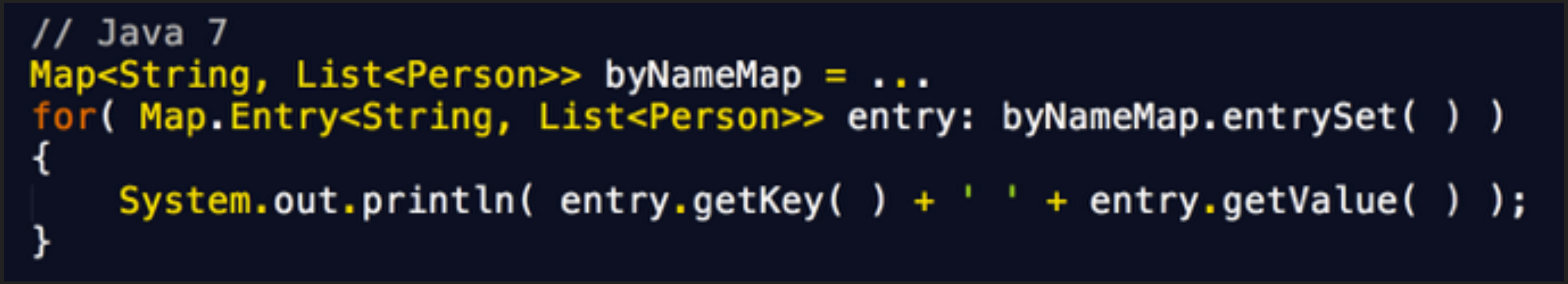
```
// Java 8
for(Person person: people)
{
    byNameMap.computeIfAbsent(person.getName(), name -> new ArrayList<>())
                    .add(person);
}
```

Map interface: forEach method

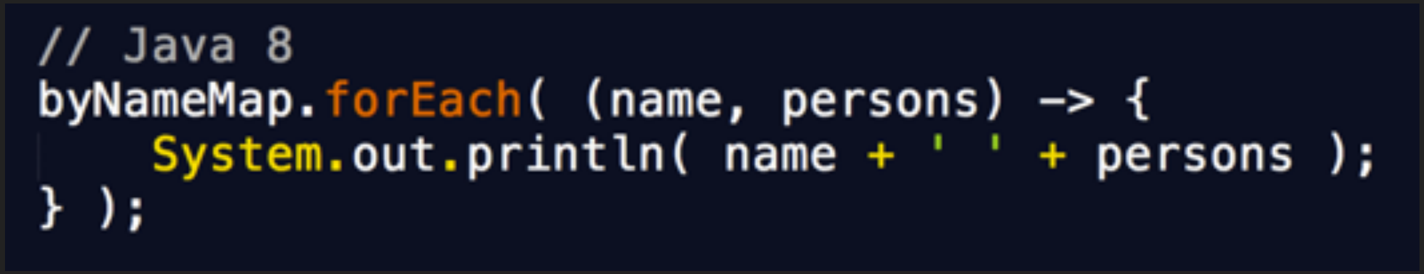
- Definition: forEach(Consumer con)
- Scenario: loop through a map



```
Input: Map<String, List<Person>> byNameMap // a list of people grouped by their names
Output: print to screen
```



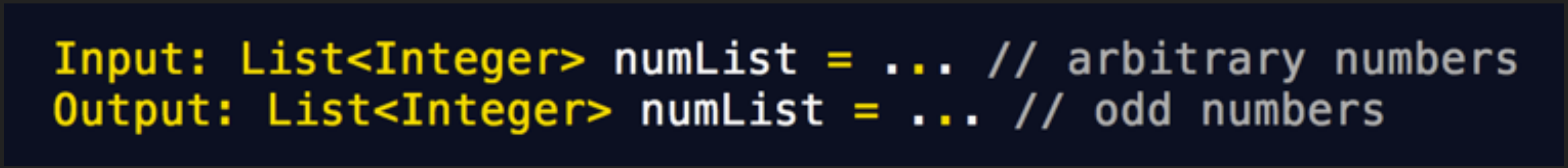
```
// Java 7
Map<String, List<Person>> byNameMap = ...
for( Map.Entry<String, List<Person>> entry: byNameMap.entrySet( ) )
{
    System.out.println( entry.getKey( ) + ' ' + entry.getValue( ) );
}
```



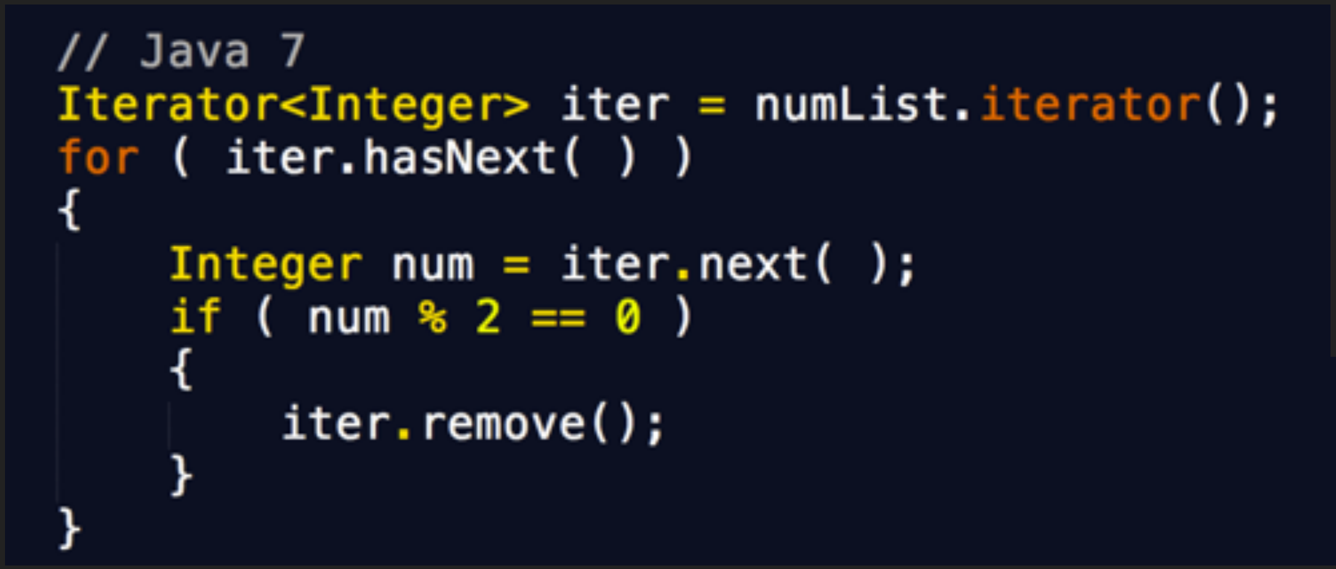
```
// Java 8
byNameMap.forEach( (name, persons) -> {
    System.out.println( name + ' ' + persons );
} );
```

Collections interface: removeIf method

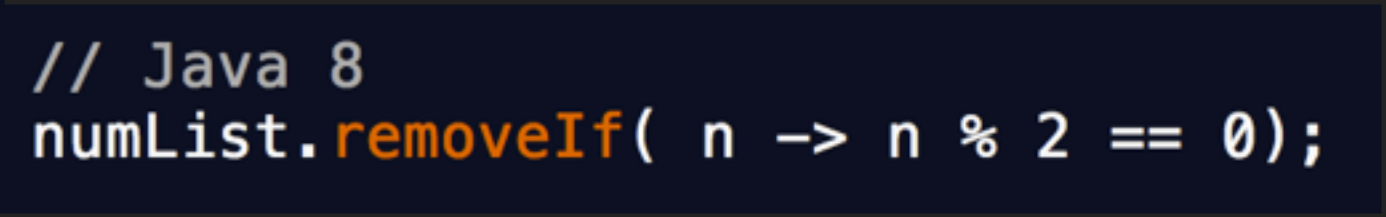
- Definition: removeIf(Predicate filter)
- Scenario: remove an element from the list if specific condition is met



```
Input: List<Integer> numList = ... // arbitrary numbers
Output: List<Integer> numList = ... // odd numbers
```



```
// Java 7
Iterator<Integer> iter = numList.iterator();
for ( iter.hasNext( ) )
{
    Integer num = iter.next( );
    if ( num % 2 == 0 )
    {
        iter.remove();
    }
}
```



```
// Java 8
numList.removeIf( n -> n % 2 == 0);
```

SIMPLE SYNTACTIC SUGAR - NEW METHODS INSIDE COLLECTIONS

Other method

- List.sort(Comparator)
- Map.putIfAbsent()
- Map.replace() / replaceAll()
- Map.merge()
- Map.compute() / computeIfAbsent() / computeIfPresent()

OUTLINE

- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

OUTLINE

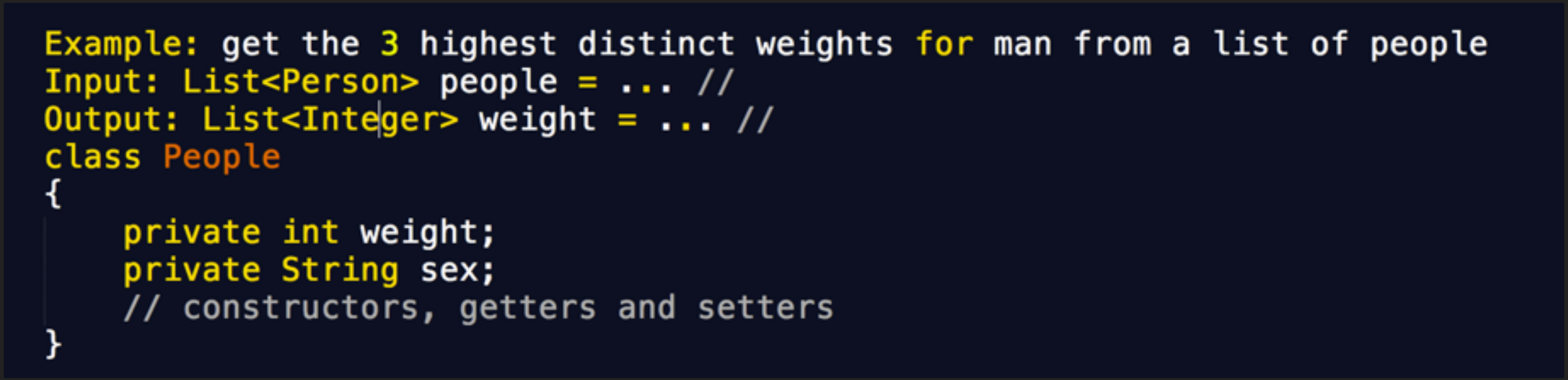
- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Stream API

