Software Engineering

Functional Programming

Prof. Agostino Poggi

Imperative and Functional Programming

- **Imperative programming** is a programming paradigm that **uses statements** to **change** a **program state**
- **Functional programming** is a programming paradigm that **treats computation** as the **evaluation** of **mathematical functions**
- A **statement** is executed to **assign variables** whereas a **function** is evaluated to **produce** a **value**
- The **imperative programming** main focus is "**how to solve"** whereas the **functional programming** main focus is on "**what to solve**"

Pure and Higher-order Functions

- **Pure functions** return the **same result** if given the **same arguments** (i.e., they are deterministic) and they do **not cause** any **side effects** (i.e., they treat data as immutable)
  - Are **easy** to **debug** ant **test** because they have **no side effect** and because they do **not need** to **mock** anything
  - Make **easier** to write **parallel/concurrent** applications because they work on **immutable data**
- **Higher-order functions** take one or more **functions** as **arguments** and/or return a **function** as **result**

Lambda Calculus

- Was developed by Alonzo Church to study **computations** with **functions** and gives the **definition** of what is **computable**
- Provides a **theoretical framework** for **describing functions** and their **evaluation** that is based on purely **syntactic symbol manipulation** and **recursion**
- Everything is a (pure) function (lambda expression)

expr → λ var . expr | expr expr | var | (expr)

- **Anything** that can be **computed** by **lambda calculus** is computable and offer the **same definitional power** of a **Turing machine**
- Forms the **basis** of almost **all** current **functional** programming **languages**

Beta Reduction

$(\lambda x.\ +\ x\ 1)\ 4 \rightarrow (+\ 4\ 1) \rightarrow 5$

$(\lambda x.\ +\ x\ x)\ 4 \rightarrow (+\ 4\ 4) \rightarrow 8$

$(\lambda x.\ 3)\ 4 \rightarrow 3$

$(\lambda x.\ \lambda y.\ x + y)\ 3\ 4 \rightarrow \lambda y.(3 + y)\ 4 \rightarrow (3 + 4) \rightarrow 7$

$(\lambda x.\ x\ x)\ (\lambda y.\ y) \rightarrow (\lambda y.\ y)\ (\lambda y.\ y)$

Functional Programming Difficulties

- Writing pure functions may **reduce** the **readability** of code
- Writing programs in **recursive style** instead of **using loops** may be **not easy** for some **programmers**
- Writing **pure functions** are **easy** but **combining** them with **other software** may be **difficult**
- **Immutable values** and **recursion** can **reduce performances**

Lambda Expressions (1/3)

- Are **anonymous method** that can **implement** a **functional interface**

- Their **type** is the **type** of the **functional interface** that they implement

- Can be **used** anywhere **functional interfaces** are expected

- Can only **access** to the **fields** of the **enclosing class** and to **the local variables** and **parameters** of the **enclosing block** that are **final** or **effectively final**

Lambda Expressions (2/3)

- A lambda consists of a **parameter list** followed by the **arrow** token and a **body**, as in

  (parameterList) -> {statements}

- For example, the following lambda receives two integers and returns their sum

  (int x, int y) -> {return x + y}

- The **parameters** and **return types** may be **omitted** when they are **determined** by the **lambda context**

  (x, y) -> {return x + y}

Lambda Expressions (3/3)

- A lambda with a **one-expression body** can be written as

    (x, y) -> x + y

- In this case, the **expression value** is **implicitly returned**
- When the **parameter list** contains only **one parameter**, the **parentheses** may be **omitted**, as in

    x-> x * x

- A lambda with an **empty parameter list** is defined with () to the left of the arrow token (->), as in

    () -> System.out.println(

    “Lambda without parameters!")

MapDemo Class

```java
public final class MapDemo
{
  private MapDemo()
  {
  }

  private static <K, V> void printMap(fi
  {
    for (Entry<K, V> e : l.entrySet())
    {
      System.out
          .println(n + " element key " +
    }

    System.out.println();
  }

  public static void main(final String[]
  {

  }
}
```

```java
Map<String, Integer> immutable = Map.of("John", 1, "Steve", 2, "Jack", 3);

printMap("immutable", immutable);

Map<String, Integer> keyVals = new HashMap<>(immutable);

keyVals.put("Mary", 4);
keyVals.remove("Bob");

printMap("KeyVals", keyVals);

keyVals = new TreeMap<>();

keyVals.put("John", 1);
keyVals.put("Mary", 2);
keyVals.put("bob", 3);

printMap("keyVals", keyVals);

Comparator<String> myComparator = (x, y) -> {
  if (x.equals(y))
  {
    return 0;
  }
  else if (x.compareTo(y) > 0)
  {
    return -1;
  }

  return 1;
};

keyVals = new TreeMap<>(myComparator);

keyVals.put("John", 1);
keyVals.put("Mary", 2);
keyVals.put("bob", 3);

printMap("keyVals", keyVals);
```

