Functional Programming in Java
Part 2

CSE 219
Department of Computer Science, Stony Brook University
Functions as Objects

- The interface `java.util.function.Function<T,R>`
  - To store a function (i.e., take a single argument and returns a single object)
  - `T` is the type of the input argument
  - `R` is the type of the returned object

- These functions can be
  - referenced just like regular Java objects
  - used to build higher-order functions
A simple higher-order function

```java
static Integer compute(Function<Integer, Integer> fun, Integer val) {
    return fun.apply(val);
}
```

- The `compute` function
  - takes another function `fun`
  - applies *that* function to the given value `val`
  - That function can be anything (e.g., increment, inversion, computing the square)
  - It has been *abstracted out* completely by the functional programming approach
Passing a function as an object

- In Java, we pass the reference to objects by value, and the same holds true for functions (which are implemented as methods in Java).
- Such a reference is called a **method reference**
- A function is used as a *Function* object by denoting it by the :: operator.
- If C is a class, and m is a function in that class implemented as a static method, you would refer to its corresponding Function object as `C::m`
- If it is implemented as an instance method, then the way to refer to it functionally is `new C()::m`
Example

```java
public class Example {
    static Integer squareOf(Integer val) { return Math.pow(val, 2); }

    static Integer compute(Function<Integer, Integer> fun, Integer val) {
        return fun.apply(val);
    }

    public static void main(String... args) {
        int i = 11;
        System.out.println(compute(Example::squareOf, i));
        // if it were an instance method, then we would write as below
        // compute(new Example()::squareOf, i)
    }
}
```
Anonymous Functions

- A.k.a., Lambdas
- Take a simple example of adding two integers
- The classical approach is

  ```java
  Integer add(Integer a, Integer b) { return a + b; }
  ```

- In Java 8, we can write this as a lambda expression

  ```java
  BiFunction<Integer,Integer,Integer> add = (a,b) -> a + b;
  ```
BiFunction

- A `java.util.function.BiFunction` is another important interface.
- It represents a function with two arguments and one return object.

```java
BiFunction<Integer,Integer,Integer> add = (a,b) -> a + b;
```

- The argument types can be made explicit in the lambda expression.

```java
(Integer a, Integer b) -> a + b;
```
Optionals

- Null is painful!
- In production environments, for every argument that can be null, you have to check, otherwise in testing environments, your code will not have robustness or graceful degradation

```java
if (arg == null) { throw new NullPointerException(); }
```

- This is sort of a boilerplate code
- Repeating it in every required place is annoying, bloats up your code, and makes everything messy!
- In Java 8, the `optional` was introduced to deal with exactly this problem!
Optionals

- The java.util.Optional class does not have a public constructor
- You should use one of the two methods
  - Optional.of(T value)
  - Optional.ofNullable(T value)
- An Optional has two states
  - If an optional holds an object, it is present
  - If an optional does not hold any object, then it is empty
- Useful methods
  - get(), isPresent(), orElse(T other), etc.
Streams

- Functional programming with collections of objects
- What’s wrong with lists, sets, etc.?
  - Nothing … much
  - But they are not suitable for newer types of situations like online algorithms
  - They are also not suitable for functional programming
- Standard approach
  - Convert your data into a stream
  - Work on the data *functionally*
  - Transform back into suitable non-stream data
Streams

- Streams use a Cascade pattern
  - Most Stream methods return a Stream

- Streams are immutable
  - Every change effectively creates a new Stream

- Streams use lazy evaluation
Creating Streams

- From objects
  
  ```java
  Stream<String> objectStream = Stream.of("Hello", "World");
  ```

- From arrays, etc.
  
  ```java
  Stream<String> arrayStream = Arrays.stream(new String[]{});
  Stream<String> listStream = (new ArrayList<String>()).stream();
  ```
Stream examples

Stream.of(0, 1, 2, 3).forEach(System.out::println);

List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(4, 5, 6);

Stream.of(numbers1, numbers2)
    .flatMap(List::stream)
    .forEach(System.out::println);

Stream.of(0, 1, 2, 3)
    .filter(num -> num < 2)
    .forEach(System.out::println);
Stream examples

Stream.of(0, 1, null, 3)
  .filter(Objects::nonNull)
  .map(num -> num * 2)
  .forEach(System.out::println);

List<Integer> filtered = Stream.of(0, 1, 2, 3)
  .filter(num -> num < 2)
  .collect(Collectors.toList());

String sentence = Stream.of("Who", "are", "you?"
  .collect(Collectors.joining(" ")));
Reducing

- Reducing is also called folding in functional programming
- An amazing tool for recursive computation

```java
Integer totalAgeReduce = roster.stream()
    .map(Person::getAge)
    .reduce(0, (a, b) -> a + b);
```