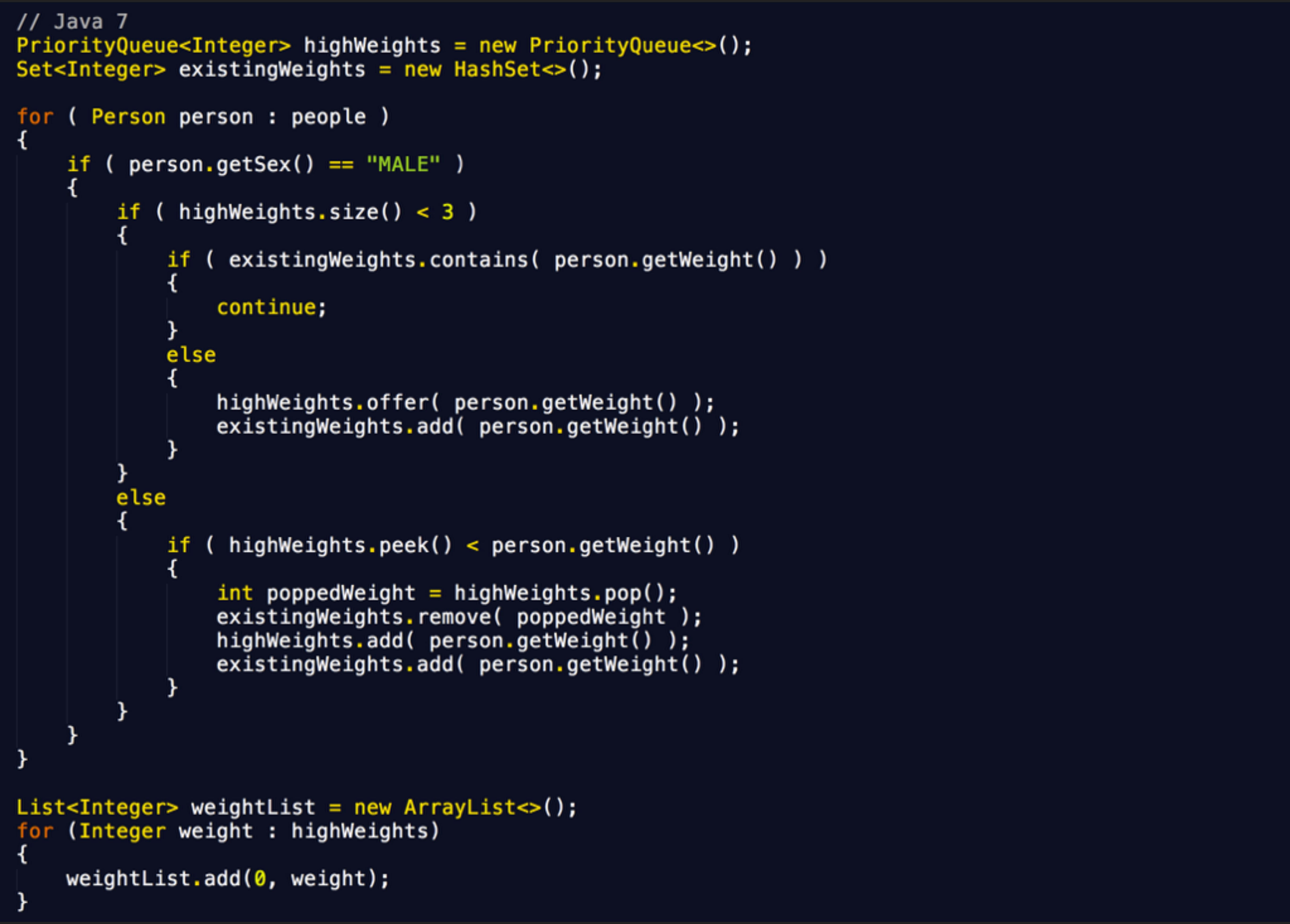
- What's wrong with collections?
  - Much business logic entails database-like operations such as grouping a list by category / find the most expensive dish. (Usually implemented with iterators, could we do it declaratively?)
  - Big data requires us to utilize multicore processor more frequently. (Usually implemented with fork/join framework introduced in Java 7. Could we save some effort?)

Stream API

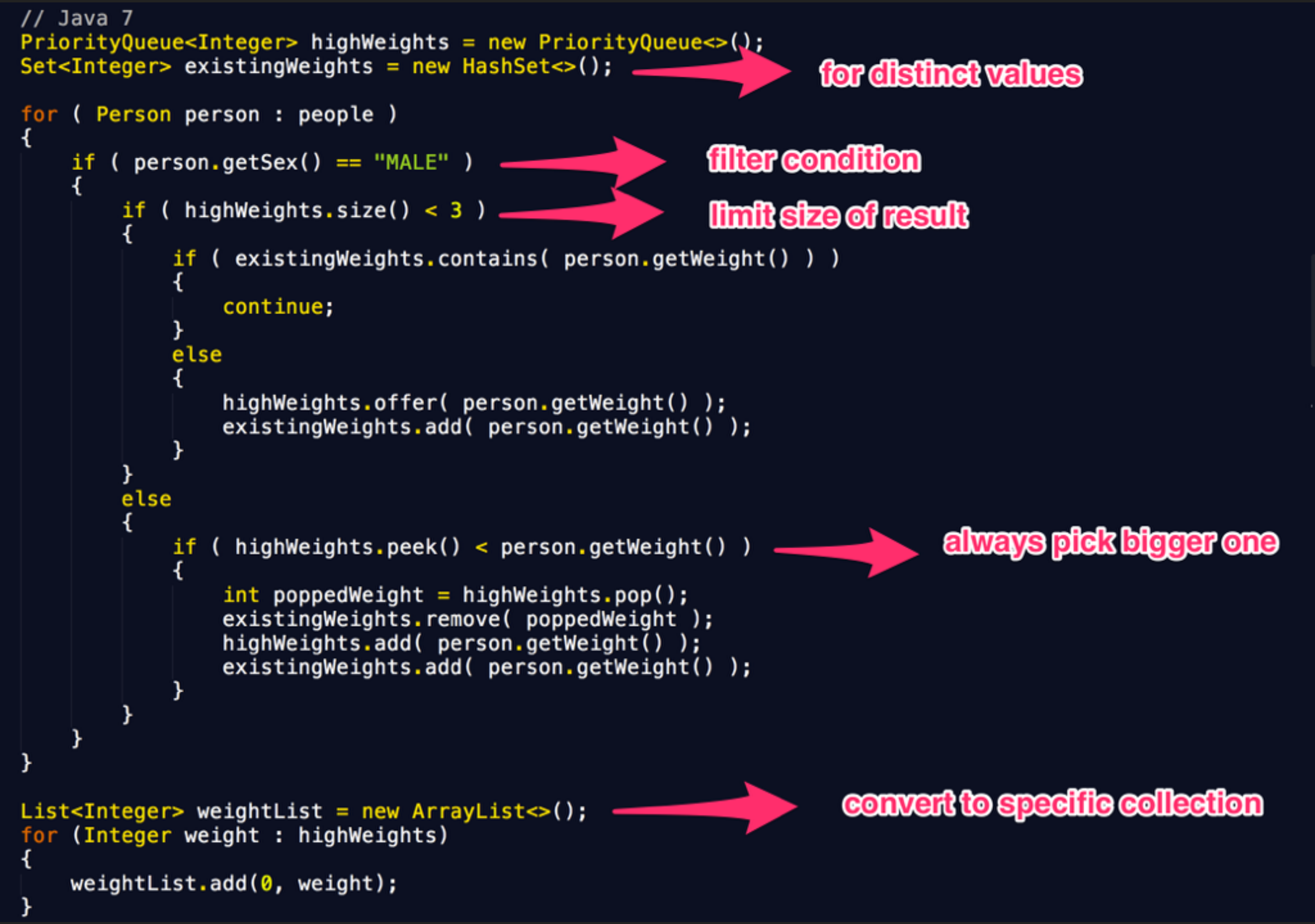
- Def: fancy iterators over collections
- Scenario:



```
Example: get the 3 highest distinct weights for man from a list of people
Input: List<Person> people = ... //
Output: List<Integer> weight = ... //
class People
{
    private int weight;
    private String sex;
    // constructors, getters and setters
}
```



```java
// Java 7
PriorityQueue<Integer> highWeights = new PriorityQueue<>();
Set<Integer> existingWeights = new HashSet<>();

for ( Person person : people )
{
    if ( person.getSex() == "MALE" )
    {
        if ( highWeights.size() < 3 )
        {
            if ( existingWeights.contains( person.getWeight() ) )
            {
                continue;
            }
            else
            {
                highWeights.offer( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
        else
        {
            if ( highWeights.peek() < person.getWeight() )
            {
                int poppedWeight = highWeights.pop();
                existingWeights.remove( poppedWeight );
                highWeights.add( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
    }
}

List<Integer> weightList = new ArrayList<>();
for (Integer weight : highWeights)
{
    weightList.add(0, weight);
}
```



```
// Java 7
PriorityQueue<Integer> highWeights = new PriorityQueue<>();
Set<Integer> existingWeights = new HashSet<>();

for ( Person person : people )
{
    if ( person.getSex() == "MALE" )
    {
        if ( highWeights.size() < 3 )
        {
            if ( existingWeights.contains( person.getWeight() ) )
            {
                continue;
            }
            else
            {
                highWeights.offer( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
        else
        {
            if ( highWeights.peek() < person.getWeight() )
            {
                int poppedWeight = highWeights.pop();
                existingWeights.remove( poppedWeight );
                highWeights.add( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
    }
}

List<Integer> weightList = new ArrayList<>();
for (Integer weight : highWeights)
{
    weightList.add(0, weight);
}
```

