Imperative to Functional Programming with Java 8

Ioannis Kostaras

JCRETE 2015
Agenda

• Basic Vocabulary and Concepts
• Changing Your Thinking
• Going Deeper
Topics

- Immutability
- First-class functions
- Higher-order functions
- Currying
- Partial application
- Function pipelines
- Function composition

- Recursion
- Map, filter, and reduce
- Continuations
- Memoization
- Monads (computation expressions)
BASIC VOCABULARY & CONCEPTS
Basic Vocabulary and Concepts

• Lambda calculus was introduced by mathematician Alonzo Church in the 1930s
• it makes functions first-class objects
• first-class functions were introduced in C# 3.0 in 2007 while in Java in 2014!
What is a \( \lambda \)-expression

- A \( \lambda \) (lambda) expression is an anonymous function that can be passed as argument or returned as the value of function calls.

- In the Java context they are similar to anonymous methods. Like a method it provides a list of formal parameters and a body—an expression or block—expressed in terms of those parameters.

- They make it easier to distribute processing of collections over multiple threads.

- Collection methods take a function and apply it to every element.
**λ-expressions**

- $f(x) = y$, maps $x \rightarrow y$, e.g. $f(x) = 2x$, maps $x \rightarrow 2x$
- `double dbl(double x) { return 2*x; }`  
  \(\rightarrow (\text{double } x) \rightarrow 2x\)

```java
FileFilter directoryFilter = new FileFilter() {
    public boolean accept(File file) {
        return file.isDirectory();
    }
};

FileFilter directoryFilter =
    (File f) -> f.isDirectory();
// f -> f.isDirectory();
```

- Lambdas are lexically scoped, meaning that a lambda recognizes the immediate environment around its definition as the next outermost scope.

06/09/15  JCrete 2015
λ-expressions

Syntax:

(parameters) -> expression
or
(parameters) -> { statements; }

E.g.
(int x, int y) -> x + y
(x, y) -> x % y
() -> Math.pi
(String s) -> s.toUpperCase() or
String::toUpperCase

x -> 2 * x
c -> { int n = c.size(); c.clear(); return n; }
A functional interface is an interface that has just one abstract method (and zero or more default or implemented methods), and thus represents a functional contract.
**Functional interfaces**

```java
@FunctionalInterface
public interface UnaryOperator<T> {
    T apply(T t);
}
```

- describes the function: \( f: T \rightarrow T \)
java.util.function

• Consumer<T>  T -> void
• Supplier<T>   () -> T
• Predicate<T>  T -> boolean
• Function<T,R> T -> R
• BiFunction<T,U,R> (T,U) -> R
• UnaryOperator<T> T -> T
• BinaryOperator<T> (T,T) -> T
interface Consumer<T> {
    void accept(T t);  // T -> void
}
interface Supplier<T> {
    T get(); // () -> T
}

java.util.function.Supplier<T>
interface Predicate<T> {
    boolean test(T t); // T -> boolean
    default Predicate<T> and(Predicate<? super T> other);
    default Predicate<T> or(Predicate<? super T> other);
    default Predicate<T> negate();
    static <T> Predicate<T> isEqual(Object other);
}
interface Function<T, R> {
    R apply(T t); // T -> R
    default <V> Function<V,R> compose(Function<? super V, ? extends T> before);
    default <V> Function<T,V> andThen(Function<? super R, ? extends V> after);
    static <T> Function<T, T> identity();
}
First-class functions

• Can be used wherever we use values, e.g.

```java
Function<Person, String> byName = person -> person.getName();
List<Person> people;
people.stream().map(byName).collect(toList());
```
First-class functions

- Can be used wherever we use values, BUT

```java
List<Function<Integer, Integer>> functionList = new ArrayList<>()
    { (Integer x) -> double(x), (Integer x) -> square(x) };
```
Function closures

- Functions that use variables defined outside of them, e.g.

```java
public static Predicate<String> startsWith(final String letter) {
    return name ->
        name.startsWith(letter);
}
```

- `letter` is defined in the method that contains the `startsWith()` function and *letter* needs to be "effectively final"
Higher-order Functions

