Introduction to Functional Programming

Functional Programming Basic Concepts

Introduction

- In this lecture, we'll understand the functional programming paradigm's core principles and how to practice them in the Java programming language. We'll also cover some of the advanced functional programming techniques.
- This will also allow us to evaluate the benefits we get from functional programming, especially in Java.

What Is Functional Programming?

- Basically, functional programming is **a style of writing computer programs that treat computations as evaluating mathematical functions**.
- So, what is a function in mathematics?
- A function is an expression that relates an input set to an output set.
- Importantly, the output of a function depends only on its input. More interestingly, we can compose two or more functions together to get a new function.

Programming Paradigms

- Of course, functional programming is not the only programming style in practice. Broadly speaking, programming styles can be categorized into imperative and declarative programming paradigms:
- Imperative Programming
  - In an imperative approach, a program is a sequence of steps that changes its state until reaching the target result.
  - That approach is closely related to von Neumann computer architecture, in which instructions in machine code change a global state. The machine state consists of memory contents and processor registries. Most computers use this model.
  - Procedural programming is a type of imperative programming where we construct programs using procedures or subroutines. One of the popular programming paradigms known as object-oriented programming (OOP) extends procedural programming concepts.

Programming Paradigms

- The declarative programming.
  - The **declarative approac**h focuses on the final conditions of the desired result instead of the sequence of steps needed to achieve it.
  - Thus, a declaratively written program isn't a sequence of statements, but a set of property declarations that the resulting object should have.
- The attributes that characterize declarative software are:
  - the final result doesn't depend on any external state
  - lack of any internal state between executions
  - determinism — for the same input arguments, the program always produces the same result
  - order of execution isn't always important — it can be asynchronous

Fundamental Principles and Concepts

- First-Class Functions
- Higher-Order Functions
- Pure Functions

First-Class Functions

- A programming language is said to have first-class functions if it treats functions as first-class citizens. Basically, it means that **functions are allowed to support all operations typically available to other entities**. These include:
  - assigning functions to variables
  - passing them as arguments to other functions
  - returning them as values from other functions.

Higher-order functions

- Higher-order functions are capable of receiving function as arguments and returning a function as a result.
- Traditionally it was only possible to pass functions in Java using constructs like functional interfaces or anonymous inner classes. Functional interfaces have exactly one abstract method and are also known as Single Abstract Method (SAM) interfaces.

```
Collections.sort(numbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer n1, Integer n2) {
    return n1.compareTo(n2)
    }
});
```

Higher-order functions

- As we can see, this is a tedious and verbose technique — certainly not something that encourages developers to adopt functional programming.
- Fortunately, Java 8 brought many **new features to ease the process, like lambda expressions, method references, and predefined functional interfaces**.

```
Collections.sort(numbers, (n1, n2) -> n1.compareTo(n2));
```

- Definitely, this is more concise and understandable. However, please note that while this may give us the impression of using functions as first-class citizens in Java, that's not the case.

Pure Functions

- The definition of pure function emphasizes that a pure function should return a value based only on the arguments and should have no side effects.
- Java, being an object-oriented language, recommends encapsulation as a core programming practice. It encourages hiding an object's internal state and exposing only necessary methods to access and modify it. these methods aren't strictly pure functions.
- Now, this can sound quite contrary to all the best practices in Java.
- Of course, encapsulation and other object-oriented principles are only recommendations and not binding in Java. In fact, developers have recently started to realize the value of defining immutable states and methods without side-effects.

Functional Programming vs. OOP

Functional approach:

1. Order of execution isn't always important, it can be asynchronous.
2. Functions are basic program building elements.
3. It follows the declarative paradigm.
4. It focuses on the result desired and its final conditions.
5. It ensures immutability; programs should be stateless.
6. The primary activity is writing new functions.

OOP approach:

1. A specific execution sequence of statements is crucial.
2. Objects are the main abstractions for data modeling.
3. It follows the imperative paradigm.
4. It focuses on how the desired result can be achieved.
5. State changes are an important part of the execution.
6. The primary activity is building, composing, and extending objects.

Which One Is Better?

- OOP is all about modeling data reflecting real word objects as much as possible. The core elements of OOP are abstraction, encapsulation, inheritance, and polymorphism.
- Functional programming separates the data and behaviors of a program. The main focus of functional programming is writing functions to perform a particular task.
- **Functional programming languages usually perform better when there is a fixed set of things.** As the program grows, we can add new operations to existing things.
- **OOP languages are a good choice when there is a fixed set of actions on things.** In the OOP approach, we can extend code by adding new classes that implement existing methods.
- Both paradigms have a lot of advantages. Thus, most modern programming languages allow developers to use both approaches in a single code base.

Java Lambda Expressions

Lambda Expressions

- Java lambda expressions are new in Java 8.
- Java lambda expressions are Java's first step into functional programming.
- A Java lambda expression is thus a function which can be created without belonging to any class.
- A Java lambda expression can be passed around as if it was an object and executed on demand.
- Java lambda expressions are commonly used to implement simple event listeners / callbacks, or in functional programming with the Java Streams API.
- Java Lambda Expressions are also often used in functional programming in Java .