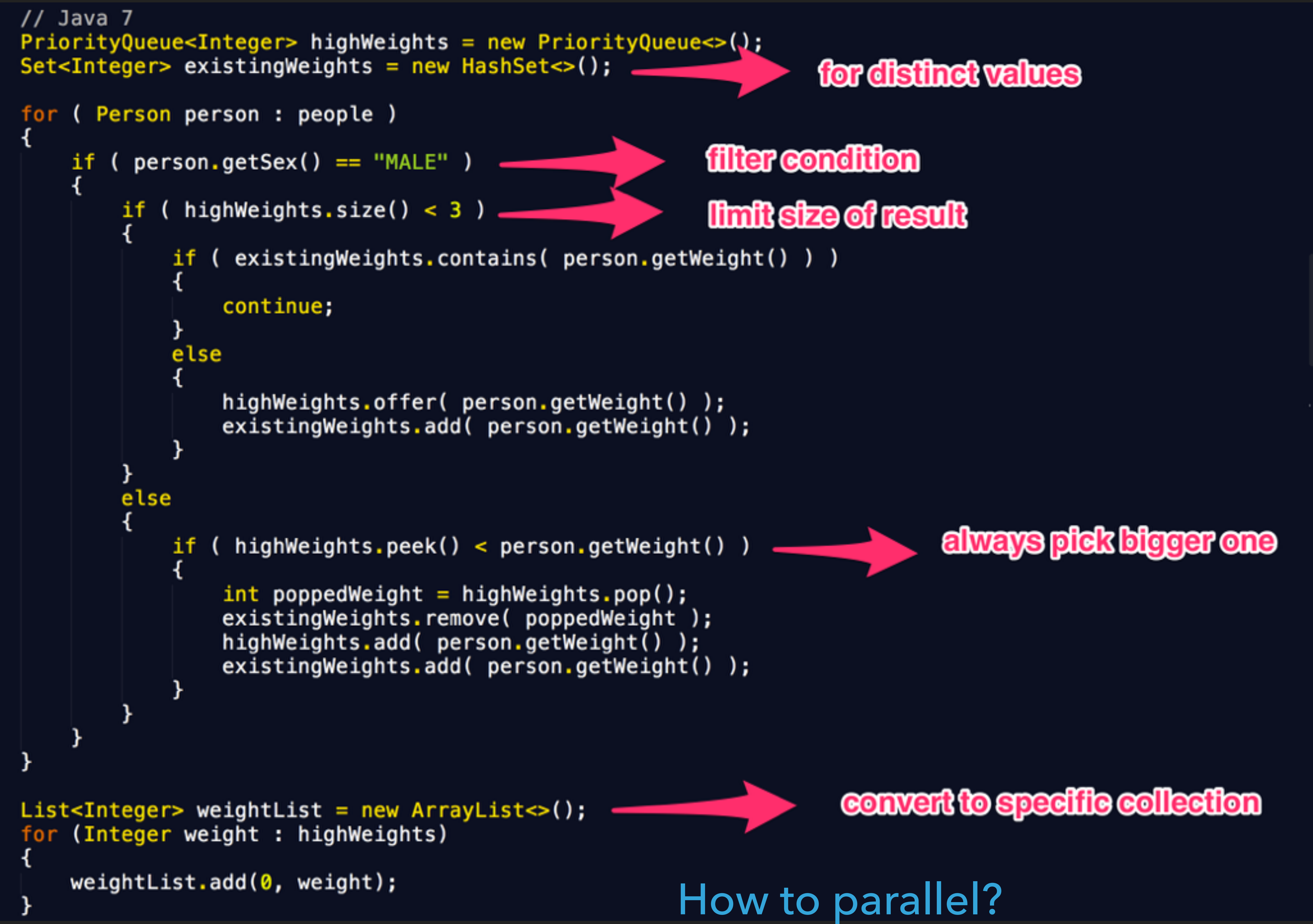
for distinct values

filter condition

limit size of result

always pick bigger one

convert to specific collection



```java
// Java 7
PriorityQueue<Integer> highWeights = new PriorityQueue<>();
Set<Integer> existingWeights = new HashSet<>();

for ( Person person : people )
{
    if ( person.getSex() == "MALE" )
    {
        if ( highWeights.size() < 3 )
        {
            if ( existingWeights.contains( person.getWeight() ) )
            {
                continue;
            }
            else
            {
                highWeights.offer( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
        else
        {
            if ( highWeights.peek() < person.getWeight() )
            {
                int poppedWeight = highWeights.pop();
                existingWeights.remove( poppedWeight );
                highWeights.add( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
    }
}

List<Integer> weightList = new ArrayList<>();
for (Integer weight : highWeights)
{
    weightList.add(0, weight);
}
```

for distinct values

filter condition

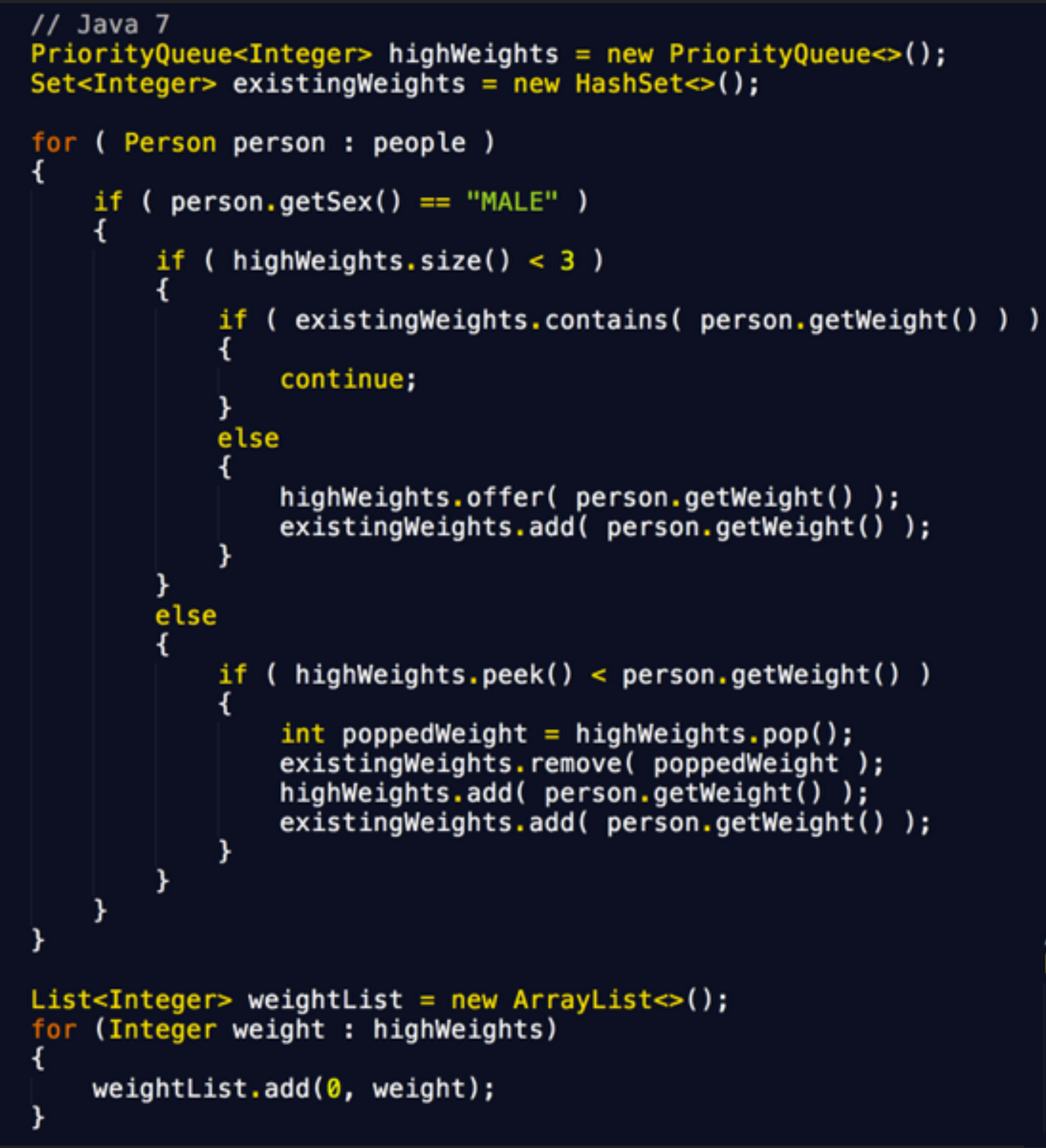
limit size of result

always pick bigger one

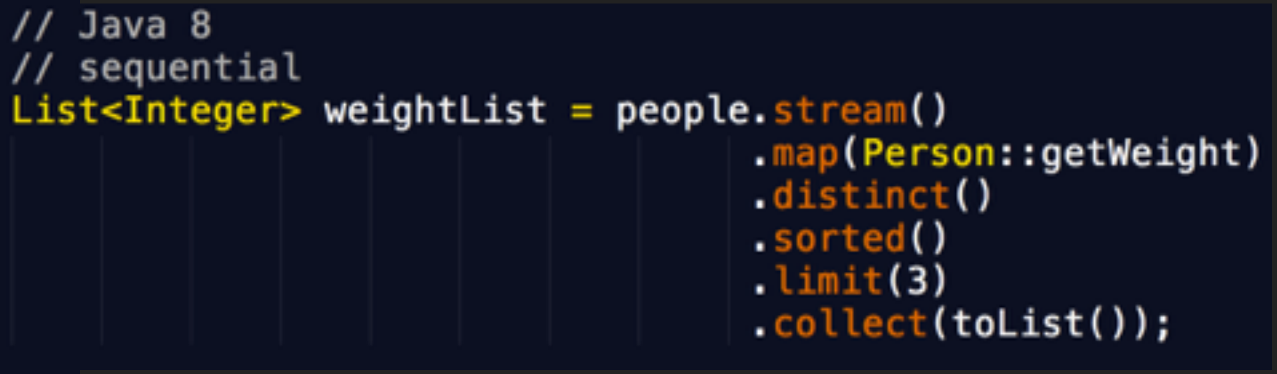
convert to specific collection

How to parallel?

