НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ
КОРАБЛЕБУДУВАННЯ
ІМЕНІ АДМІРАЛА МАКАРОВА

Java 8,11,15,16,17 Lambda Expressions and Streams

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua

Life before Java 8

Extracting employee names

```
public List<String> empNames(List<Employee> employees) {
  List<String> e = new ArrayList<>();
  for (Employee emp : employees)
    e.add(emp.getName());
  return e;
}
```

Extracting employee ages

```
public List<Integer> empAges(List<Employee> employees) {
  List<Integer> e = new ArrayList<>();
  for (Employee emp : employees)
    e.add(emp.getAge());
  return e;
}
```

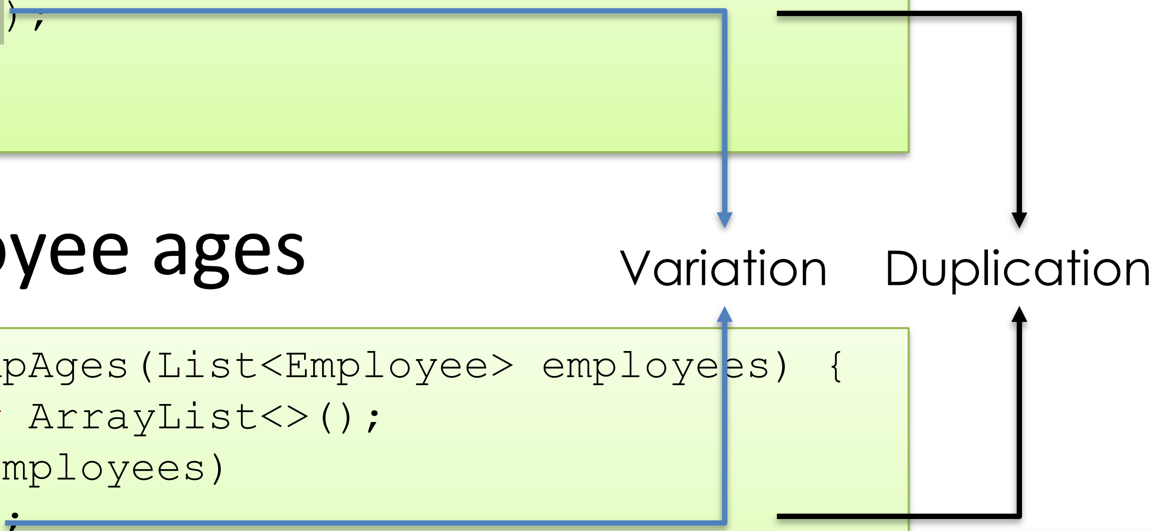
Life before Java 8

Extracting employee names

```java
public List<String> empNames(List<Employee> employees) {
  List<String> e = new ArrayList<>();
  for (Employee emp : employees)
    e.add(emp.getName());
  return e;
}
```