- Functions that take or return other functions
  - pass functions to functions
  - create functions within functions
  - return functions from functions

```java
public static int sum(Function<Integer, Integer> function, int a, int b) {
    if (a > b) return 0;
    else return function.apply(a) + sum(function, a + 1, b);
}
```
Return functions

- Higher-order functions can also return functions:

```java
Function<Integer, Integer> evenOrOdd(int n) {
    return (n % 2 == 0) ?
        x -> 2*x : x -> 3*x;
}
```
Method References

• When a λ-expression simply calls a single method e.g.
  \( s \rightarrow s.\text{toUpperCase}() \)
  the following syntax is preferred:
  \( \text{String::toUpperCase} \)

• This syntax is called method reference
# Method References

<table>
<thead>
<tr>
<th>Category</th>
<th>Syntax</th>
<th>λ-expression</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>ClassName::staticMethod</td>
<td>(args) -&gt; ClassName.</td>
<td>String::valueOf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>staticMethod(args)</td>
<td></td>
</tr>
<tr>
<td>Bound Instance</td>
<td>ObjectName::instMethod</td>
<td>(args) -&gt; ObjectName.</td>
<td>str::replace</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instMethod(args)</td>
<td></td>
</tr>
<tr>
<td>Unbound Instance</td>
<td>ClassName::instMethod</td>
<td>(arg0,rest) -&gt; arg0.</td>
<td>String::concat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instMethod(rest)</td>
<td></td>
</tr>
<tr>
<td>Constructor</td>
<td>ClassName::new</td>
<td>(args) -&gt; new ClassName(args)</td>
<td>File::new</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Map<String, String> map = new TreeMap<>();
map.put("alpha", "A");
map.put("bravo", "B");
map.put("charlie", "C");
String s = "alpha-bravo-charlie";
map.replaceAll(s::replace);
Bound Method References (cont.)

// <T> the type of the first argument to the function
// <U> the type of the second argument to the function
// <R> the type of the result of the function
BiFunction<T, U, R> { R apply(T t, U u); }

void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) // (K,V)->V

String replace(CharSequence target, CharSequence replacement)
Bound Method References (cont.)

```scala
map { "alpha" -> "A",
    "bravo" -> B,
    "charlie" -> "C"}
```

s = “alpha-bravo-charlie”

s::replace(K, V)

s = “A-bravo-charlie”
s.replace("alpha", "A");  // s == "A-bravo-charlie"
s.replace("bravo", "B");  // s == "alpha-B-charlie"
s.replace("charlie", "C");  // s == "alpha-bravo-C"

==================================
{alpha=A-bravo-charlie, bravo=alpha-
B-charlie, charlie=alpha-bravo-C}
Unbound Method References

Map<String, String> map = new TreeMap<>();
map.put("alpha", "A");
map.put("bravo", "B");
map.put("charlie", "C");
map.replaceAll(String::concat);
Unbound Method References (cont.)

String `concat(String str)`;

key.concat(value);

"alpha".concat("A"); // "alphaA"

"bravo".concat("B"); // "bravoB"

"charlie".concat("C"); // "charlieC"

=================================

{alpha=alphaA, bravo=bravoB, charlie=charlieC}
Immutability

- Imperative programming is based on mutability – state change => side effects
- Functional programming is based on immutability – no state change; a new object is created each time
- = sign indicates equation not assignment

```javascript
var = 1
var = var + 1   // Error!!!
```
Currying

• “...the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument (partial application)”. 

• def f(args₁)...(argsₙ₋₁)(argsₙ) = E

  ⇔ def f = (args₁ => (args₂ => ... (argsₙ => E)...) )

  def sum(f: Int => Int)(a: Int , b: Int ) : Int = if (a > b) 0 else f(a) + sum(f)(a + 1, b)
Partial Application

• “…partial application (or partial function application) refers to the process of fixing a number of arguments to a function, producing another function of smaller arity.”

“Arity” means the number of arguments that a function accepts.