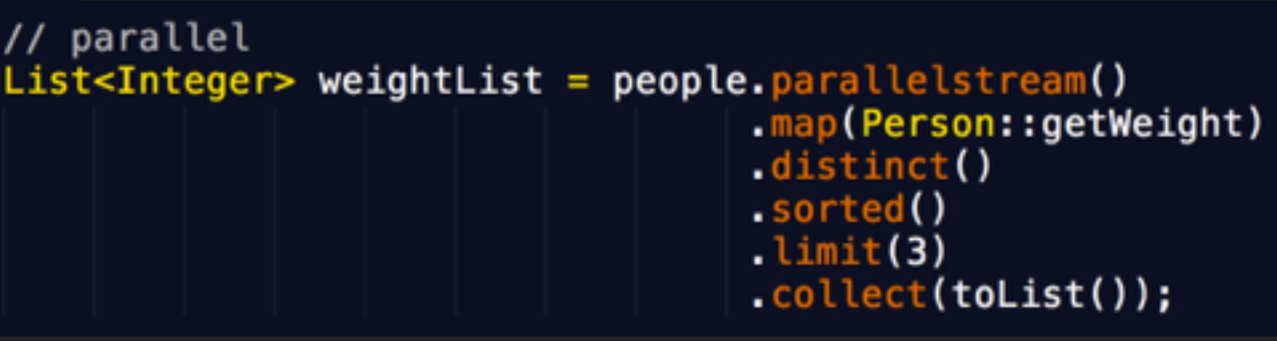
FUNCTIONAL PROGRAMMING - STREAM API



```
// Java 7
PriorityQueue<Integer> highWeights = new PriorityQueue<>();
Set<Integer> existingWeights = new HashSet<>();

for ( Person person : people )
{
    if ( person.getSex() == "MALE" )
    {
        if ( highWeights.size() < 3 )
        {
            if ( existingWeights.contains( person.getWeight() ) )
            {
                continue;
            }
            else
            {
                highWeights.offer( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
        else
        {
            if ( highWeights.peek() < person.getWeight() )
            {
                int poppedWeight = highWeights.pop();
                existingWeights.remove( poppedWeight );
                highWeights.add( person.getWeight() );
                existingWeights.add( person.getWeight() );
            }
        }
    }
}

List<Integer> weightList = new ArrayList<>();
for (Integer weight : highWeights)
{
    weightList.add(0, weight);
}
```

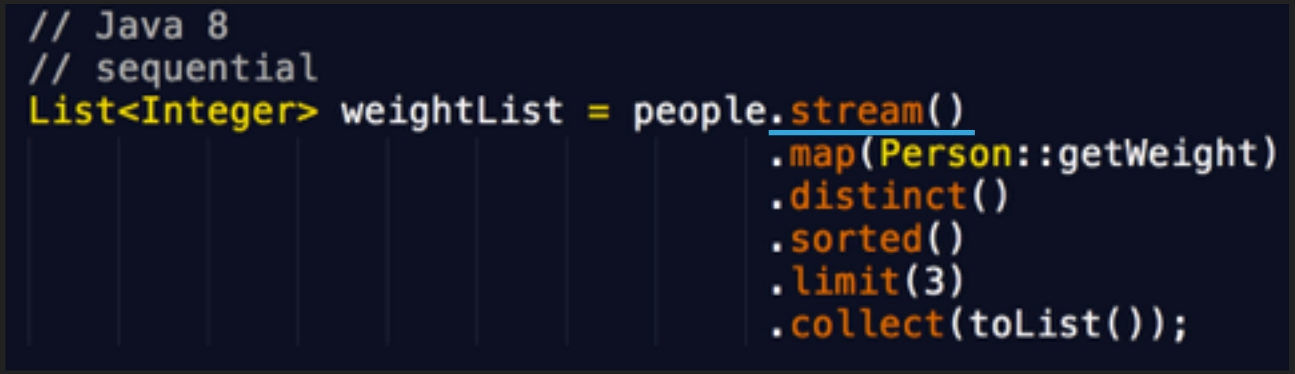


```
// Java 8
// sequential
List<Integer> weightList = people.stream()
                                 .map(Person::getWeight)
                                 .distinct()
                                 .sorted()
                                 .limit(3)
                                 .collect(toList());
```

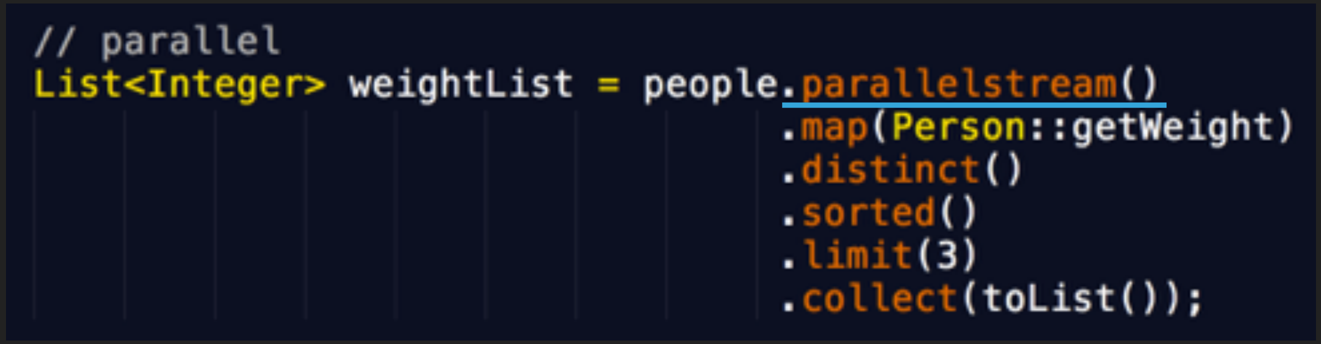


```
// parallel
List<Integer> weightList = people.parallelstream()
                                 .map(Person::getWeight)
                                 .distinct()
                                 .sorted()
                                 .limit(3)
                                 .collect(toList());
```

FUNCTIONAL PROGRAMMING - STREAM API



```
// Java 8
// sequential
List<Integer> weightList = people.stream()
                                 .map(Person::getWeight)
                                 .distinct()
                                 .sorted()
                                 .limit(3)
                                 .collect(toList());
```

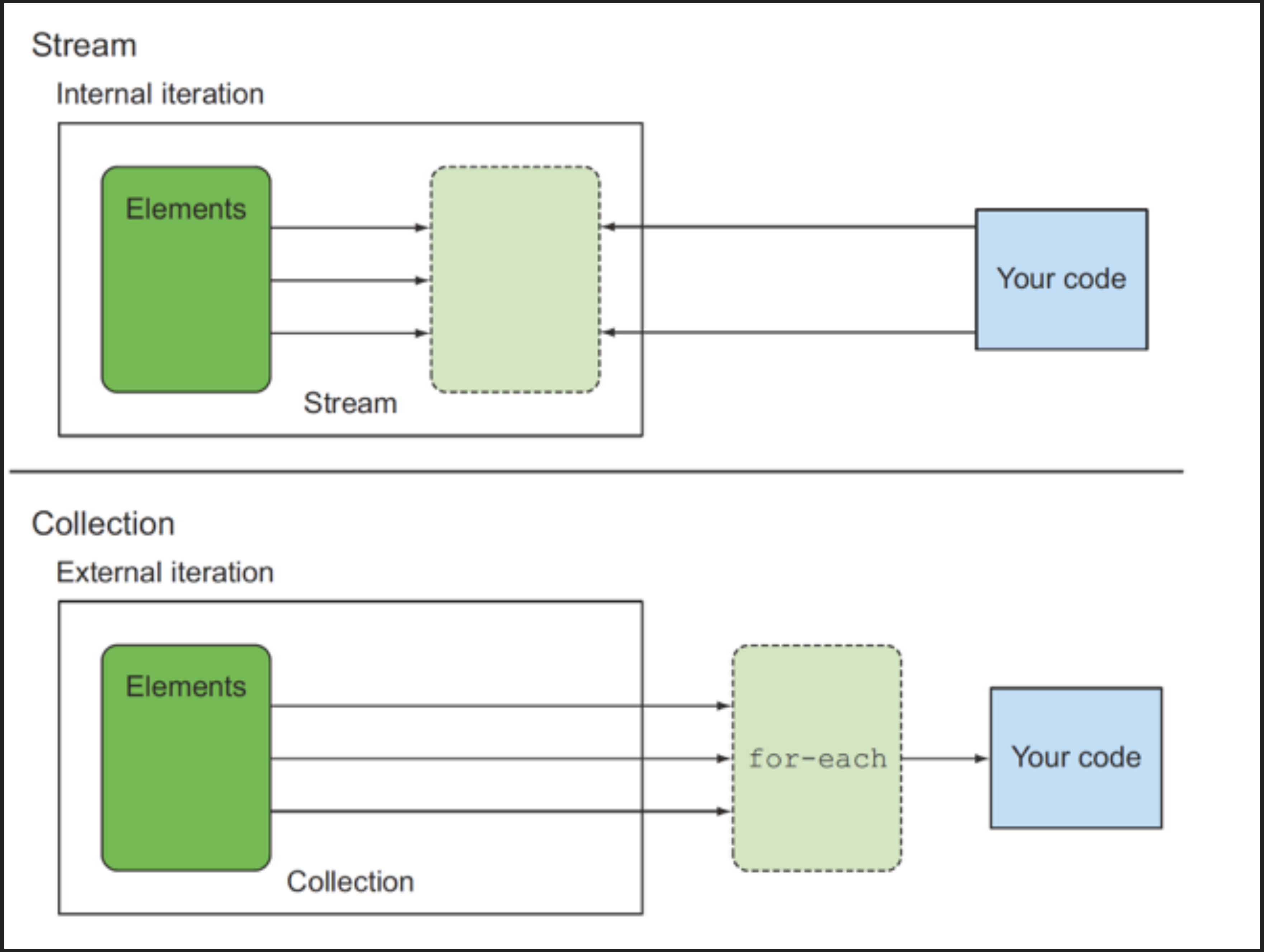


```
// parallel
List<Integer> weightList = people.parallelstream()
                                 .map(Person::getWeight)
                                 .distinct()
                                 .sorted()
                                 .limit(3)
                                 .collect(toList());
```