Using Lambda Expressions

- Suppose you have a class apple with a method getColor and a inventory list holding applies; then you wish to select all the green apples and return them in a list:

```
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;
}
```

Using Lambda Expressions

- But next, somebody would like the list of heavy apples (say over 150g), then you need to modify the code.

```
public static List<Apple> filterHeavyAppleApples(List<Apple> inventory) {

    List<Apple> result = new ArrayList<>();
        for (Apple apple: inventory) {

            if (apple.getWeight() > 150.0 ) {

                result.add(apple);
            }
        }
        return result;
    }

}
```

Using Lambda Expressions

- The above methods only differ in one line; Can we save time and avoid the code duplication of the filter method? Can we treat the code (the single line which differs in the examples) as parameter of the filter method so that we can change it easily?
- Before Java 8, there is no solution; You can't treat code as parameters; the best you might do is create two different classes which contain these two methods separately (and they implement the same interface), while the filter method takes the objects of these classes as parameters. But this is too verbose.
- In Java 8, you can pass code as parameters using lambda expressions (which are considered as objects). **Treating a code segment as an object!!!**

Using Lambda Expressions

- Functional programming is accomplished with lambda expressions.

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

```
public static List<Apple> filterApples(List<Apple> inventory, Predicate<Apple> p){
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {

        if(p.test(apple)) {

            result.add(apple);
        }
    }
    return result;
}
```

Using Lambda Expressions

- To use the above program, You may write:

```
filterApples(colorApple, (Apple a) -> "green".equals(a.getColor())
```

```
filterApples(colorApple, (Apple a) -> a.getWeight() > 150)
```

Java Lambdas and The Functional Interface

- A single method interface is also sometimes referred to as a *functional interface*. Matching a Java lambda expression against a functional interface is divided into these steps:
  - Does the interface have only one abstract (unimplemented) method?
  - Does the parameters of the lambda expression match the parameters of the single method?
  - Does the return type of the lambda expression match the return type of the single method?
- If the answer is yes to these three questions, then the given lambda expression is matched successfully against the interface.
- Java lambda expressions can only be used where the type they are matched against is a single method interface. In the example above, a lambda expression is used as parameter where the parameter type was the **`Predicate<T>`** interface.
- Starting from Java8, interface can also contain default and static methods, which are non-abstract methods.

Lambda Parameters

- A lambda expression consists of a parameter list followed by the arrow token (->) and a body as in:
  - `(parameterList) -> {statements}`
- The following lambdas receive two ints and returns their sum:
  - `(int x, int y) -> { return x+y; }`
  - `(x, y) -> { return x+y; }` (the compiler determines the types by the context)
  - `(x, y) -> x + y;` (when the body contains only one expression)
- Zero Parameters
  - `( ) -> System.out.println("Zero parameter lambda");`
- One Parameter
  - `Param -> System.out.println("One parameter: " + parameter);`

Lambdas as Objects

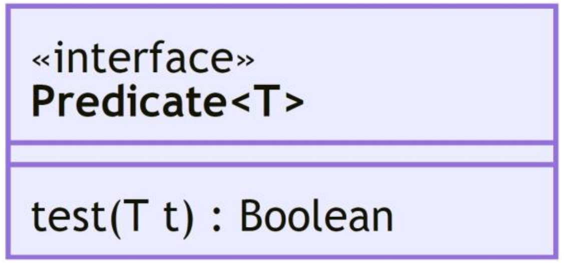
- A Java lambda expression is essentially an object. You can assign a lambda expression to a variable and pass it around, like you do with any other object. Here is an example:

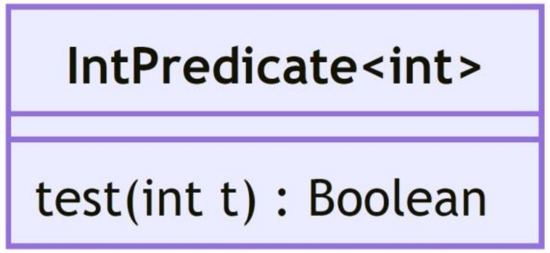
```
public interface MyComparator {

    public boolean compare(int a1, int a2);

}
```

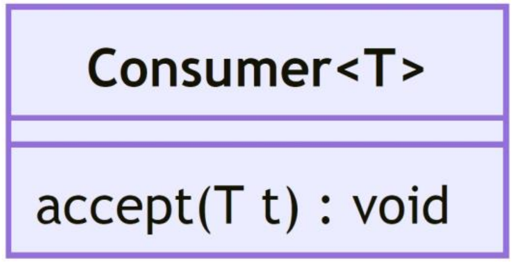
```
MyComparator myComparator = (a1, a2) -> a1 > a2;

boolean result = myComparator.compare(2, 5);
```

Lambda Expressions

- Several basic functional interfaces in java.util.function:
  - `Predicate<T>` -- Contains method `test` that takes a `T` argument and returns a `Boolean`. Tests whether the `T` argument satisfies a condition.
  - `IntPredicate<int>` -- Similar to `Predicate` but takes `int` argument
  - `Consumer<T>` -- Contains method `accept` that takes a `T` argument and returns `void`. Performs a task with its `T` argument

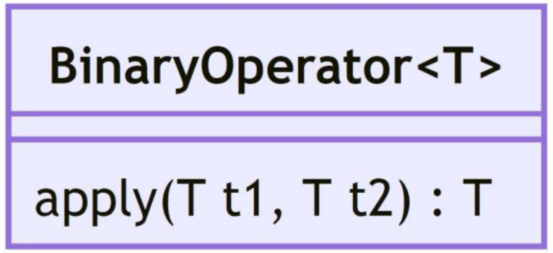


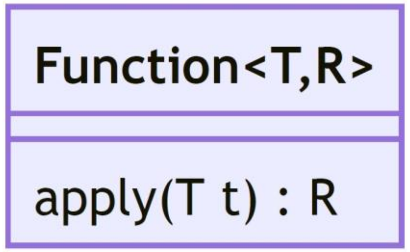




Lambda Expressions

- `BinaryOperator<T>` -- Contains method `apply` that takes two `T` arguments, performs an operation on them and returns a value of type `T`.
- `Function<T,R>` -- Contains method `apply` that takes a `T` argument and returns a value of type `R`.
- In principle, the lambda expression can implement any functional interface, provided that the parameter types and the output types of the lambda match those (the signature or the function descriptor) of the abstract method of the interface.
- You can even create your own functional interface to match the lambda expressions used.





Lambda Expressions

- Examples of defining lambda expressions:
- IntPredicate even = value -> value % 2 == 0;
- IntPredicate greaterThan5 = value -> value > 5;
- IntPredicate evenAndGreaterThan5 = even.and(greaterThan5);
  - and is a default method in the IntPredicate interface, which performs a logical AND between the IntPredicate on which it's called and the Intpredicate it receives as an argument
  - https://docs.oracle.com/javase/8/docs/api/java/util/function/IntPredicate.html
  - Most standard functional interfaces have default methods for producing or combining functions.

Lambda Expressions

- Example of using lambda expressions

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

The lambda is the implementation of the test method from Predicate

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

```java
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class PredicateDemo {
    public static void main(String[] args) {

    List<String> myList = new ArrayList<>();
    myList.add("");
    myList.add("This ");
    myList.add("statement");
    myList.add("");
    myList.add("contain some empty strings");
    myList.add("  ");
    myList.add("");

    Predicate<String> nonEmptyStringPredecate = (String s) -> !s.isEmpty();

    List<String> nonEmpty = filter(myList, nonEmptyStringPredecate);

    System.out.println(nonEmpty);

}
```

This program filters out the empty strings from the myList and returns a list with non-empty strings.

RUN

Lambda Expressions

- Example of using lambda expressions

```
public interface Consumer<T>{
    void accept(T t);
}
```

The lambda is the implementation of the accept method from Consumer.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
    public class ConSumerDemo {
        public static void main(String[] args) {
        forEach(Arrays.asList(1, 2, 3, 4, 5), (Integer i) -> System.out.println(i));
    }

    public static <T> void forEach(List<T> list, Consumer<T> c) {
        for(T i: list) {
            c.accept(i);
        }
    }

}
```

Lambda Expressions

- Example of using lambda expression to shortening the code for running a thread

```java
public interface Runnable {
    public abstract void run();
}
```

```java
public class RunnableDemo {

    public static void main(String[] args) {
        repeatMessage("Hello", 100);
    }
    public static void repeatMessage(String mesg, int count) {
        Runnable r = () ->{
            for (int i=0; i<count; i++) {
                System.out.println(mesg);
                Thread.yield();
            }
        };
        new Thread(r).start();
    }
}
```

This program creates a thread
Which print "Hello" 100 times

Before Java 8, one may need to
Create a taskClass which contains
the run() implementation. And then
Instantiate the taskClass and insert
it into a thread!

Lambda Expressions

- Example of using lambda expressions

```
public interface IntConsumer {
    void accept(int value);
}
```

```
import java.util.function.IntConsumer;
public class IntConsumerDemo {
    public static void main(String[] args) {

    repeat(3, i -> System.out.println("Count " + i));
    repeat(3, i -> System.out.println("Countdown " + (2-i)));

    }

    public static void repeat(int n, IntConsumer action) {
        for(int i=0; i<n; i++) action.accept(i);
    }

}
```

RUN

Review Exercises

- Which of these interfaces are functional interfaces?
- The answer is: **a**

```
a. public interface Adder{
     int add(int a, int b);
   }

b. public interface SmartAdder extends Adder{
     int add(double a, double b);
   }

c. public interface Nothing{

   }
```

Review Exercises

- Which of the following are valid uses of lambda expressions?
- a.

```
public static void execute(Runnable r) {
    r.run();
}

execute(()->{});
```

- b.

```
Predicate<Apple> p = (Apple a) -> a.getWeight();
```

- The answer is : **a**