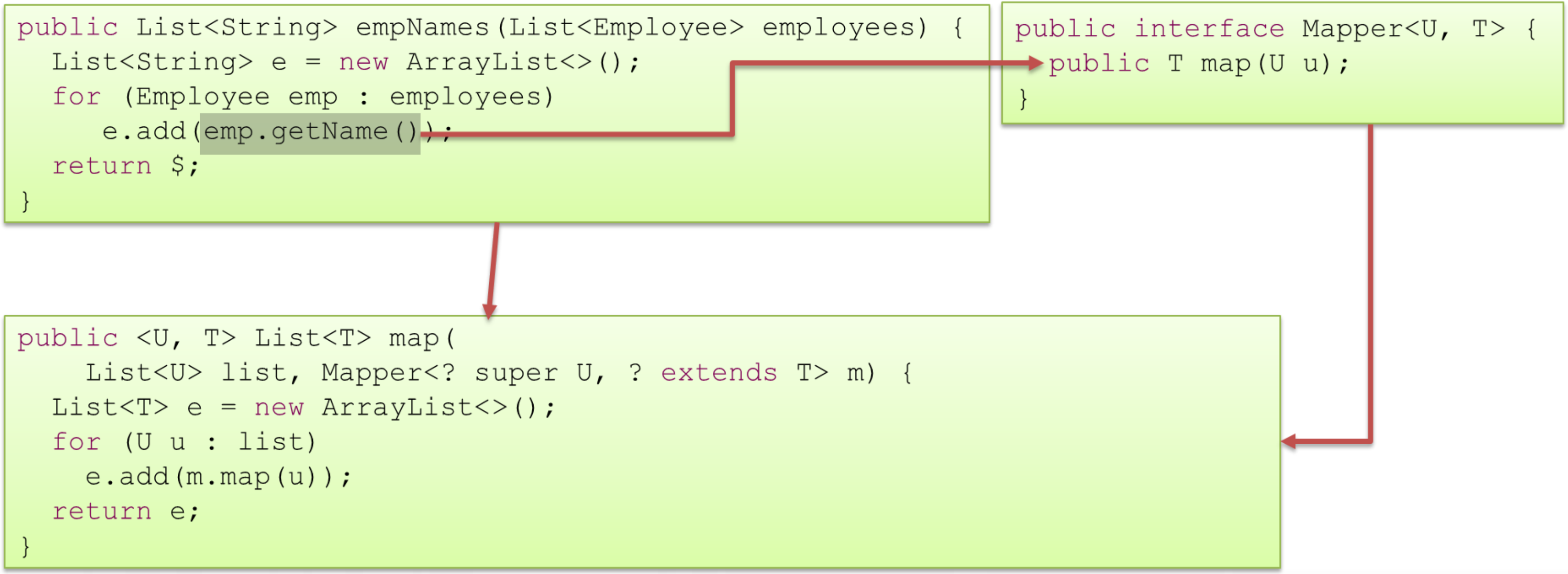
Extracting employee ages

```java
public List<Integer> empAges(List<Employee> employees) {
  List<Integer> e = new ArrayList<>();
  for (Employee emp : employees)
    e.add(emp.getAge());
  return e;
}
```



Life before Java 8 (cont.)