• it is a process of binding values to parameters resulting in a function with fewer parameters; it depends on currying

• currying is a process that replaces a single multi-parameter function with a nested set or chain of single-parameter functions
Currying in Java

- Java provides:
  - `Function<T,R>` \( T \rightarrow R \)
  - `BiFunction<T,U,R>` \( (T,U) \rightarrow R \)

- But how can we calculate a function that takes >2 arguments?
  - \( f(x, y, z) = x \times y + z \)
  - There is no `TriFunction<T,U,W,R>` \( (T,U,W) \rightarrow R \)
  - \( (A, B, C) \rightarrow D \Rightarrow A \rightarrow B \rightarrow C \rightarrow D \)
  - \( f(2, y, z) = g(y, z) = 2 \times y + z \Rightarrow \)
  - \( g(3, z) = 2 \times 3 + z \)
Currying in Java

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

private Stream<Integer>
calculate(Stream<Integer> stream,
    Integer a, Integer b) {
    return stream.map(
        ((Function<Integer,
            Function<Integer,
                Function<Integer, Integer>>>)
            x -> y -> z -> x + y * z).
            apply(a).apply(b));
    } // a => x, b => y, stream arg => z
    // [5, 8, 11, 14, 17]
Currying in Java

interface TriIntegerFunction extends Function<Integer, Function<Integer, Function<Integer, Integer>>> {}

private Stream<Integer> calculate(Stream<Integer> stream, Integer a, Integer b) {
    TriIntegerFunction sline =
        x -> y -> z -> x + y * z;
    return
        stream.map(sline.apply(a).apply(b));
}
public class Functions {
    public static <A, B, C> Function<A, Function<B, C>> curry(final BiFunction<A, B, C> f) {
        return (A a) -> (B b) -> f.apply(a, b);
    }

    public static <A, B, C> BiFunction<A, B, C> uncurry(Function<A, Function<B, C>> f) {
        return (A a, B b) -> f.apply(a).apply(b);
    }
}

Function<Integer, Function<Integer, Integer>> c1 = curry((y, z) -> y * z);
Function<Integer, Function<Integer, Integer>> c2 = curry((t, x) -> t + x);
return stream.map(c1.apply(b)).map(c2.apply(a));
// a => x, b => y, stream arg => z, t
CHANGE YOUR THINKING
Expression-based programming

• stop thinking in terms of statement-based programming (imperative)
• start thinking in terms of expression-based programming (functional)
• A statement mutates the current value; does not return anything
• An expression always returns a result or a new value

```
point.moveTo(x1, y1);
newpoint = point.move(x1, y1);
```
Eager vs lazy evaluation

• In **eager evaluation**, an expression is evaluated as soon as it is bound to a variable.
• The alternative to eager evaluation is **lazy evaluation**, where expressions are only evaluated when evaluating a dependent expression, i.e. until its value is needed

Java is an “eager” language

• List<Integer> list = Arrays.asList(1, 2, 3, 4, 5); is eagerly (strictly) evaluated
• Java 8 streams allow for lazy evaluation
Lazy constructions in Java

- `&&` and `||`
- the ternary operator `?:`
- `if... then... else`
- the `for` loop
- the `while` loop
- the Java 8 `Stream`
- the Java 8 `Optional`
Lazy evaluated methods in Java

• Java methods evaluate their arguments eagerly, i.e. before the method is called

```java
public static boolean andMethod (boolean arg1, boolean arg2) {
    return arg1 && arg2;
}
```

• To lazily evaluate its arguments:

```java
public static boolean andMethod (Supplier<Boolean> arg1,
                                 Supplier<Boolean> arg2) {
    return arg1.get() && arg2.get();
}
```
Start thinking immutable