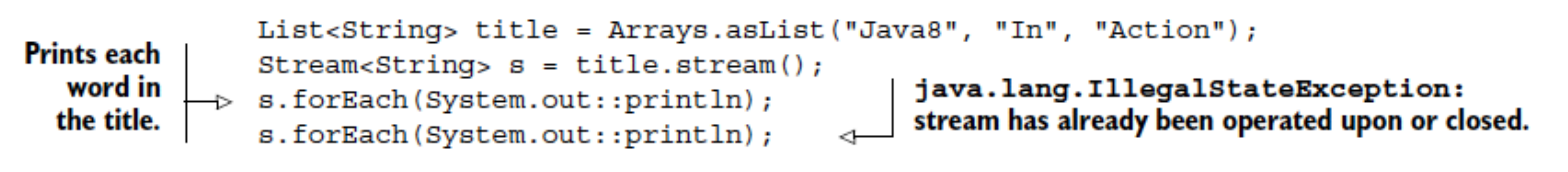
Parallelization realized by Java 7’s fork/join framework underneath

Stream Definition

- Stream definition: fancy “internal” iterators over collections



Notice: iterable only once



```
List<String> title = Arrays.asList("Java8", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);
```

Prints each word in the title.

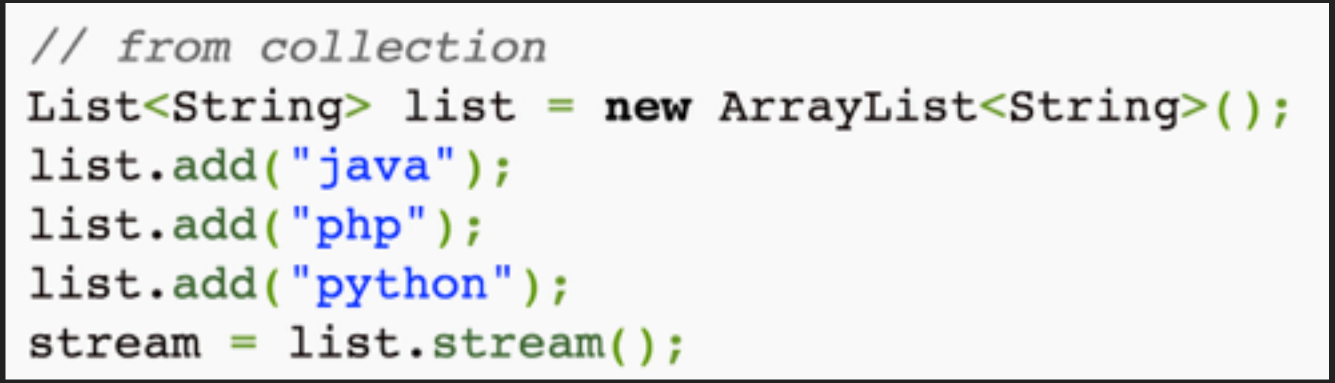
java.lang.IllegalStateException: stream has already been operated upon or closed.

OUTLINE

- What will Java 8 give us
- Behavior parameterization
  - More comprehensive functional interface
  - Lambda expression
  - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
  - Intuition
  - Creating, intermediate and terminal operations
  - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

How to create stream?

- From arrays
- From collections



```
// from collection
List<String> list = new ArrayList<String>();
list.add("java");
list.add("php");
list.add("python");
stream = list.stream();
```

- Custom generators – Stream.generate() / iterate() method
- From other popular APIs

Stream operations

- intermediate operations
- terminal operations

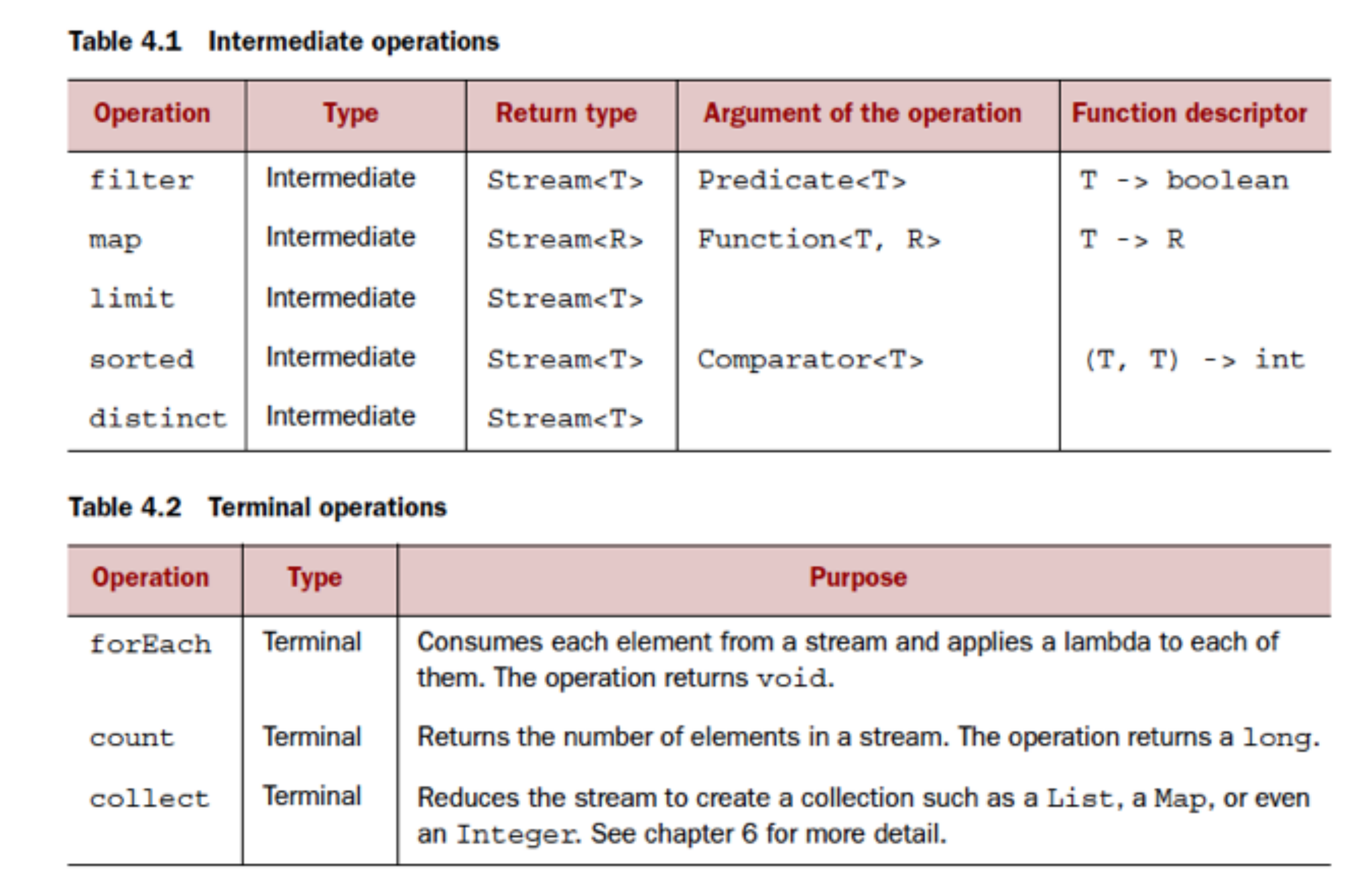


Table 4.1 Intermediate operations

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>`		

Table 4.2 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.