Lets identify the control structure, and extract the behavior into an object



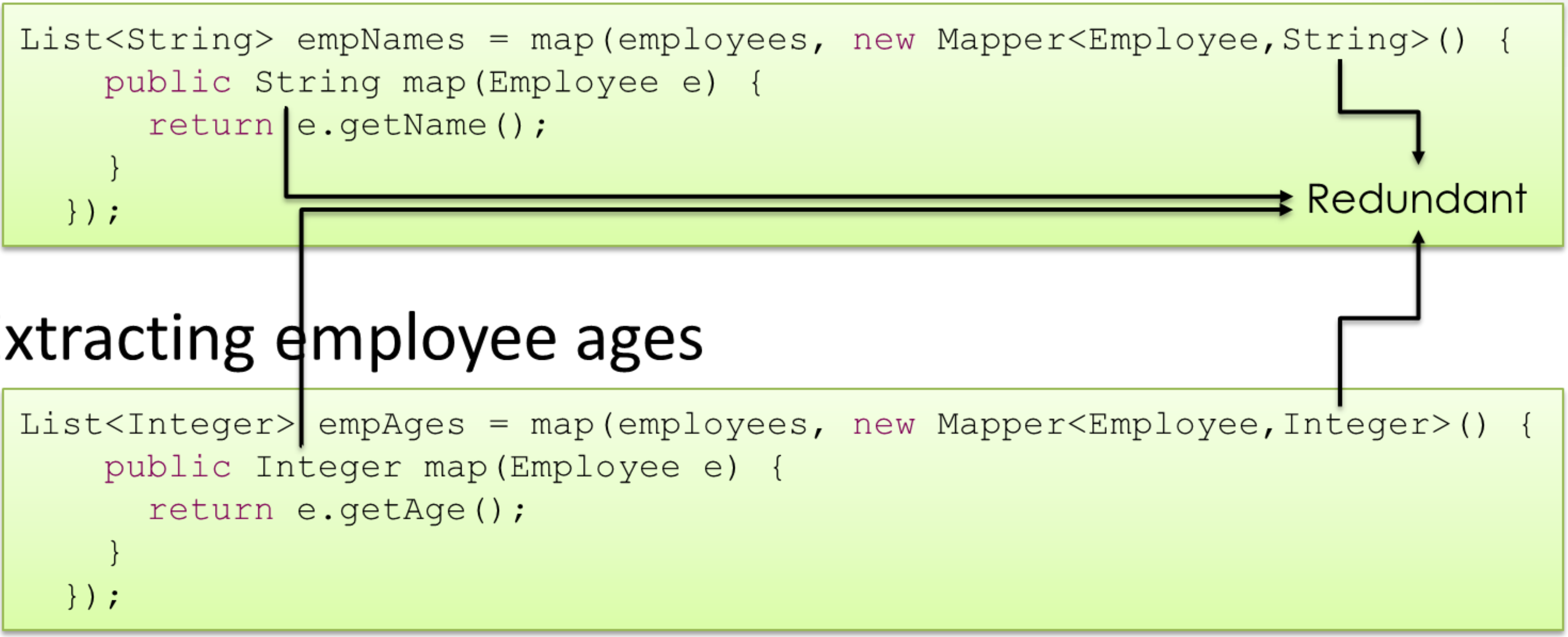
```java
public List<String> empNames(List<Employee> employees) {
  List<String> e = new ArrayList<>();
  for (Employee emp : employees)
     e.add(emp.getName());
  return $;
}
```

```java
public interface Mapper<U, T> {
  public T map(U u);
}
```

```java
public <U, T> List<T> map(
    List<U> list, Mapper<? super U, ? extends T> m) {
  List<T> e = new ArrayList<>();
  for (U u : list)
    e.add(m.map(u));
  return e;
}
```

Life before Java 8 (cont.)

Extracting employee names



```
List<String> empNames = map(employees, new Mapper<Employee,String>() {
    public String map(Employee e) {
      return e.getName();
    }
  });
```

Redundant

Extracting employee ages

```
List<Integer> empAges = map(employees, new Mapper<Employee,Integer>() {
    public Integer map(Employee e) {
      return e.getAge();
    }
  });
```

In the Kingdom of Nouns

We removed the code duplication, but this is still very verbose…

- Semantically, `map` is a **higher level function**
  - This means that it accepts a function as an **argument** (or returns a function)
- Syntactically, functions do not exist as first class entities
  - All verbs (functions) have be accompanied by a noun (class)
  - http://steve-yegge.blogspot.co.il/**2006**/03/execution-in-kingdom-of-nouns.html
  - translation: https://ru.hexlet.io/blog/posts/javaland
- Prior to Java 8, Java was the only programming language in popular use without anonymous functions / blocks / lambdas / function pointers
  - This is **not** purely a **syntactic** issue; Java also lacked proper support for such function in its **collections** and **standard libraries**
  - Some libraries, like Guava, attempted to fill the void

Enter Java 8 !

- Extracting employee names:

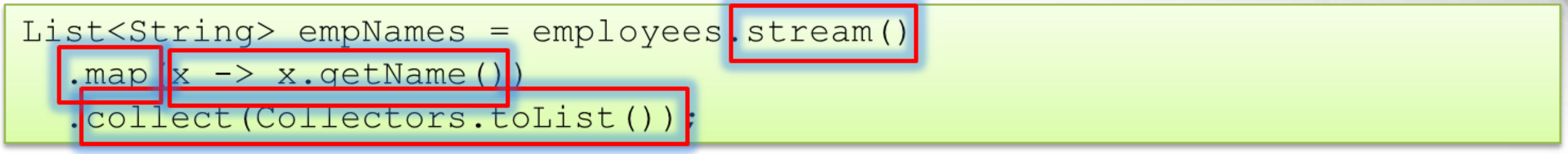
```
List<String> empNames = employees.stream()
   .map(x -> x.getName())
   .collect(Collectors.toList());
```