• Reduce encapsulation by creating associations explicitly as maps
  • Think like relational database architecture, i.e. in terms of tables and relations
  • Identify core types i.e. fields
  • Associate with other types via foreign keys
  • normalize
• Create smaller functions that do just one thing; then combine them together using partial application or function pipelining or composition
Example

```java
public class User {
    final List<String> roles;
    String username;
    String emailAddress;
    byte[] passwordHash;

    public User(String username, String emailAddress, byte[] passwordHash) {
        this.username = username;
        this.emailAddress = emailAddress;
        this.passwordHash = passwordHash;
    }

    public String getUsername() {
        return username;
    }

    public String getEmailAddress() {
        return emailAddress;
    }

    public byte[] getPasswordHash() {
        return passwordHash;
    }
}
```

```java
public class UserRoles {
    final User user;
    final List<UserRole> roles;

    public UserRoles(User user) {
        this.user = user;
    }

    public User getUser() {
        return user;
    }

    public List<UserRole> getRoles() {
        return roles;
    }
}
```

```java
public class UserRole {
    final String roleName;

    public UserRole(String roleName) {
        this.roleName = roleName;
    }

    public String getRoleName() {
        return roleName;
    }
}
```
• Find sum of odd numbers:

```java
int[] numbers = {1, 2, 3, 4, 5};
int sumOfOddNumbers =
    IntStream.of(numbers).
    filter(x -> x % 2 == 1).sum();
```
/* Create an output list that contains
* only the odd-length words, converted
* to upper case.
*/

List<String> result = input.stream()
    .filter(s -> s.length() % 2 != 0)
    .map(String::toUpperCase)
    .collect(Collectors.toList());
Filter-map-reduce

Input: $\text{Stream}<\text{T}>$

<table>
<thead>
<tr>
<th>method</th>
<th>$\lambda$</th>
<th>function</th>
<th>output</th>
<th>eager/lazy</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>$\text{T} \rightarrow \text{R}$</td>
<td>$\text{Function}&lt;\text{T},\text{R}&gt;$</td>
<td>$\text{Stream}&lt;\text{R}&gt;$</td>
<td>lazy</td>
</tr>
<tr>
<td>flatMap</td>
<td>$\text{T} \rightarrow \text{Stream}&lt;\text{R}&gt;$</td>
<td>$\text{Function}&lt;\text{T},\text{Stream}&lt;\text{R}&gt;$</td>
<td>$\text{Stream}&lt;\text{R}&gt;$</td>
<td>lazy</td>
</tr>
<tr>
<td>filter</td>
<td>$\text{T} \rightarrow \text{boolean}$</td>
<td>$\text{Predicate}&lt;\text{T}&gt;$</td>
<td>$\text{Stream}&lt;\text{T}&gt;$</td>
<td>lazy</td>
</tr>
<tr>
<td>reduce</td>
<td>$(\text{T},\text{T}) \rightarrow \text{T}$</td>
<td>$\text{BinaryOperator}&lt;\text{T}&gt;$</td>
<td>$\text{Optional}&lt;\text{T}&gt;$</td>
<td>eager</td>
</tr>
<tr>
<td>collect</td>
<td>$\text{R} \rightarrow \text{R}$</td>
<td>$\text{Supplier}&lt;\text{R}&gt;$</td>
<td>$\text{R}$</td>
<td>eager</td>
</tr>
<tr>
<td>forEach</td>
<td>$\text{T} \rightarrow \text{void}$</td>
<td>$\text{Consumer}&lt;\text{T}&gt;$</td>
<td>$\text{void}$</td>
<td>eager</td>
</tr>
</tbody>
</table>

- Streams are evaluated when we apply to them some specific operations called terminal (eager). This may be done only once. Once a terminal operation is applied to a stream, the stream is no longer usable.
Infinite streams - Lazyness

Streams.iterate(seed, UnaryOperator)
  .limit(count)
  .collect(Collectors.toList());

E.g.
Function<Integer, Stream<Integer>>
f = x -> Stream.iterate(1,
  y -> y + 1).limit(x);
Infinite stream example

```java
public class Primes {
    public static boolean isPrime(final int number) {
        return number > 1 && IntStream.rangeClosed(2, (int) Math.sqrt(number)).noneMatch(divisor -> number % divisor == 0);
    }

    private static int nextPrime(final int number) {
        return isPrime(number + 1) ? number + 1 : nextPrime(number + 1);
    }

    public static List<Integer> primes(final int count) {
        return Stream.iterate(2, Primes::nextPrime)
            .limit(count)
            .collect(Collectors.<Integer>toList());
    }
}
```
Terminal Operations