OUTLINE

- What will Java 8 give us
- Behavior parameterization
  - More comprehensive functional interface
  - Lambda expression
  - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
  - Intuition
  - Creating, intermediate and terminal operations
  - Use cases
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Use cases: whenever you want to perform database-like operations

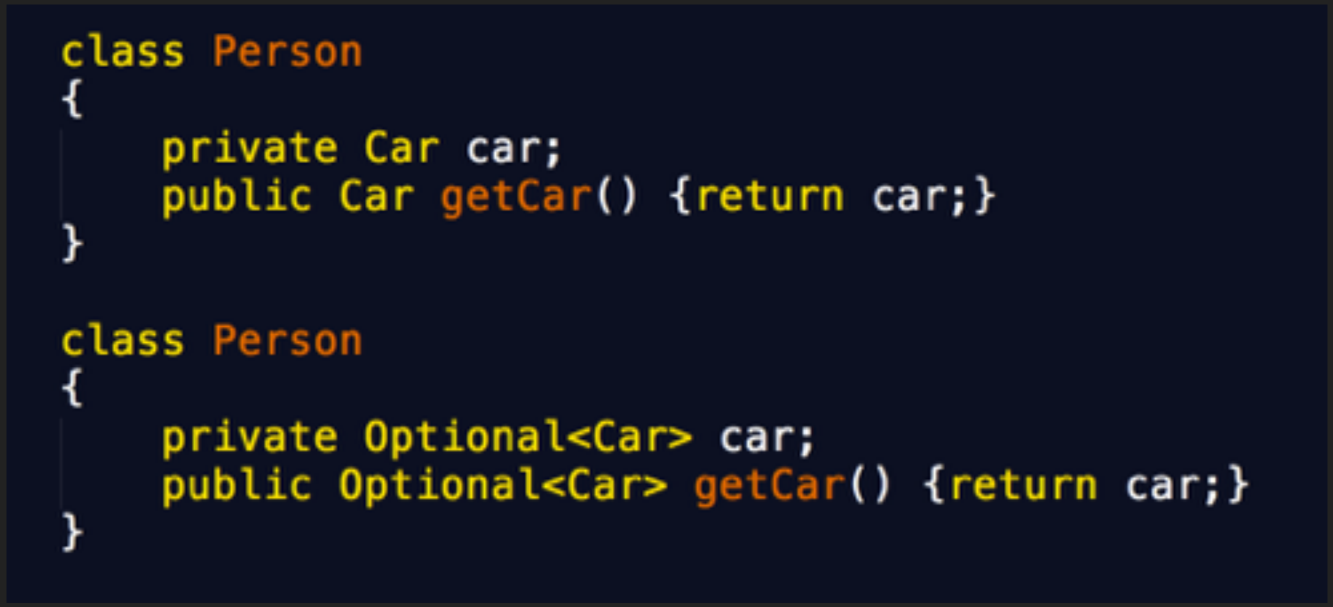
- Group/Multi-level group
- Filter
- Sum/Max/Min/Average/Distinct/Count
- Extracting specific properties
- ……

OUTLINE

- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Use cases
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Optional definition:

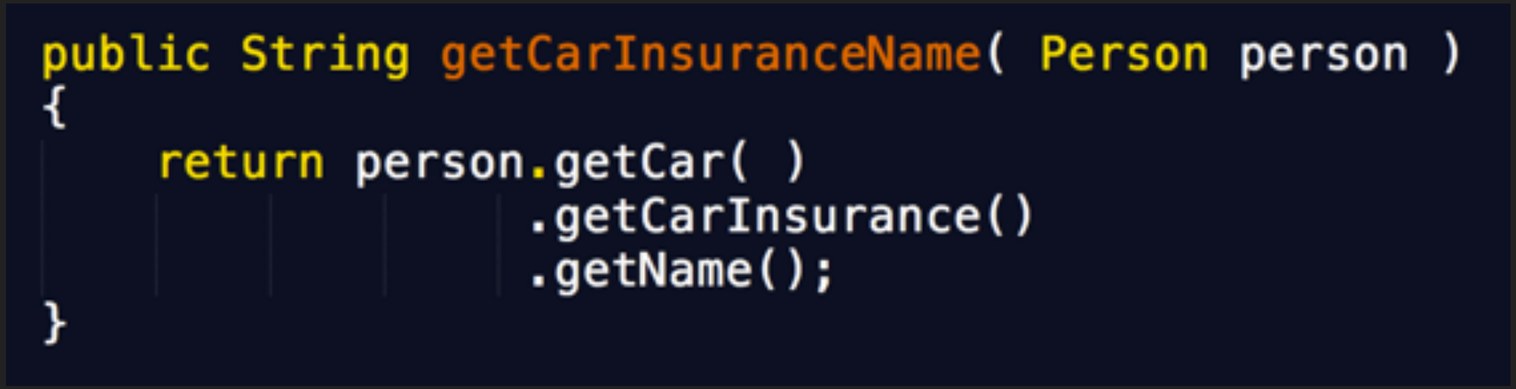
- Def: a container may or may not cannot value - just like reference



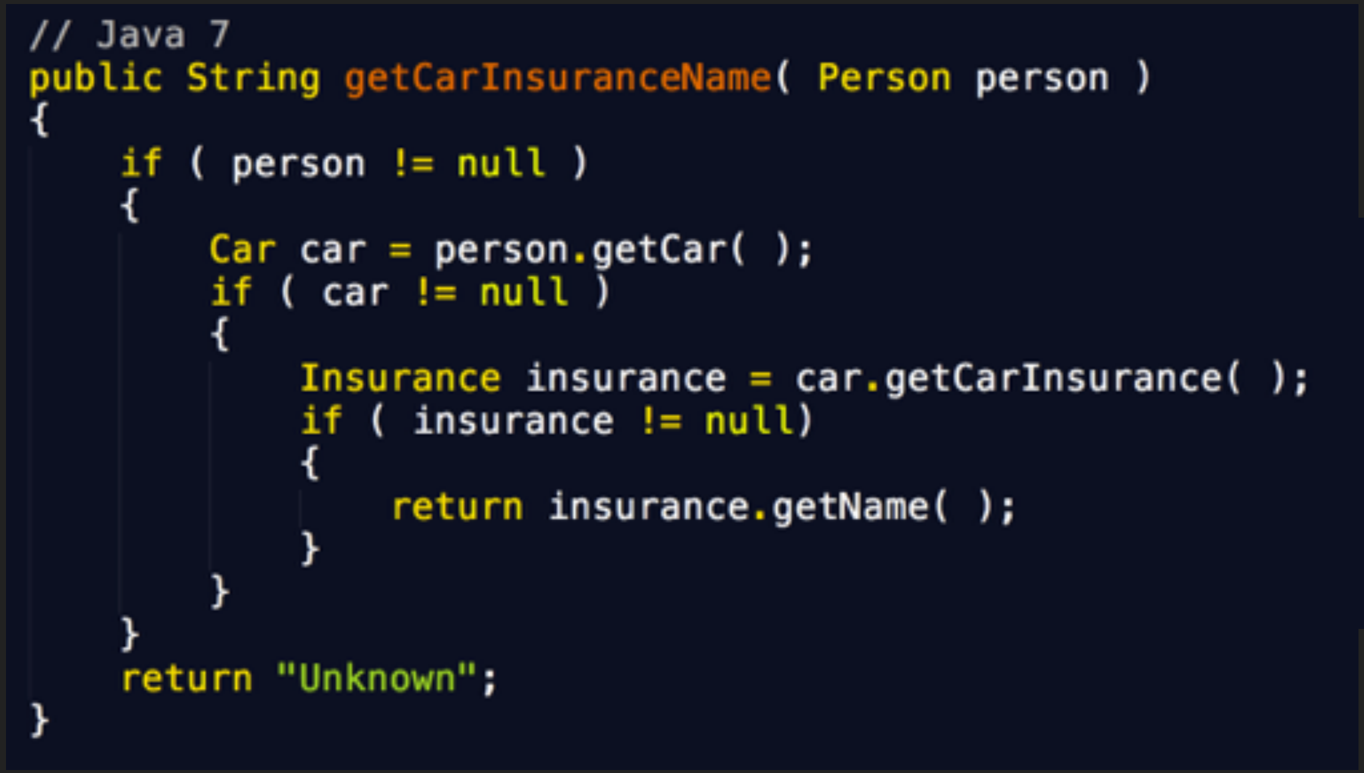
```
class Person
{
    private Car car;
    public Car getCar() {return car;}
}

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

- Benefit 1:
  - NullPointerException will always be thrown out during runtime
  - Optional enforces “empty checking” in grammar during compile time
- Benefit 2:
  - Optional interface supports a set of methods makes handling “empty case” easy

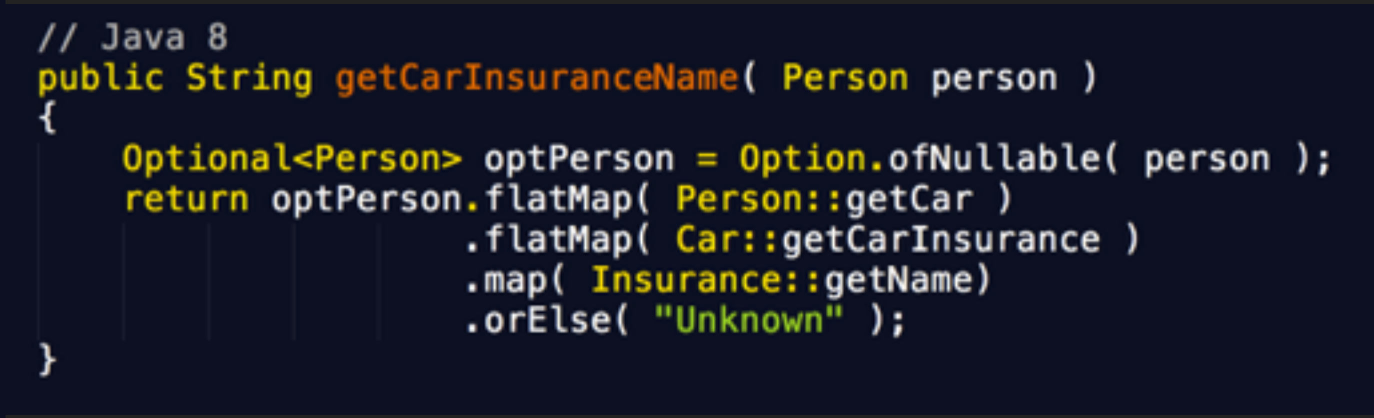
Optional use case :



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



