Programming Fundamentals 2

Pierre Talbot

11 May 2021

University of Luxembourg

Chapter XI. Modern Java Concepts

A Tour of Modern Java Concepts

Programming languages are not static entities, but evolve over the years by incorporating new concepts:

- Java 8: lambda expressions, unsigned integer arithmetic, ...
- Java 9: modules, ...
- Java 10: local-variable type inference:

```
var list = new ArrayList<String>();
// equivalent to
ArrayList<String> list = new ArrayList<String>();
```

- Java 14: switch expressions

```
static void howMany(int k) {
    System.out.println(
        switch (k) {
            case  1 -> "one";
            case  2 -> "two";
            default -> "many";
        }
    );
```

- Java 15: text block, new garbage collectors, ...
- Java 16: pattern matching for `instanceof`, records:

```
record PokemonCard(String name, String description, int hp, int level) {}

PokemonCard c = new PokemonCard("Pikachu", "Great Pokemon", 100, 1);
System.out.println(c.name()); // accessors automatically generated.
PokemonCard c2 = ...;
if(c.equals(c2)) { ... }  // equals automatically generated.
```

The constructors, accessors, methods `equals`, `hashCode`, `toString`, ... are automatically generated.

Limitation

Records should solely be used when we need to store "data" and no interesting treatment is performed on those. They are **immutable** and cannot inherit from classes or implement interfaces.

A Tour of Modern Java Concepts

We explain in more depth two advanced concepts of Java:

- Nested classes
- Lambda expressions

Nested classes

Nested classes

For now, we have only used classes declared at "top-level".
A class can also be declared inside other Java entities:

- *Inner class*: a class within a class with access to the containing class's fields and methods.
- *Static nested class*: a class within a class *without* access to the containing class's *non-static* fields and methods.
- *Local class*: a class declared within a method.
- *Anonymous class*: a class without a name.

See also *Effective Java. Item 24: Favor static member classes over nonstatic*.

Inner class

```
public class Arena {
  private ArrayList<Team> teams;
  private Battlefield battlefield;

  private class ApplyAction implements ActionVisitor {
    ArrayList<Integer> championsWhoActed;

    public void visitSpawn(int teamID, int championID, int x, int y) {
      teams.get(teamID).spawnChampion(championID, x, y));
    }
    ...
  }
}
```

- `ApplyAction` has access to all private fields of `Arena`.
- `ApplyAction` can be used like a normal class inside `Arena`.
- As `ApplyAction` is `private`, it is not visible outside of `Arena`.
- `Arena` cannot access the private members of `ApplyAction`.

Inner class

How does it work?

An instance of an inner class has a *hidden field* containing the reference of an instance of the containing class.

```
public class Arena {
  private class ApplyAction implements ActionVisitor {
    Arena arena;
    public ApplyAction(Arena arena) {
      this.arena = arena;
    }
    public void visitSpawn(int teamID, int championID, int x, int y) {
      arena.teams.get(teamID).spawnChampion(championID, x, y));
    }
    ...
  }
```

Therefore, you can only instantiate an inner class in the context of its containing class.

Static nested class

```
public class Battlefield {
  public static enum GroundTile {
    GRASS,
    ROCK;
    // ...
  }
}
```

- Very similar to a class declared "normally", but in addition can access private static members of the containing class.
- It is a way to indicate a class is very dependent w.r.t. another one.
- Here `GroundTile` existence depends on `Battlefield`.
- However, it is often better to use package to group classes together, so static nested classes have a limited usage.

Local class

Perhaps surprisingly, you can declare a class almost anywhere a local variable can be declared:

```
public class NumberValidator {
  public static void validatePhoneNumber(String phoneNumber) {
    class PhoneNumber {
        String formattedPhoneNumber;
        PhoneNumber(String phoneNumber){...}
        public String getNumber() {
            return formattedPhoneNumber;
        }
    }
    PhoneNumber myNumber1 = new PhoneNumber(phoneNumber);
    if (myNumber1.getNumber() == null) { ... }
    ..
  }
}
```

This kind of nested class is not very useful and used.

Anonymous class

The last kind of nested class, anonymous class, is very common.

```
public class Arena {
  // ...
  @Override public String toString() {
    StringBuilder map = new StringBuilder();
    visitFullMap(new TileVisitor() {
      @Override public void visitGround(GroundTile groundTile, int x, int y) {
        map.append(GroundTile.stringOf(groundTile));
        newline(x);
      }

      @Override public void visitChampion(Champion c) {
        map.append('C');
        newline(c.x());
      }
      // ...
    });
    return map.toString();
  }
```

Anonymous class

- It is similar to a local class, but without a name.
- Can access and modify local objects (*e.g.*, `map` in the previous example).
- Useful when a class is local to a method, only need to be instantiated in one place, and is not reusable outside of the current context.

Limitations

- Cannot implement multiple interfaces.
- Cannot be used inside expression relying on type name such as `instanceof`.
- Clients of anonymous class can only call the methods of the super types (which can be overridden by the anonymous class).

Nested classes: rules of thumb

- Should be kept short.
- Use static nested classes if you don't need a reference to an object of the containing class.
- Use lambda expressions instead of anonymous classes when it is possible.

Lambda Expressions

Lambda expressions

Lambda expressions are inspired by the *functional programming* paradigm (that you will learn in PF3).

Functional programming

- **Immutable memory.**
- **First-order functions**: we can pass functions to functions as arguments, and functions can be returned too!

Lambda expressions are first-order functions over mutable memory.