Advantages of Lambda Expressions

- Support **functional programming**
- Allow to write **leaner** and **more compact code**
- Help in writing **parallel programs**
- Provides **more generic**, **flexible** and **reusable APIs**
- Support the **passing** of **behaviors** to **methods**

Method References

- Often a **lambda expression** simply **calls** an **existing method**

- In those cases, should often **clearer** to **refer** to the **existing method** by **name**

- **Method references** enable it through a **compact** and **easy form** of **lambda expressions**

- Can be **used** for **static** and **instance methods** and for **constructors**

Method Reference Types

- Can refer a **static method** with the form

  ContainingClass::staticMethodName

- Can refer an **instance method** with the form

  containingObject::instanceMethodName

- Can refer a **constructor** with the form

  ClassName::new

StaticMethodReferenceDemo Class

```java
public final class StaticMethodReferenceDemo
{
  private StaticMethodReferenceDemo()
  {
  }

  private static boolean numCheck(
      final IntPredicate p, final int n)
  {
    return p.check(n);
  }

  public static void main(final String[] args)
  {
    final int max = 50;

    Random r = new Random();

    int num = r.nextInt(max) - max / 2;

    IntPredicate intPredicate = number -> (number % 2) == 0;

    System.out.println("Lambda expression: " + num + " is even: "
        + numCheck(intPredicate, num));

    System.out.println("Static method reference: " + num + " is even: "
        + numCheck(IntPredicatesChecker::isEven, num));

    intPredicate = number -> number > 0;

    System.out.println("Lambda expression: " + num + " is positive: "
        + numCheck(intPredicate, num));

    System.out.println("Static method reference: " + num + " is positive: "
        + numCheck(IntPredicatesChecker::isPositive, num));
  }
}
```

```java
public final class IntPredicatesChecker
{
  private IntPredicatesChecker()
  {
  }

  public static boolean isPositive(final int n)
  {
    return n > 0;
  }

  public static boolean isEven(final int n)
  {
    return (n % 2) == 0;
  }
}
```

```java
@FunctionalInterface
public interface IntPredicate
{
  boolean check(int i);
}
```

InstanceMethodReferenceDemo Class

```java
public class InstanceMethodReferenceDemo
{
  private static final int MAX = 50;

  private int    num;
  private Random random;

  public InstanceMethodReferenceDemo()
  {
    this.random = new Random();

    this.num = this.random.nextInt(MAX);
  }

  public int getNum()
  {
    return this.num;
  }

  boolean isBigger(final int n)
  {
    return this.num > n;
  }

  public static void main(final String[] args)
  {

  }
}
```

```java
InstanceMethodReferenceDemo demo = new InstanceMethodReferenceDemo();

int numToCompare = demo.random.nextInt(MAX);

IntPredicate p = demo::isBigger;

if (p.check(numToCompare))
{
  System.out.println(demo.getNum() + " is bigger than " + numToCompare);
}
else
{
  System.out.println(demo.num + " is smaller or equal than " + numToCompare);
}
```

ConstructorReferenceDemo Class

```java
public class ConstructorReferenceDemo
{
  private int num;

  public ConstructorReferenceDemo(final int num)
  {
    this.num = num;
  }

  public ConstructorReferenceDemo(final ConstructorReferenceDemo n)
  {
    this.num = n.getNum();
  }

  public int getNum()
  {
    return this.num;
  }

  public static void main(final String[] args)
  {
    final int max = 50;

    Random r = new Random();

    int num = r.nextInt(max);

    IntSupplier s1 = ConstructorReferenceDemo::new;

    ConstructorReferenceDemo newObj1 = s1.apply(num);

    System.out.println("new object has a instance value " + newObj1.num);

    ObjectSupplier s2 = ConstructorReferenceDemo::new;

    ConstructorReferenceDemo newObj = s2.apply(newObj1);

    System.out.println("new object has a instance value " + newObj.num);
  }
}
```

```java
public interface IntSupplier
{
  ConstructorReferenceDemo apply(int n);
}
```

```java
public interface ObjectSupplier
{
  ConstructorReferenceDemo apply(ConstructorReferenceDemo o);
}
```

Car Class

```java
public class Car
{
  private String make;
  private String model;
  private int    year;

  public Car(final String p, final String m, final int y)
  {
    make  = p;
    model = m;
    year  = y;
  }

  public String getMake()
  {
    return make;
  }

  public String getModel()
  {
    return model;
  }

  public int getYear()
  {
    return year;
  }
}
```