- Extracting employee ages:

```
List<Integer> empAge = employees.stream()
   .map(Employee::getAge) // method reference instead of lambda
   .collect(Collectors.toList());
```

- Still very verbose compared to other languages (C#, Scala, Python)
  - “boiler-plate” ratio lessens when we compose actions (see later)

Let’s take a deeper look…



```
List<String> empNames = employees.stream()
  .map(x -> x.getName())
   .collect(Collectors.toList());
```

- `stream()` is a **`default`** method of `List`
- `map` is a higher level function of `Stream`
- `x -> x.getName()` is a **lambda expression**
- `collect` turns the `Stream` back to a normal `Collection` (in our case, a `List`)
- Let’s go over each of these terms one by one

default Methods

```
List<String> empNames = employees.stream()
  .map(x -> x.getName())
  .collect(Collectors.toList());
```

- **default** methods are (default) implementations for **interfaces**
  - Can be **overridden** extending interfaces and implementing classes

```
interface Foo {
  void a(); // regular abstract method
  default void b() { // can also be overriden
    System.out.println("I'm a default method!");
  }
}
```

- Adds **new** functionality to an existing interface without **breaking** all client code
  - In our case, we added the **stream()** method to Collection

Comparison to other languages / features

- So is this the same as **multiple inheritance**?
  - Nope; more similar to **Traits**
  - There is neither **conflict resolution** nor **constructors**, so the model is much **simpler**
- So are these **extension methods** (a la C#)?
  - No, because extension methods are actually **syntactic sugar** for **static decorators**
  - *You* can't add methods to library classes (e.g., in C# you can add extension methods to `String`).
- Solutions in other languages
  - Ruby – mixins
  - Python/Javascript – monkey patching
  - Scala – implicits / pimp my library
  - Haskell – type classes

Higher order functions

```
List<String> empNames = employees.stream()
  .map(x -> x.getName())
  .collect(Collectors.toList());
```

- `map` is a higher order function in `stream`
  - A function that takes a function
- Other higher order functions in `Stream`
  - `filter`, `map`, `flatMap`, `sorted`, `reduce`, …
- Similar libraries in other languages
  - `LINQ` in C#, `itertools` in Python, `Enumerable` in Ruby, etc.

Streams

- `Stream` is the **gateway** to the "functional collections" in Java 8
    - ➨ Provide a **uniform API** (why is this important?)
- We only iterate over a stream once, even if we have two or more higher level functions
- This is because streams are **lazily evaluated**
    - – Until we **`collect`** (or form some other **reduction**), no iteration takes place
    - ➨ **`collect`** is a form of **mutable reduction**
        - ➨ i.e., it reduces to a mutable container
        - ➨ Other reductions include `forEach` and, well, `reduce`
- Streams also give us “free” **parallelization** (why is it so easy?)

```
List<String> empNames = employees.stream()
   .parallel()
   .map(x -> x.getName())
   .collect(Collectors.toList());
```

`Streams`: Caveats

- Streams are “single serving” only!
  - This code will throw an exception:

```
Stream<Student> stream = students.stream();
Stream<String> names = stream.map(Student::getName);
Stream<Integer> ages = stream.map(Student::getAge);
```

  - This too:

```
Stream<String> names = students.stream.map(Student::getName);
stream.forEach(this::printStudent);
stream.forEach(this::addStudentToDatabase);
```

- Avoid returning `Stream` from a **`public`** function, or keeping one as a field,
  - An `Iterable` or `Collection` is usually more suitable
  - Although there are some (rare) cases where it’s appropriate, there are usually better (monadic) types

Lambdas and SAMs

```
List<String> empNames = employees.stream()
  .map(x -> x.getName())
  .collect(Collectors.toList());
```

- The signature for `map` is:
  `map(Function<? super T,? extends R> mapper)`
- And here is the signature for `Function` (**`default`** methods retracted):

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

- An **interface** which has **single `abstract`** (i.e., non-**`default`**) method (often abbreviated **SAM**) can be called a **functional interface**
- **Lambdas** are just **syntactic sugar** for implementing functional interfaces
  - Method reference (`::`) and lambdas are interchangeable, where applicable
  - References are considered “more elegant” (as we will see later)
- So is Java a **functional** language now?
  - Functions aren’t first-class citizens; functions aren’t even a proper part of the Java language, just a standard library **interface**
  - Although an alternative interpretation could argue that interfaces are the new functions

Lambdas (cont.)