- forEach
- forEachOrdered
- toArray
- reduce
- collect
- min, max
- count

- anyMatch
- allMatch
- noneMatch
- findFirst
- findAny
- iterator
- spliterator
Intermediate Operations

- filter
- map
- mapTo...*
- flatMap
- flatMapTo...*
- Distinct
- sorted

- peek
- limit
- skip
- sequential
- parallel
- unordered
- onClose

*(Int, Long or Double)
Parallel streams

- Things to consider:
  - size of data
  - do we really want to run the lambda expressions concurrently?
  - the code should be able to run independently without causing any side effects or race conditions
  - the correctness of the solution should not depend on the order of execution of the lambda expressions that are scheduled to run concurrently
Function pipelining

List<String> result =
input.stream()
.filter(s -> s.length() % 2 != 0)
.map(String::toUpperCase)
.collect(Collectors.toList());

• stream | filter | map | collect
Function composition

• The ability to compose functions into a chain of operations
• The lack of mutability reduces the chance of errors and makes it easier to parallelize the code
• We can alter a few links in the chain and easily alter the behavior along the way
Recap

• Favor immutability
• Reduce side-effects
• Prefer expressions over statements
• Expressions can be composed
• Design with higher-order functions
GOING DEEPER
Continuations

- “[A] continuation represents the remainder of a computation given a point in the computation.”
- the idea is that a function is provided with another function that is invoked with the result of the first function’s computation
- Java 8 provides CompletableFuture<T> and CompletableStage<T> (a stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes)
- CompletableFuture<T> is functional, monadic, asynchronous and event-driven
Continuations in Java 8

public void processOrder(String custId) {
    return CompletableFuture.supplyAsync(() -> getFromDB(custId))
        .thenApplyAsync(c -> getOrder(c.getCustId()))
        .thenAccept(System.out::println);
}

private Customer getFromDB(String custID) {
    // TODO get customer from database
}

private Order getOrder(String custID) {
    // TODO get order ...
}
Continuation Frameworks

- **Javaflow** *(Play framework uses Javaflow)*
- **RIFE**
- **JauVM** implements tail call / continuation
- **Scala 2.8**
- **Cocoon**
- **Rhino**
- **Jetty** retry request.
- **Coroutines**
- **Jconts**
- **jyield**
- **Kilim**
- **ATCT**
Start thinking immutable - recursion

- Use recursion to avoid mutability
- Example of recursion:

```java
public int fact(int n) {
    return n == 0 ? 1 : n * fact(n-1);
}
```

- Biggest problem with recursion is stack overflow.
- If a function calls itself as its last action, the function’s stack frame can be reused. This is called tail recursion. This resolves stack overflow.
- If the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient for both functions. Such calls are called tail-calls.
Start thinking immutable - recursion

- Convert loops to tail recursion
- Tail recursion example:

```java
public static int foo(int a, int b) {
    int x = ...; int y = ...;
    return foo(x, y);
}
```
• Example: calculate the sum of:

```java
public static int sum(IntUnaryOperator func, int a, int b) {
    if (a > b) return 0;
    else return function.apply(a) +
             sum(function, a + 1, b);
}
```

This is not tail recursive; why!!!
Fibonacci numbers

• Standard fibonacci function is not tail-recursive:

\[ f(n) = f(n - 1) + f(n - 2) \]

• Why?
public static BigInteger fibTailRec(int x) {
    return fibTailRecHelper(BigInteger.ONE, BigInteger.ZERO, BigInteger.valueOf(x));
}

public static BigInteger
fibTailRecHelper(BigInteger acc1, BigInteger acc2, BigInteger x) {
    if (x.equals(BigInteger.ZERO)) {
        return BigInteger.ONE;
    } else if (x.equals(BigInteger.ONE)) {
        return acc1.add(acc2);
    } else {
        return fibTailRecHelper(acc2, acc1.add(acc2), x.subtract(BigInteger.ONE));
    }
}

fibTailRec(9813);  // 9814 => stack overflow
public static BigInteger factTailRec(int number) {
    return factTailRecHelper(BigInteger.ONE, BigInteger.valueOf(number));
}

public static BigInteger factTailRecHelper(BigInteger factorial, BigInteger number) {
    if (number.equals(BigInteger.ZERO)) {
        return BigInteger.ONE;
    } else if (number.equals(BigInteger.ONE)) {
        return factorial.multiply(number);
    } else {
        return factTailRecHelper(factorial.multiply(number), number.subtract(BigInteger.ONE));
    }
}

factTailRec(11799);   // 11800 => stack overflow
Tail Recursion Optimisation (TCO)

- We need a way for the last operation to be a (delayed/lazy) call to itself, and no further computation to carry out on the result upon return.

```java
@FunctionalInterface
public interface RecCall<T> {

    RecCall<T> next();

    default boolean isDone() { return false; }

    default T getValue() { throw new IllegalStateException(); }

    default T apply() {
        return Stream.iterate(this, RecCall::next)
            .filter(RecCall::isDone)
            .findFirst()
            .get()
            .getValue();
    }

    // ...
}
```

06/09/15 JCrete 2015
Tail Recursion Optimisation (TCO)

```java
static <T> RecCall<T> call(final RecCall<T> nextCall) {
    return nextCall;
}
static <T> RecCall<T> done(final T value) {
    return new RecCall<T>() {
        @Override
        public boolean isDone() {
            return true;
        }
        @Override
        public T getValue() {
            return value;
        }
        @Override
        public RecCall<T> next() {
            throw new IllegalStateException();
        }
    };
}
```
**TCO fibonacci function**

```java
public static BigInteger fibTailRec(int x) {
    return fibTailRecHelper(BigInteger.ONE, BigInteger.ZERO, BigInteger.valueOf(x).apply());
}

public static BigInteger fibTailRecHelper(BigInteger acc1, BigInteger acc2, BigInteger x) {
    if (x.equals(BigInteger.ZERO)) {
        return Recall.done(BigInteger.ONE);
    } else if (x.equals(BigInteger.ONE)) {
        return Recall.done(acc1.add(acc2));
    } else {
        return Recall.call(() -> fibTailRecHelper(acc2, acc1.add(acc2), x.subtract(BigInteger.ONE)));
    }
}
```

```
06/09/15 JCrete 2015
```
TCO factorial function

```java
public static BigInteger factTailRec(int number) {
    return factTailRecHelper(BigInteger.ONE,
    BigInteger.valueOf(number).apply());
}

public static RecCall<BigInteger> factTailRecHelper(BigInteger factorial, BigInteger number) {
    if (number.equals(BigInteger.ZERO)) {
        return Recall.done(factTailRec);
    } else if (number.equals(BigInteger.ONE)) {
        return Recall.done(factorial.multiply(number));
    } else {
        return Recall.call(() -> factTailRecHelper(
        factorial.multiply(number),
        number.subtract(BigInteger.ONE)));
    }
}
```

Tail recursive lazily executed

```java
factTailRec(500_000);
```
Tuples

- ordered collections of values of different types of fixed size, e.g.
  (person, "John", 42, 1.78)

- Java supports tuples:
  - Implicitly, e.g.:
    ```java
    int sum2(int a, int b) {
        return a+b;
    }
    (int a, int b) -> a + b;
    ```
  - but we cannot use them for a function’s return values
Tuples

- Explicitly, e.g.

```java
/** A Tuple: double -> (int, double) for integer and decimal parts */
public class Tuple {
    public final int integerPart;
    public final double decimalPart;

    public Tuple(int integerPart, double decimalPart) {
        super();
        this.integerPart = integerPart;
        this.decimalPart = decimalPart;
    }

    @Override
    public String toString() {
        return String.format("(%s, %s)", integerPart, decimalPart);
    }
}
```
private static Tuple split(double x) {
    int integerPart = (int) x;
    return new Tuple(integerPart, x - integerPart);
}
System.out.println(split(3.14));
(3, 0.14000000000000012)
Fibonacci function with tuples

- 1, 1, 2, 3, 5, 8, 13, 21, ...
- (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), ...

```java
Tuple2<BigInteger, BigInteger> seed = new Tuple2<>(BigInteger.ONE, BigInteger.ONE);
UnaryOperator<Tuple2<BigInteger, BigInteger>> fbc = x -> new Tuple2<>(x.elem2, x.elem1.add(x.elem2));
BigInteger fiboCoRecursive = Stream.iterate(seed, fbc)
.map(x -> x.elem1)
.limit(1_000_000).reduce((a, b) -> b).get();
```
Memoization

• Memoization means caching the results of functions in order to speed them up when they are called several times with the same argument.
• The first call implies computing and storing the result in memory before returning it.
• Subsequent calls with the same parameter imply only fetching (lookup) the previously stored value and returning it.
• Memoizing is about maintaining state between function calls.
• A memoized function is a function which behavior is dependent upon the current state.
public class Cache {
    private static final Map<Integer, Integer> cache = new ConcurrentHashMap<>();

    public static Integer square(Integer x) {
        return cache.computeIfAbsent(x, y -> y * y);
    }
}

---

int sq3 = Cache.square(3);
public class Cache {
    private static final Map<Integer, Integer> cache = new ConcurrentHashMap<>();

    public static Function<Integer, Integer> square = x ->
        cache.computeIfAbsent(x, y -> y * y);
}

int sq3 = Cache.square.apply(3);

We have to repeat this modification for all functions.
Function\langle Integer, Integer \rangle \ f = x \rightarrow \ x * \ x;

Function\langle Integer, Integer \rangle \ g = Memoizer.memoize(f);  // reference to the function to be memoized

int sq3 = g.apply(3);

- All values returned by function \( g \) will be calculated through the original function \( f \) the first time, and returned from the cache for all subsequent accesses.
public class Memoizer {
    private Memoizer() {}

    public static <T, U> Function<T, U> memoize(final Function<T, U> function) {
        final Map<T, U> cache = new ConcurrentHashMap<>();
        return input -> cache.computeIfAbsent(input, function::apply);
    }
}
What about functions with >1 arguments?

- i.e. functions of *tuples* that may also return *tuples*
- More difficult to store multiple values to a Map
- For 2 arguments, \( f \) can be a
  \[
  \text{BiFunction}\langle T, U, R \rangle:
  \]
  \[
  \text{BiFunction}\langle \text{Integer, Integer, Integer} \rangle \ h = (x, y) \rightarrow x \times x + y \times y;
  \]
- Or use *currying* for \( \geq 2 \) arguments:
  \[
  \text{Function}\langle \text{Integer, Function}\langle \text{Integer, Integer} \rangle \rangle \ hc = x \rightarrow y \rightarrow x \times x + y \times y;
  \]
Memoizer with a parameter

```java
public class Memoizer2 {
    private Memoizer2() {}

    public static <T, R> R memoize(final BiFunction<Function<T, R>, T, R> function, final T input) {
        final Map<T, R> cache = new ConcurrentHashMap<>();

        Function<T, R> memoized = new Function<T, R>() {
            @Override
            public R apply(final T input) {
                return cache.computeIfAbsent(input, key -> function.apply(this, key));
            }
        };

        return memoized.apply(input);
    }
}
```
Memoizer with currying

Function<Integer, Function<Integer, Function<Integer, Integer>>> f2m =
    Memoizer.memoize(x ->
        Memoizer.memoize(y -> x*x + y*y));

f2m.apply(2).apply(3);

Function<Integer, Function<Integer, Function<Integer, Integer>>> f3 =
    x -> y -> z -> x*x + y*y + z*z;
Function<Integer, Function<Integer, Function<Integer, Integer>>> f3m =
    Memoizer.memoize(x -> Memoizer.memoize(y ->
        Memoizer.memoize(z -> x*x + y*y + z*z)));

f3m.apply(2).apply(3).apply(-1);
Memoizer with tuples (cont.)

private static final Function\<Tuple3\<Integer, Integer, Integer\>, Integer\> ft = t ->
 t.elem1*t.elem1 + t.elem2*t.elem2 +
 t.elem3*t.elem3;

private static final Function\<Tuple3\<Integer, Integer, Integer\>, Integer\> ftm =
 Memoizer.memoize(ft);

-----------------------------------------
ftm.apply(new Tuple3<>(2, 3, 4));
Memoizer with tuples (cont.)

```java
public class Tuple3<T, U, V> {
    public final T elem1;
    public final U elem2;
    public final V elem3;
    public Tuple3(T t, U u, V v) {
        elem1 = Objects.requireNonNull(t);
        elem2 = Objects.requireNonNull(u);
        elem3 = Objects.requireNonNull(v);
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Tuple3)) return false;
        else {
            Tuple3 that = (Tuple3) o;
            return elem1.equals(that.elem1) &&
                   elem2.equals(that.elem2) &&
                   elem3.equals(that.elem3);
        }
    }
    @Override
    public int hashCode() {
        return elem1.hashCode() + elem2.hashCode() + elem3.hashCode();
    }
}
06/09/15  JCrete 2015
```
Monads (Computation expressions)

- A **monad** is a structure that represents computations defined as sequences of steps
- Monads can be chained together in order to build pipelines that process data in steps
- One of the uses for a monad is retain state without resorting to mutable variables
- Monads are a great way to create *workflows* where you have to manage data, control logic, and state (side effects)
- Continuations are a key aspect of computational expressions
int a = 5, b = 6;
int c = a + b;

==================================

int pipeFunc(int x, IntUnaryOperator f) {
    return f.applyAsInt(x);
}

int result = pipeFunc(5,
    a -> pipeFunc(6,
        b -> pipeFunc(a+b, c -> c)));

Simple Monad example
Monads

- Formally, a monad consists of a type constructor $M$ and two operations, $bind$ and $return$ (where $return$ is often also called $unit$):
  - The $return$ operation takes a value from a plain type and puts it into a monadic container using the constructor, creating a monadic value.
  - The $bind$ operation takes as its arguments a monadic value and a function from a plain type to a monadic value, and returns a new monadic value.
Monads

• A monad is a set of three things:
  • A parameterized type $M<T>$
  • A unit or return function $T \rightarrow M<T>$
  • A bind operation
    $M<T> \ bind \ T \rightarrow M<U> = M<U>$

• Java supports monads, e.g.: `Optional<T>`
  • Parameterized type: `Optional<T>`
  • unit: `Optional.of()`
  • bind: `Optional.flatMap()`
E.g.

```java
// unit
Optional<Integer> maybeInteger = Optional.of(1);

// bind
Optional<Integer> maybePlusOne = maybeInteger.flatMap(
    n -> Optional.of(n + 1));
```

*Optional* is a part of Java 8's *Optional* class which is used to represent values that are either present or not. It avoids running null-check code that can be error-prone.
Other monads in Java 8

- Streams
- `CompletableFuture<T>`
- [Better-Java-Monads](http://example.com) by Jason Goodwin
  - Try
  - Futures
    - `List<CompletableFuture<T>>`  $\rightarrow$ `CompletableFuture<List<T>>`
Monads

- Maybe or Try
- I/O
- Identity
- State
- Continuation
- …
State Monad in Java 8

• a state monad allows to attach state information of any type to a calculation
• a state monad maintains state within a specific workflow.
• a state monad is basically a function from a state to a pair (state, content).
• Parameterized type: \( <T, S> \):
  • unit: \( t \rightarrow (t, s) \)
  • bind: \( x \rightarrow (y, s) \)
• See ref[9].
Conclusion

OO style

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance
- Statements
- imperative

Functional style

- Higher order functions
- Function composition
- Immutability
- Expressions
- declarative
Conclusion

• Java 8 is not a fully functional programming language
• Allows you to combine imperative with functional style
• Thinking functionally can help you become a better coder
• Think of immutability
• Think of (small) functions (lambdas)
• Think of Streams
References

References

References


16. Infinite Streams