```
// Java 7
public String getCarInsuranceName( Person person )
{
    if ( person != null )
    {
        Car car = person.getCar( );
        if ( car != null )
        {
            Insurance insurance = car.getCarInsurance( );
            if ( insurance != null)
            {
                return insurance.getName( );
            }
        }
    }
    return "Unknown";
}
```



```
// Java 8
public String getCarInsuranceName( Person person )
{
    Optional<Person> optPerson = Option.ofNullable( person );
    return optPerson.flatMap( Person::getCar )
                    .flatMap( Car::getCarInsurance )
                    .map( Insurance::getName)
                    .orElse( "Unknown" );
}
```

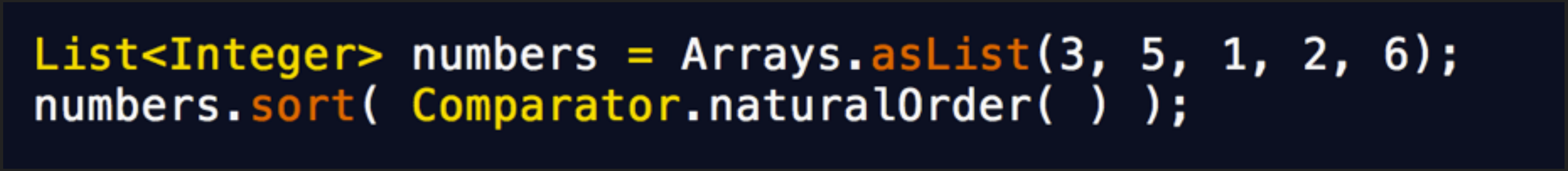
More use cases:

- http://www.nurkiewicz.com/2013/08/optional-in-java-8-cheat-sheet.html

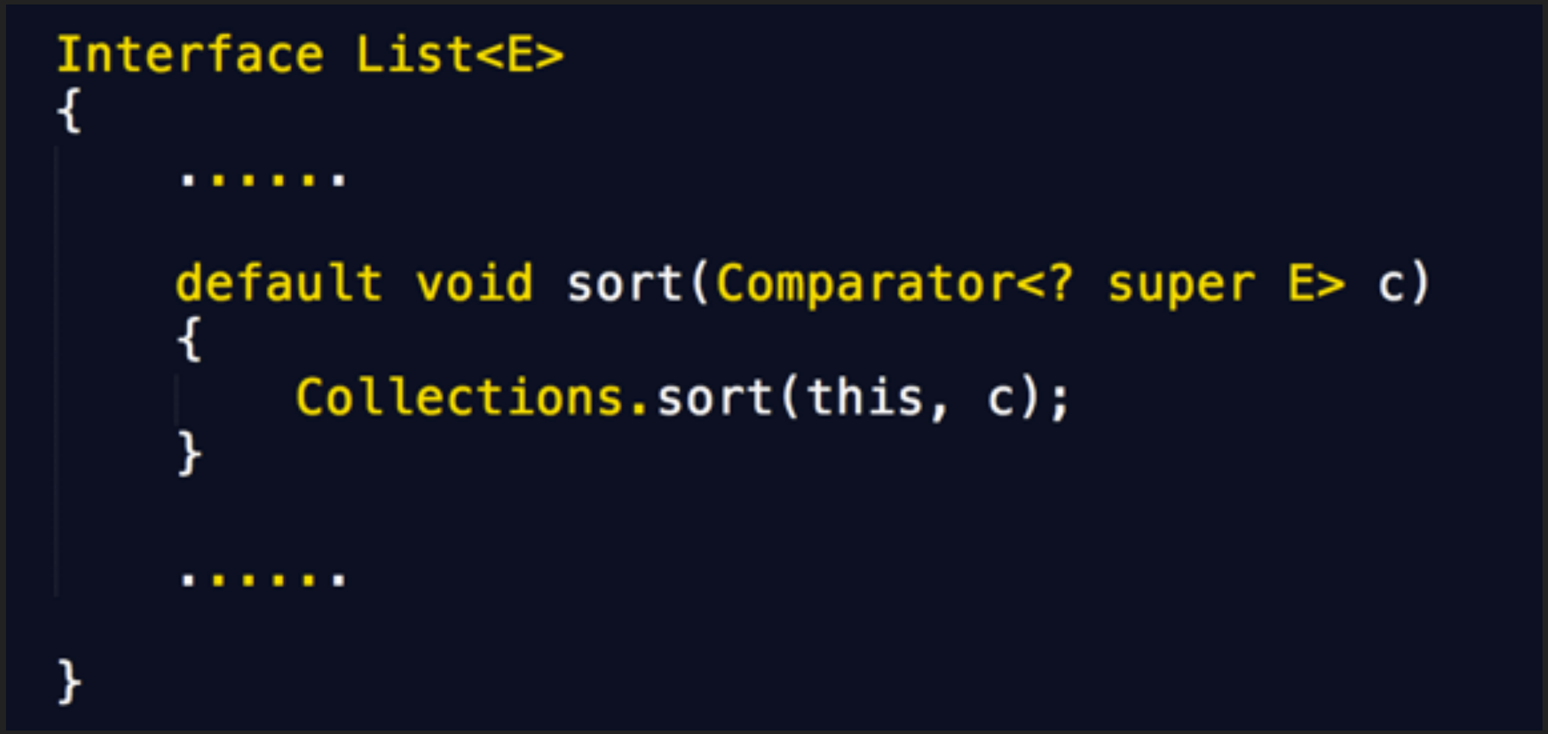
OUTLINE

- What will Java 8 give us
- Behavior parameterization
  - More comprehensive functional interface
  - Lambda expression
  - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
  - Intuition
  - Intermediate and terminal operations
  - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Default method



```
List<Integer> numbers = Arrays.asList(3, 5, 1, 2, 6);
numbers.sort( Comparator.naturalOrder( ) );
```



```
Interface List<E>
{
    ......

    default void sort(Comparator<? super E> c)
    {
        Collections.sort(this, c);
    }

    ......

}
```

Default method

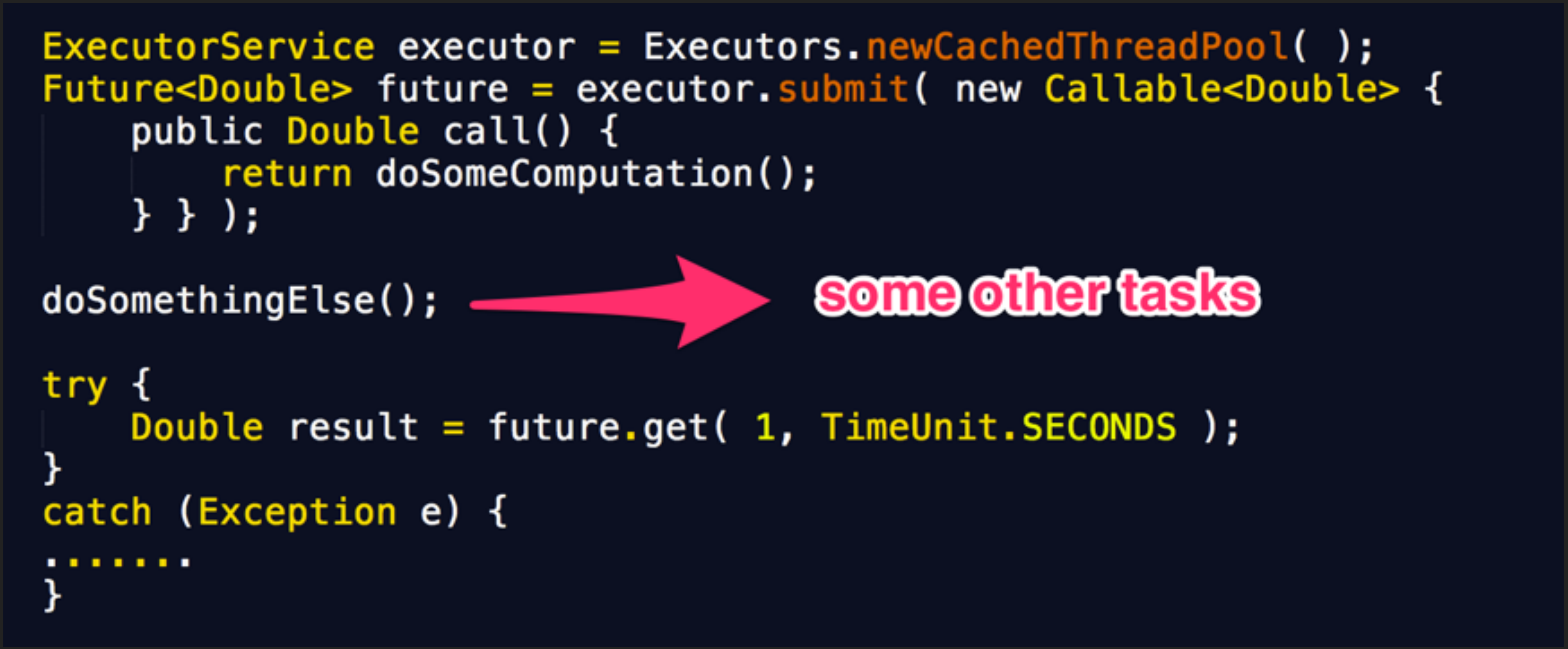
- Definition: A way to evolve Interface APIs in a compatible way.
- As a result, interface could now have methods with implementation
- This means "Java supports multiple inheritance"
- How does Java solves traditional "Diamond Problem"?
    - Three resolution rules
- http://www.javabrahman.com/java-8/java-8-multiple-inheritance-conflict-resolution-rules-and-diamond-problem/

OUTLINE

- What will Java 8 give us
- Behavior parameterization
    - More comprehensive functional interface
    - Lambda expression
    - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
    - Intuition
    - Intermediate and terminal operations
    - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Review: Future

- Future is used for asynchronous programming

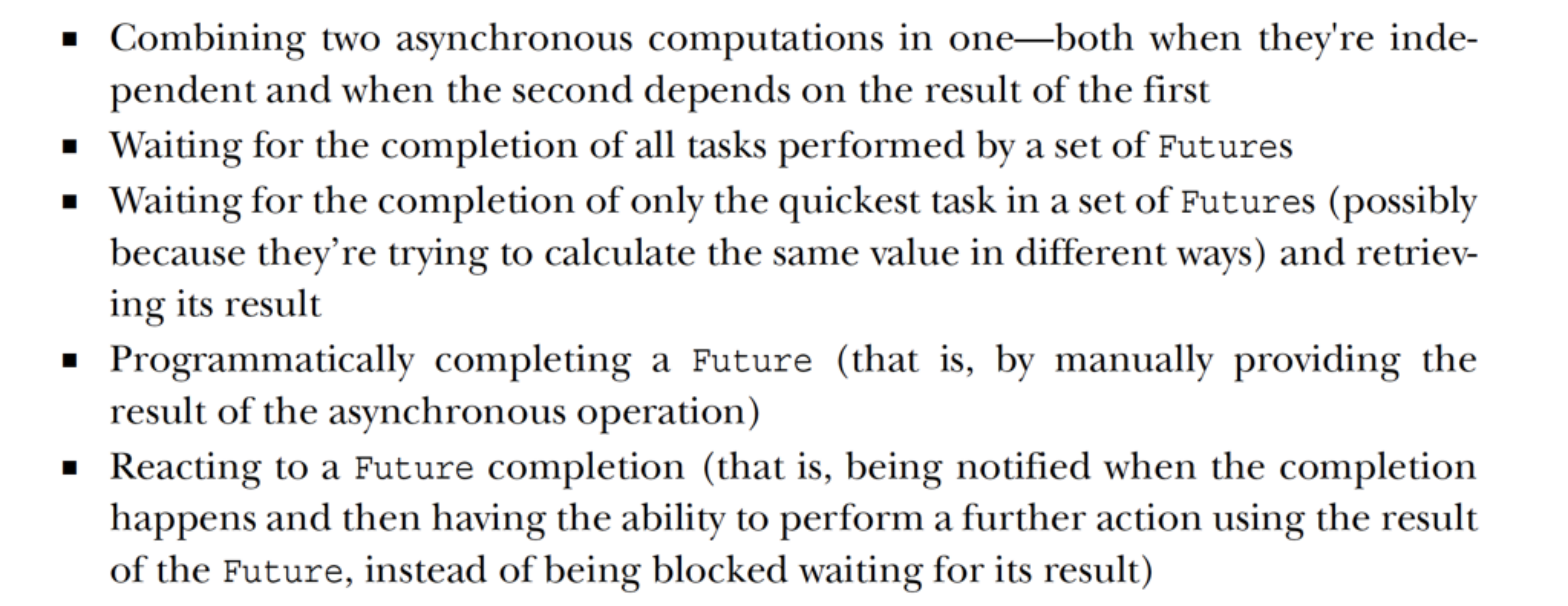