This design choice has a great pro: we can also use lambda with legacy API!

- Old code

```
new Thread(new Runnable() {
  @Override
  public void run() {
    System.out.println("Kill me :[");
  }
}).start();
```

- New code

```
new Thread(() -> System.out.println("PARTEH! :D|-< :D/-< :D\-<)).start();
```

- We can use the convenience `@FunctionalInterface` annotation to tell the compiler that the interface should be functional (a la `@Override`)

```
@FunctionalInterface
interface Foo { void bar(); void bazz(); } // won’t compile
```

More API examples

- Assure we are not hiring anyone underage

```
assert employees.stream().noneMatch(x -> x.age < 18);
```

- Find the highest paid individual in the company

```
Optional<Employee> opt = employees.stream().maxBy(
                                   (x, y) -> x.salary - y.salary);
```

- What is returned if the list is **empty**?
- Instead of working with **`null`**, a new type **`Optional<T>`** is returned
  - **`Optional<T>`** can be present (i.e. not **`null`**) or `empty` (i.e. **`null`**)
  - Has a method **`get()`** that returns **`T`** or throws an exception

More API examples

What is this?

- Assure we are not hiring anyone underage

```
assert employees.stream().noneMatch(x -> x.age < 18);
```

- Find the highest paid individual in the company

```
Optional<Employee> opt = employees.stream().maxBy(
                                   (x, y) -> x.salary – y.salary);
```

- What is returned if the list is **empty**?
- Instead of working with **`null`**, a new type **`Optional<T>`** is returned
  - **`Optional<T>`** can be present (i.e. not **`null`**) or `empty` (i.e. **`null`**)
  - Has a method **`get()`** that returns **`T`** or throws an exception

Wait, what’s wrong with `null`s?

- The billion dollar mistake
- **`null`**s are incredibly dangerous!
  - Often **unchecked** until **used**
    - a “sleeper agent” that destroys the application, its origin is hard to trace
  - By returning an `Optional`, we are **explicit** in our result type
    - **Types are better than comments!**
- `Optional` also has **higher order functions**

```
Optional<Employee> richest = …
Optional<Integer> ageOfRichest =
richest.map(Employee::getAge);
```

- `filter` will return `empty` if the predicate returns **`false`**

```
richestEmployee.filter(x -> x.age >= 18);
```

Composing Optionals

- Optionals **compose** using **flatMap**

```
// working with nulls
Student s = getStudent();
if (s == null)
  return null;
Course c = s.getCourse("Software Design");
if (c == null)
  return null;
Exam e = c.getMoedA();
if (e == null)
  return null;
return e.getGrade();
```

```
// but if we returned Optionals…
getStudent()
  .flatMap(Student::getCourse)
  .flatMap(Course::getMoedA)
  .flatMap(Exam::getGrade)
```

A more complex example

- Get Ukrainian students with a top grade sorted by name in Java 7

```
List<Student> topGrades = new ArrayList<>();
Collections.sort(students, new Comparator<Student>() {
  public int compare(Student student1, Student student2) {
    return student1.getName().compareTo(student2.getName());
  }
});
for (Student student: students)
  if ("Ukraine".equals(student.getCountry()))
    if (student.getGrade() >= 90)
      topGrades.add(student);
```

