Functional Programming in Java

CSE 219
Department of Computer Science, Stony Brook University
What is functional programming?

- There is no single precise definition of functional programming (FP)
  - We should think of it as a programming paradigm

- It is **NOT** the opposite of object-oriented programming.
  - Some FP languages are OO, while others are not
  - It’s not a binary choice either. Some language are more FP-friendly than others.
  - We can think of FP as a set of techniques that supplement or replace techniques found elsewhere. For example
    - First class functions
    - Anonymous functions
    - Closures
    - Currying
    - Lazy evaluation
    - Algebraic data types
    - Polymorphism of parameters
What it isn’t …

- Functional Programming ≠ Imperative Programming
- Imperative programs are composed of elements that perform some action
  - *An action* implies an initial state, a final state, and transition from one to the other
  - Strictly imperative programs are described by a series of state transitions
- Functional programs are composed of elements defined in terms of what they *are*, not in terms of what they *do*
  - The main consequence of this is that we can replace an evaluation expression with the corresponding evaluation result
Replacement and Side Effects

Consider this example pseudocode for adding two non-negative values a and b:

1. if b == 0
2.   return a
3. else
4.   increment a
5.   decrement b
6. goto 1.

- This is an imperative-style pseudocode
- Note that the original values are destroyed by this program
- In Java, this will not happen due to passing by value, so local changes will not have a global effect
- This kind of a change is a side effect of the program

Purely functional programs have no observable side effects

- Observable, because in practice, programs are written for computers that are not functional. All computers are based on the same imperative paradigm.
Functions

- Functional programs are built by composing functions that
  - take in a single argument, and
  - return a single value

- Since computers were designed for imperative programs, functions are treated like black boxes as far as the theory of functional programming is concerned
  - In practice, a lot of imperative-style processes and state transitions happen inside functions, but in a way that no effect is observable from the outside
More on side effects

- In practice, it is impossible to avoid side effects
  - A program may create an out of memory error, which is a transition to a special error state
  - A program may have some other error and handle it, which is a transition to a special error state and then back from it
  - Threads, etc.

- So, again, *in practice*
  - when we talk about functional programs, we mean programs written to have no *intentional* side effects
  - and as few non-intentional side effects as possible
Referential Transparency

- Having no intentional observable side effects is just one aspect of functional programming.
- The output of a functional program must only depend on its arguments, and not depend on any other external factors.
  - This means functional code may not read data from the console, a file, a database, etc.
  - A code is said to be **referentially transparent** if it does not cause state changes and does not depend on the external world.
    - A referentially transparent expression can be replaced with its corresponding value without changing the program’s behavior.
    - Referentially transparent codes are far easier for testing, debugging, and formal reasoning.
Properties of Referential Transparency

- **Self-contained.** Doesn’t depend on any external context
- **Deterministic.** Always return the same value for the same argument
- **No Exceptions.** Will never throw an exception.
  - *Runtime exceptions like out of memory error, etc., are due to bugs that need to be fixed, and the imperative part of the program is responsible for this*

A note on thread-safety

- Functional programs are inherently thread-safe because they avoid mutation of shared states (which would be an observable side effect).
- This doesn’t mean that all data has to be immutable. But all shared data must be!
Reasoning with substitution

```java
public static void main(String[] args) {
    int x = add(mult(2, 3), mult(4, 5));
}

public static int add(int a, int b) {
    System.out.println(String.format("Returning %s as the result of " +
                                    "%s + %s", a + b, a, b));
    return a + b;
}

public static int mult(int a, int b) {
    return a * b;
}
```
A Donut

```java
public static Donut buyADonut(CreditCard card) {
    Donut donut = new Donut();
    card.charge(donut.price);  // observable side effect
    return donut;             // return value
}
```

- The above code returns a donut, but also has an observable side effect
- This code will be hard to test (unless you can contact the actual bank or use a mockup bank for testing)
- We are going to convert this simple example into a functional program
  - Which is important because it makes testing a lot easier
A Functional Donut

How can we charge the credit card without observable side effects?

- Represent the charge as a Payment operation
- We will need buyADonut to return the donut as well as the Payment

```java
public class Payment {
    public final CreditCard creditCard;
    public final int amount;

    public Payment(CreditCard creditCard, int amount) {
        this.creditCard = creditCard;
        this.amount = amount;
    }
}
```
A Functional Donut

How can we charge the credit card without observable side effects?
- Represent the charge as a Payment operation
- We will need buyADonut to return the donut as well as the Payment

```java
public class Payment {
    public final CreditCard creditCard;
    public final int amount;

    public Payment(CreditCard creditCard, int amount) {
        this.creditCard = creditCard;
        this.amount = amount;
    }
}
```

How can we return both?
Advantages of Functional Programming

- Functional programming is
  - **programming with functions** and
  - having **no side effects**

- Functional programs
  - are **easy to reason** about and **easy to test**
  - offer a **high level of abstraction** and **reusability**
  - more **robust** than their imperative counterparts
  - offer **better thread-safety** because they avoid shared mutable states
First-Class Functions

Functions become first-class citizens!

- pass functions as arguments to other functions
- return them as the values from other functions
- assign them to variables
- store them in data structures

Before JDK 1.8

- Java ‘kind of’ had a simulation of this capability to some extent with anonymous classes

What all this essentially means is that we can start treating functions as objects in JDK 1.8 and beyond
Functions as Objects

- More private than `private`
  - Functions in a class can now be visible to only method in that class
- Simplification of code in design patterns
- Higher-order functions
  - Methods that take a function as its formal parameter
  - Or return a function object
- So the natural question is: how to create these function objects in Java?