```java
ExecutorService executor = Executors.newCachedThreadPool( );
Future<Double> future = executor.submit( new Callable<Double> {
    public Double call() {
        return doSomeComputation();
    } } );

doSomethingElse();

try {
    Double result = future.get( 1, TimeUnit.SECONDS );
}
catch (Exception e) {
.......
}
```

some other tasks

How to combine multiple Future task???

CompletableFuture comes into play



- Combining two asynchronous computations in one—both when they're independent and when the second depends on the result of the first
- Waiting for the completion of all tasks performed by a set of `Futures`
- Waiting for the completion of only the quickest task in a set of `Futures` (possibly because they're trying to calculate the same value in different ways) and retrieving its result
- Programmatically completing a `Future` (that is, by manually providing the result of the asynchronous operation)
- Reacting to a `Future` completion (that is, being notified when the completion happens and then having the ability to perform a further action using the result of the `Future`, instead of being blocked waiting for its result)

Other new features

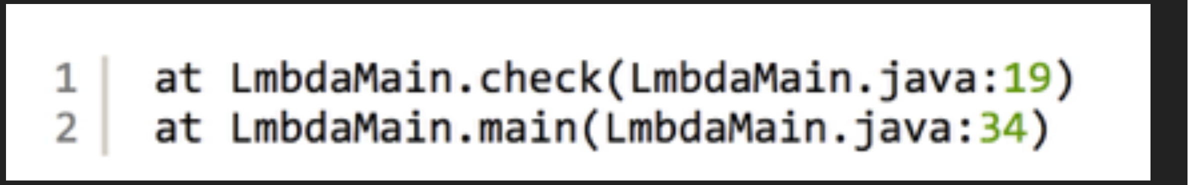
- Date and Time API – Handle time zone, separate concerns
- JVM Javascript engine – Nashorn

OUTLINE

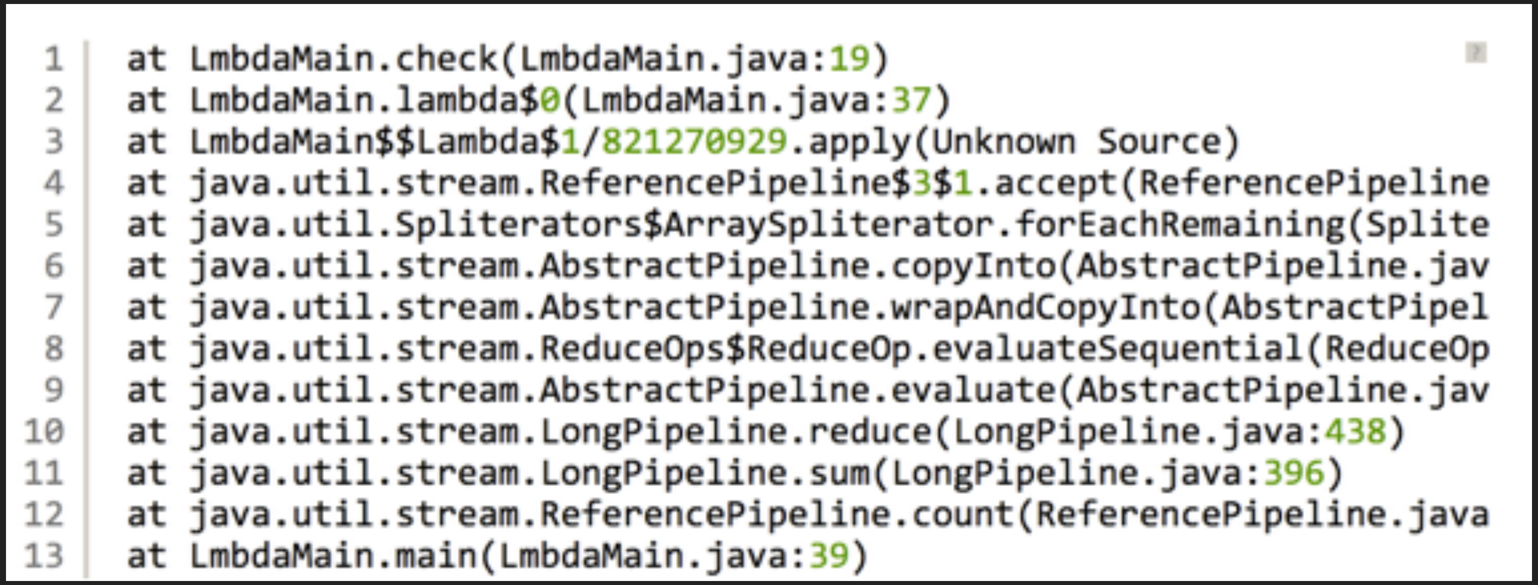
- What will Java 8 give us
- Behavior parameterization
  - More comprehensive functional interface
  - Lambda expression
  - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
  - Intuition
  - Intermediate and terminal operations
  - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

Dark side of Java 8

- Lambda expressions will make debugging log much longer



```
at LmbdaMain.check(LmbdaMain.java:19)
at LmbdaMain.main(LmbdaMain.java:34)
```



```
at LmbdaMain.check(LmbdaMain.java:19)
at LmbdaMain.lambda$0(LmbdaMain.java:37)
at LmbdaMain$$Lambda$1/821270929.apply(Unknown Source)
at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
at java.util.Spliterators$ArraySpliterator.forEachRemaining(Splite
at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.jav
at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipel
at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOp
at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.jav
at java.util.stream.LongPipeline.reduce(LongPipeline.java:438)
at java.util.stream.LongPipeline.sum(LongPipeline.java:396)
at java.util.stream.ReferencePipeline.count(ReferencePipeline.java
at LmbdaMain.main(LmbdaMain.java:39)
```

- Not truly functional
- Additional reading
- http://blog.takipi.com/6-reasons-not-to-switch-to-java-8-just-yet/
- Google search “DZone what’s wrong with Java 8”

OUTLINE

- What will Java 8 give us
- Behavior parameterization
  - More comprehensive functional interface
  - Lambda expression
  - Method reference
- Simple syntactic sugar - new methods inside Collections
- Functional programming - Stream API
  - Intuition
  - Intermediate and terminal operations
  - Properties
- Alternative to NULL – Optional
- Changeable Interface – Default methods
- Asynchronous programming enhancement - Future vs CompletableFuture
- Other features
- Dark side of Java 8
- Conclusion

CONCLUSION

- Anonymous class - lambdas
- Database-like routines - Stream API
- Get to know functional programming
- Null pointer exception - Optional
- Lots of syntactic sugar in collections method

THANK YOU

Shijie Zhang