A more complex example

- Get Ukrainian students with a top grade sorted by name in Java 7

Sorts in place! Why is this bad?

```
List<Student> topGrades = new ArrayList<>();
Collections.sort(students, new Comparator<Student>() {
  public int compare(Student student1, Student student2) {
    return student1.getName().compareTo(student2.getName());
  }
});
for (Student student: students)
  if ("Ukraine".equals(student.getCountry()))
    if (student.getGrade() >= 90)
      topGrades.add(student);
```

A more complex example

- Get Ukrainian students with a top grade sorted by name in Java 7

Sorts in place! Why is this bad?

```
List<Student> topGrades = new ArrayList<>();
Collections.sort(students, new Comparator<Student>() {
  public int compare(Student student1, Student student2) {
    return student1.getName().compareTo(student2.getName());
  }
});
for (Student student: students)
  if ("Ukraine".equals(student.getCountry()))
    if (student.getGrade() >= 90)
      topGrades.add(student);
```

Depth of 3!

A more complex example

- Get Ukrainian students with a top grade sorted by name in Java 7

Sorts in place! Why is this bad?

```
List<Student> topGrades = new ArrayList<>();
Collections.sort(students, new Comparator<Student>() {
  public int compare(Student student1, Student student2) {
    return student1.getName().compareTo(student2.getName());
  }
});
for (Student student: students)
  if ("Ukraine".equals(student.getCountry()))
    if (student.getGrade() >= 90)
      topGrades.add(student);
```

Depth of 3!

- In Java 8 and later:

```
List<Students> topStudents = students.stream()
  .filter(x -> "Ukraine".equals(x.getCountry()))
  .filter(x -> x.getGrade() >= 90)
  .sorted(Comparator.comparing(Student::getName))
  .collect(Collectors.toList());
```

Other cool tricks

- Sum of all salaries in the company with "map-reduce“

```
employees.stream()
   .mapToInt(Employee::getSalary)// note the mapToInt... why?
   .reduce(0, Integer::sum)
// could also be done with Lambdas, or simply .sum()
```

- Count the number of employees **by** rank

```
Map<Rank, Long> countByRank =
employees.stream().collectors(
       Collectors.groupingBy(Employee::getRank,
                                     Collectors.counting());
```

- Streams **compose** using **`flatMap`** too!

```
List<Student> allIsraeliStudents = universities.stream()
   .flatMap(u -> u.getFaculties().stream())
   .flatMap(f -> f.getStudents().stream())
   .collect(Collectors.toList());
```

Declarative versus Imperative programming

`Streams` and `Optionals` are an example of moving from **imperative** code to **declarative** code

- In imperative code we write the **exact, low level** steps:
  - **Create** a new list object
  - **Iterate** over the original list
    - For every entry, **apply some function `f`** on it
    - **Add** the result of **`f`** in the new list
  - **Return** the new list
- In declarative programming, we write a **higher level description:**
  - **map** all elements in the list using some function `f`
  - **collect** to a `List`

Declarative versus Imperative (Cont.)

Declarative code is **shorter**, more **precise** and **explicit**, more **readable**, and less **error-prone**

- You can do pretty **anything** inside a **`for`** loop
- That means you have to **read** the **entire body** to know what’s going on
- More room for **bugs**