The following slides are based on *Effective Java. Chapter 7: Lambdas and Streams* and `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`.

Motivation

We take the example of Pokedeck (lab 2), where we have a collection of cards that we must search. We can search a card by ID, by name, by card type:

```
public class Deck {
  private ArrayList<Card> deck;
  // ...
  public ArrayList<Card> getCardsByName(String name) {
    ArrayList<Card> searchResults = new ArrayList<Card>();
    for (Card card : deck) {
      if(name.equals(card.getName())) {
        searchResults.add(card);
      }
    }
    return searchResults;
  }
  public ArrayList<Card> getCardsById(String id) {
    ArrayList<Card> searchResults = new ArrayList<Card>();
    for (Card card : deck) {
      if(id.equals(card.getId())) {
        searchResults.add(card);
      }
    }
    return searchResults;
  }
  public ArrayList<Card> getCardsByCardType(CardType cardType) {...}
```

Motivation

- We can observe a recurring pattern.
- All these methods look the same: don't repeat yourself (DRY) principle!

Two solutions

1. The old one with interfaces and anonymous classes.
2. The new and cool one with lambda expressions.

Solution 1: Interface and anonymous class

First, we create a `DeckFilter` interface:

```
interface DeckFilter {
  boolean keepCard(Card c);
}
```

We can then code a generic search method using this filter:

```
public ArrayList<Card> search(DeckFilter filter) {
  ArrayList<Card> searchResults = new ArrayList<Card>();
  for (Card card : deck) {
    if(filter.keepCard(card)) {
      searchResults.add(card);
    }
  }
  return searchResults;
}
```

Solution 1: Interface and anonymous class

The user of the class `Deck` can choose its own search criterion:

```
Deck deck = ...;
deck.search(new DeckFilter() {
  @Override boolean keepCard(Card card) {
    return name.equals(card.getName());
  }
})
```

As a bonus, we also made the class more respectful of the open-closed principle!

Solution 1: Interface and anonymous class

Note that we can provide a number of default filters:

```
public class DeckSearch {
  public static class FilterByName implements DeckFilter {
    private String name;
    public FilterByName(String name) {
      this.name = name;
    }
    @Override boolean keepCard(Card card) {
      return name.equals(card.getName());
    }
  }
  public static class FilterByID implements DeckFilter { ... }
}
```

That can be used in client code:

```
Deck deck = ...;
deck.search(new DeckSearch.FilterByName(name));
```

Drawbacks of solution 1

- It introduces a lot of "boilerplate code": interfaces, static nested classes.
- And it is not always very readable: anonymous classes.

Lambda expressions provide a more satisfying way to solve this problem.

Solution 2: using lambda expressions

The interface `DeckFilter` is what is called a **functional interface** because it has only one abstract method:

```
@FunctionalInterface interface DeckFilter {
  boolean keepCard(Card c);
}
```

As a syntactic shortcut, we can use *lambda expressions* to implement this interface:

```
Deck deck = ...;
deck.search(card -> name.equals(card.getName()));
```

The code `card -> name.equals(card.getName())` is a lambda expression.

Lambda expressions

Another example: sorting a list of strings by length:

```
Collections.sort(words,
  (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

- We declare a function taking two arguments `s1` and `s2`.
- Implicit `return` keyword.
- All types are automatically inferred by the compiler.

Syntax of lambda expressions

```
(arg1, arg2, ...) -> A Java expression
(arg1, arg2, ...) -> {
  One or more statements (possibly ending with a return statement)
}
```

Improving solution 2 using streams

Streams encapsulate a number of generic and common operations on list: `filter`, `map`, `reduce`, `sorted`, ...

Lambda expressions really shine in cooperation with streams:

```
public List<Card> search(DeckFilter filter) {
  return deck.stream()
    .filter(filter)
    .collect(Collectors.toList());
}
```

We create a stream, filter on this stream and then collect the result.

Compare with the previous solution: it is much shorter.

Improving (again) solution 2 using standard functional interface

The interface `DeckFilter` contains a single filtering method which is actually quite common. Java defines a number of standard interface so we do not need to redefine ours:

```
public List<Card> search(Predicate<Card> keepCard) {
  return deck.stream()
    .filter(keepCard)
    .collect(Collectors.toList());
}
```

`Predicate<Card>` contains a generic method `boolean test(T t)`. The call to `search` stays the same, as the lambda expression automatically implements `Predicate<Card>`:

```
deck.search(card -> name.equals(card.getName()));
```

Standard functional interface

Functional interfaces already present in `java.util.function`:

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

Method references

A (static) method can also be used where a lambda expression is expected:

Method Ref Type	Example	Lambda Equivalent
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then = Instant.now();` `t -> then.isAfter(t)`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>`
Array Constructor	`int[]::new`	`len -> new int[len]`

Summary

Method references often provide a more succinct alternative to lambdas. Where method references are shorter and clearer, use them; where they aren't, stick with lambdas.

A more complete example using lambda and streams

Here a function retrieving the sorted list of ID of all cards fulfilling a criterion:

```
public List<Integer> filteredSortedID(Predicate<Card> keepCard) {
  return deck.stream()
    .filter(keepCard)
    .map(Card::getID)
    .sorted()
    .collect(Collectors.toList());
}
```

- *Fluent interface*: we can chain the operations.
- *Lazily evaluated*: the whole thing is only evaluated when we arrive on `collect`. It means the collection is not traversed more than once in case of multiple `map`/`filter` operations!