CarDemo Class

```java
public final class CarDemo
{
  private CarDemo()
  {
  }

  private static List<Car> carsSortedByYear = new ArrayList<>();

  public static void main(final String[] args)
  {

  }
}
```

```java
public static List<String> funFilter(final List<Car> cars)
{
  return cars.stream().filter(car -> car.getYear() > 2000)
      .sorted(Comparator.comparing(Car::getYear))
      .map(Car::getModel).collect(Collectors.toList());
}
```

```java
public static List<String> impFilter(final List<Car> cars)
{
  for (Car car : cars)
  {
    if (car.getYear() > 2000)
    {
      carsSortedByYear.add(car);
    }
  }

  Collections.sort(carsSortedByYear,
      (x, y) -> Integer.valueOf(x.getYear()).compareTo(y.getYear()));

  List<String> models = new ArrayList<>();

  for (Car car : carsSortedByYear)
  {
    models.add(car.getModel());
  }

  return models;
}
```

```java
List<Car> l = Arrays.asList(new Car("Jeep", "Wrangler", 2011),
    new Car("Jeep", "Comanche", 1990), new Car("Dodge", "Avenger", 2010),
    new Car("Buick", "Cascada", 2016), new Car("Ford", "Focus", 2012),
    new Car("Chevrolet", "Geo Metro", 1992));

impFilter(l).forEach(System.out::println);
funFilter(l).forEach(System.out::println);
```

Streams

- Are **similar** to **collections**, but they do **not maintain data**
  - Data come from elsewhere, e.g., a collection, a file and a database
- Are **designed** to **work** well with **lambda expressions**
- Are **immutable** their **processing** may create **new streams**
- Move elements through a **sequence** of **processing steps**, known as a **stream pipeline**, formed by **chaining method calls**
  - A pipeline **begins** with a **data source**, performs various **intermediate operations** and ends with a **terminal operation**
  - **Intermediate operations** are **lazy**: they are not **performed until** a **terminal operation** is **invoked**
- Are defined in the **package java.util.stream**

Producing Streams

- Using the static method “**of**” and **passing** it **some elements** or an **array** of **elements**

- **Converting** a **collection** into a stream by using the “**Collection**” method “**stream**”

- **Converting** the **lines** of a file in **strings** and then the file into a **stream** of **strings** by using the “**Files**” static method “**lines**”

- Any **stream** can be converted into a **parallel stream** by using the method “**parallel**”

Transforming Streams

- The "**map**" method transforms a stream by **applying** a **function** to its elements
- The "**filter**" method transforms a stream by **eliminating** the element that do **not satisfy** the **filter condition**
- The "**limit(n)**" method transforms a stream by **maintaining** the **first n** elements
- The "**skip(n)**" method transforms a stream by **removing** the **first n** elements
- The "**distinct**" method transforms a stream by **removing duplicated** elements
- The "**sorter**" method transforms a stream by **ordering** its elements

Collecting Results

- The methods “**count**”, “**max**”, “**min**” and “**sum**” yield a **single value**
- The method “**toArray**” collects some results in an a**rray**
- The method “**collec**t” collects results in
  - A **list** (passing “**Collectors.toList**” as parameter)
  - A **set** (passing “**Collectors.toSet**” as parameter)
- The method “**collect**” applied to a **stream** of **strings** can collected all the strings into a **single string** (passing “**Collectors.joining**” as parameter)

NumbersDemo Class

```
public final class NumbersDemo
{
  private NumbersDemo()
  {
  }

  public static void main(final String[] args)
  {
    List<Integer> values = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    values.stream().filter(v -> ((v % 2) == 0))
        .forEach(v -> System.out.println(v));

    values.stream().filter(n -> (n < 5)).forEach(System.out::println);

    int sum = values.stream().reduce(0, Integer::sum);

    System.out.println("Sum is: " + sum);
  }
}
```

NamesDemo Class

```java
public final class NamesDemo
{
  private NamesDemo()
  {
  }

  public static void main(final String[] args)
  {
    List<String> names = Arrays.asList("Jake", "Raju", "Kim", "Kara", "Paul",
        "Brad", "Mike");

    System.out.println("Found a 3 letter name?: "
        + names.stream().anyMatch(name -> name.length() == 3));

    System.out.println(
        "Found Kim?: " + names.stream().anyMatch(name -> name.contains("Kim")));

    names.stream().filter(name -> name.length() == 4)
        .forEach(System.out::println);

    names.stream().filter(name -> name.length() == 4)
        .map(name -> name.toUpperCase()).forEach(System.out::println);
  }
}
```