Declarative code is written in a higher level of abstraction

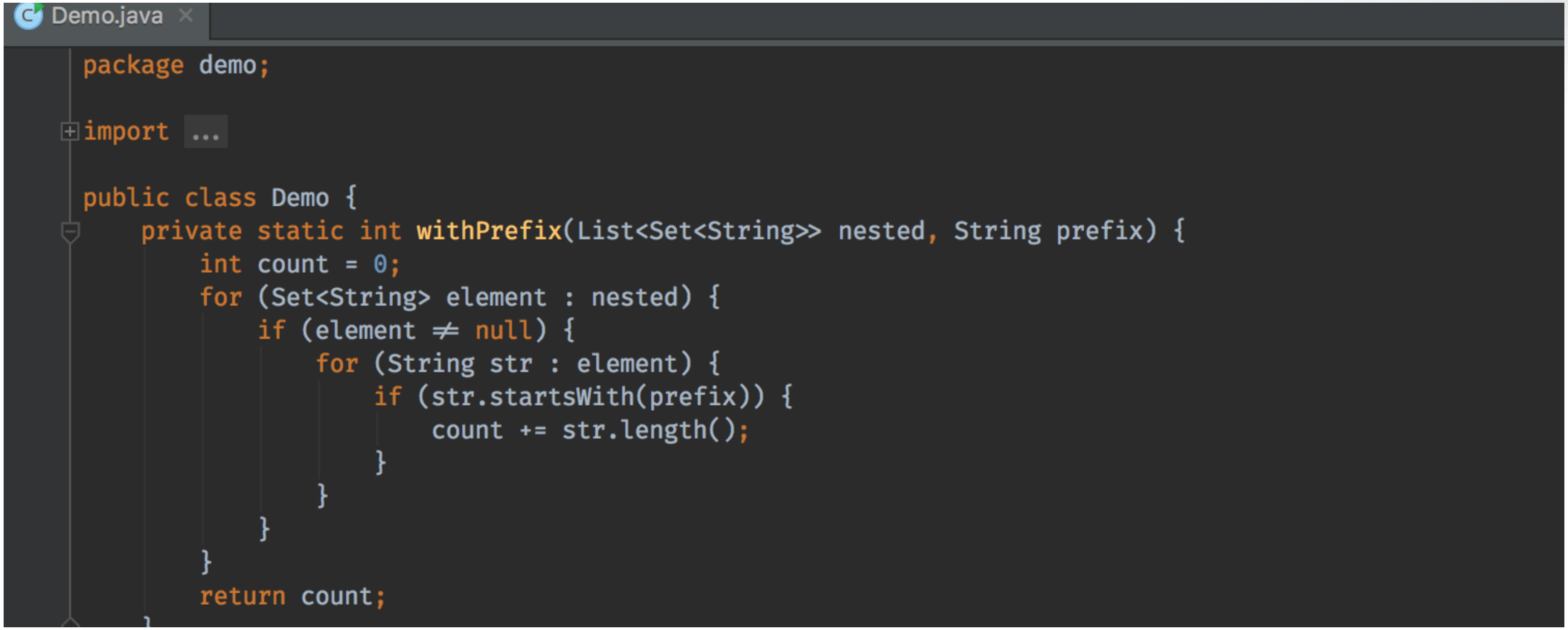
- In our case, **`maps`** and **`filters`**, rather than object **creation** and **modification**
- **Higher order functions** instead of **control structures** and **primitive checks**
- Less **moving parts**, hide the **unnecessary details**

Dec. v Imp. – A **spectrum**, not **dichotomy**

- Before Java 5, we had to iterate by **index**, or use the **iterator** directly
  - Even more bugs: infinite loop, index modifications
- Using `list.add` is more declarative than managing the internal data structure on your own
  - Using a library/function is usually more declarative than inlining its code
- Applies to **syntax**, not just **semantics**
  - An array initializer (`new int[] {1, 2, 3}`) is more declarative than doing it manually
  - A **lambda expression** is more declarative than an **anonymous functions**, but a **method reference** is more declarative than a lambda expression
  - Rule of thumb: Less **tokens** ⇒ More declarative

- Avoid loops, use `Streams`
  - Almost any loop can be replaced with a Stream call
  - The new version of IntelliJ does this automagically
- Avoid **`null`**s, use `Optionals`
  - `Optional`s are clearer, safer, compose better, and support higher level functions
  - Only use **`null`**s when dealing with **legacy** APIs
- Prefer **declarative** to **imperative** code whenever possible

Demo



```java
package demo;

import ...

public class Demo {
    private static int withPrefix(List<Set<String>> nested, String prefix) {
        int count = 0;
        for (Set<String> element : nested) {
            if (element ≠ null) {
                for (String str : element) {
                    if (str.startsWith(prefix)) {
                        count += str.length();
                    }
                }
            }
        }
        return count;
    }
```

Java 8,11,15,16 Lambda Expressions and Streams

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua