Functional Programming

Terence Sim

FP

OOP

What is Functional Programming?

It is NOT merely having functions as first-class objects, or lambdas.

- Lambdas and functional interfaces are language features

It is NOT at odds with Object-oriented Programming

- Although they view the world differently

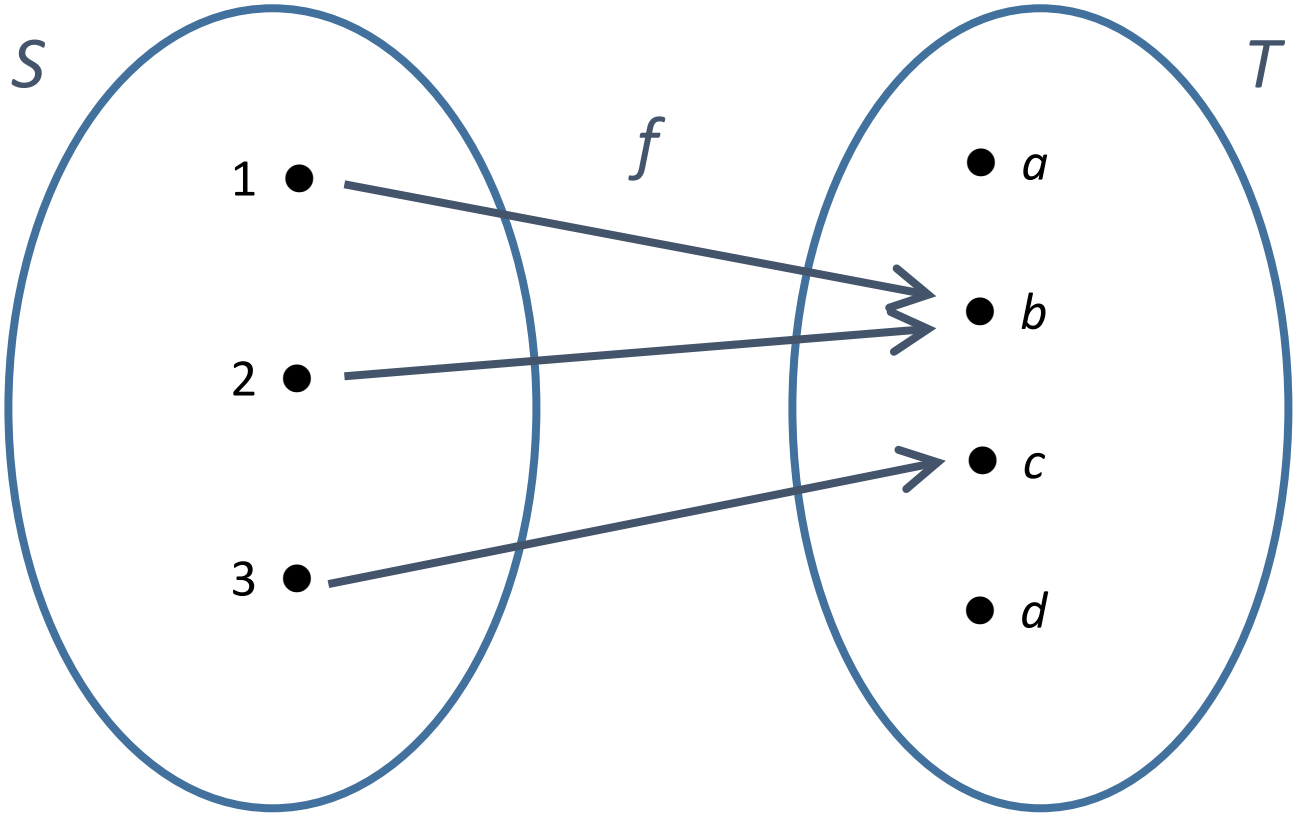
FP is a style of programming that emphasizes

- No side effects
- Immutable data
- Declarative (rather than imperative)
- Lazy evaluation

You can do FP in Java!

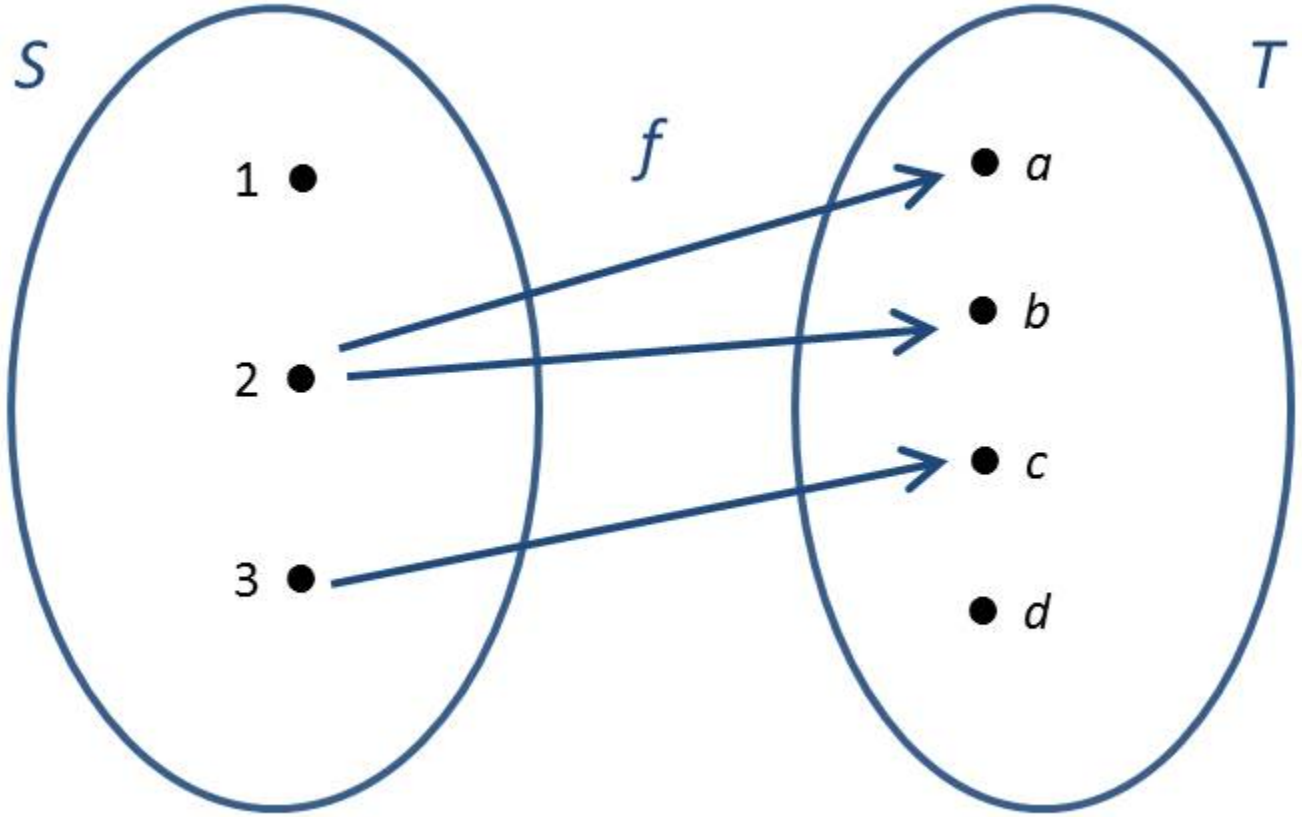
Math vs Programming

In Math, a function is an association between a domain set *S*, and a co-domain set *T*, such that ***every element in S is associated with exactly one element in T.***



Math vs Programming

This is not a function. Why not?



Math vs Programming

In programming, functions are abstractions of computational actions. They may not be the same as mathematical functions. Those that are are called *pure functions.*

```
“cs2030”.length()
//length is a pure function. Domain: String,
//Co-domain: int

System.out.println(“cs2030”)
//This is not a pure function. It does not
//return anything, but causes a side effect
```

Pure functions

Have no side effects

eg. input/output, throwing exceptions, changing external state

Do not mutate data

Instead, a change of state is represented by new data

Always return a value

Thus, a `void` function is not pure

That is the same for the same inputs

Return value is fully determined by the input arguments; and nothing else

Is `add` a pure function?

Yes!

```
class A {
    int x;

    A(int x) { this.x = x; }

    int add(int y) { return x + y; }
}
```

```
A myA = new A(5);
myA.add(6) is syntactic sugar for add(myA, 6)
```

What about this add?

No!

```
class A {
    static int x;

    A(int x) { A.x = x; }

    int add(int y) { return x + y; }
}
```

```
A myA = new A(5);
System.out.println(myA.add(6)); // 11
A myA2 = new A(7);
System.out.println(myA.add(6)); // 13
```


5x-2=13
then
x = 3
But professor, you said yesterday that x was equal to 2!

FP benefits

1. Pure functions are easier to reason with
2. Testing and debugging are easier
3. Declarative style aids understanding
   - Declarative = what to do
   - Imperative = how to do it
4. Good for parallel/concurrent computation
5. Results can be cached to speed up subsequent retrieval
   - Referential transparency: an expression can be replaced by its value without changing program's behavior

```
int[] x = {-3, 4, 6, 8};
var y = pureFunction(x);
```

```
41 @    static MappedField validateQuery(final Class clazz, final Mapper mapper, final StringBuilder origProp, final FilterOperator op, final
42          MappedField mf = null;
43          final String prop = origProp.toString();
44          boolean hasTranslations = false;
45          if (!origProp.substring(0, 1).equals("$")) {
46              final String[] parts = prop.split( regex: "\\.");
47              if (clazz == null) { return null; }
48              MappedClass mc = mapper.getMappedClass(clazz);
49              //CHECKSTYLE:OFF
50              for (int i = 0; ; ) {
51                  //CHECKSTYLE:ON
52                  final String part = parts[i];
53                  boolean fieldIsArrayOperator = part.equals("$");
54                  mf = mc.getMappedField(part);
55                  //translate from java field name to stored field name
56                  if (mf == null && !fieldIsArrayOperator) {
57                      mf = mc.getMappedFieldByJavaField(part);
58                      if (validateNames && mf == null) {
59                          throw new ValidationException(format("The field '%s' could not be found in '%s' while validating - %s; if you wis
60                      }
61                      hasTranslations = true;
62                      if (mf != null) {
63                          parts[i] = mf.getNameToStore();
64                      }
65                  }
66                  i++;
67                  if (mf != null && mf.isMap()) {
68                      //skip the map key validation, and move to the next part
69                      i++;
70                  }
71                  if (i >= parts.length) {
72                      break;
73                  }
74                  if (!fieldIsArrayOperator) {
75                      //catch people trying to search/update into @Reference/@Serialized fields
76                      if (validateNames && !canQueryPast(mf)) {
77                          throw new ValidationException(format("Cannot use dot-notation past '%s' in '%s'; found while validating - %s", pa
78                      }
79                      if (mf == null && mc.isInterface()) {
80                          break;
81                      } else if (mf == null) {
82                          throw new ValidationException(format("The field '%s' could not be found in '%s'", prop, mc.getClazz().getName())
83                      }
84                      //get the next MappedClass for the next field validation
85                      mc = mapper.getMappedClass((mf.isSingleValue()) ? mf.getType() : mf.getSubClass());
86                  }
87              }
88
89              //record new property string if there has been a translation to any part
90              if (hasTranslations) {
91                  origProp.setLength(0); // clear existing content
92                  origProp.append(parts[0]);
93                  for (int i = 1; i < parts.length; i++) {
```

What's a prop?

What's a part?

Eeek!

Why all the null checks?

Control the loop

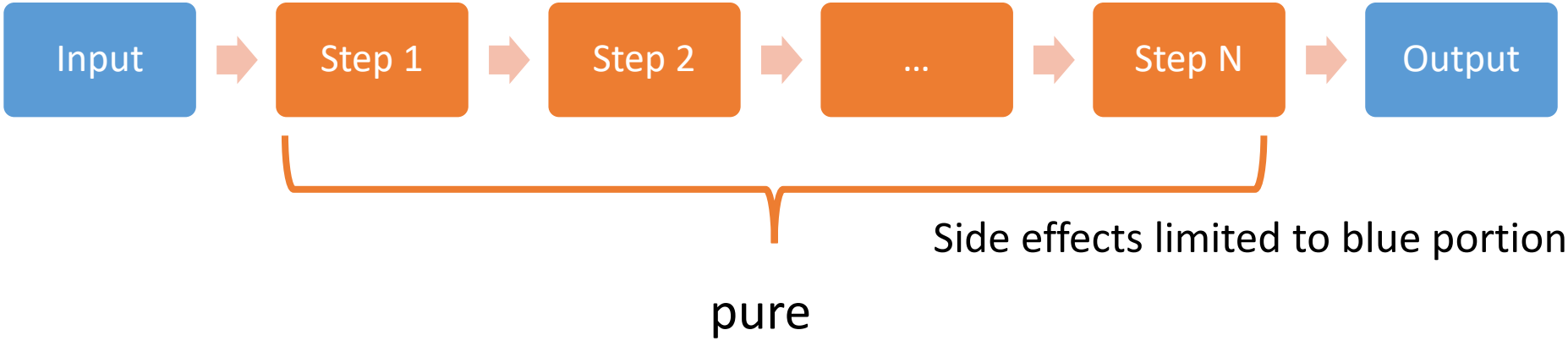
Comments, because code is unclear

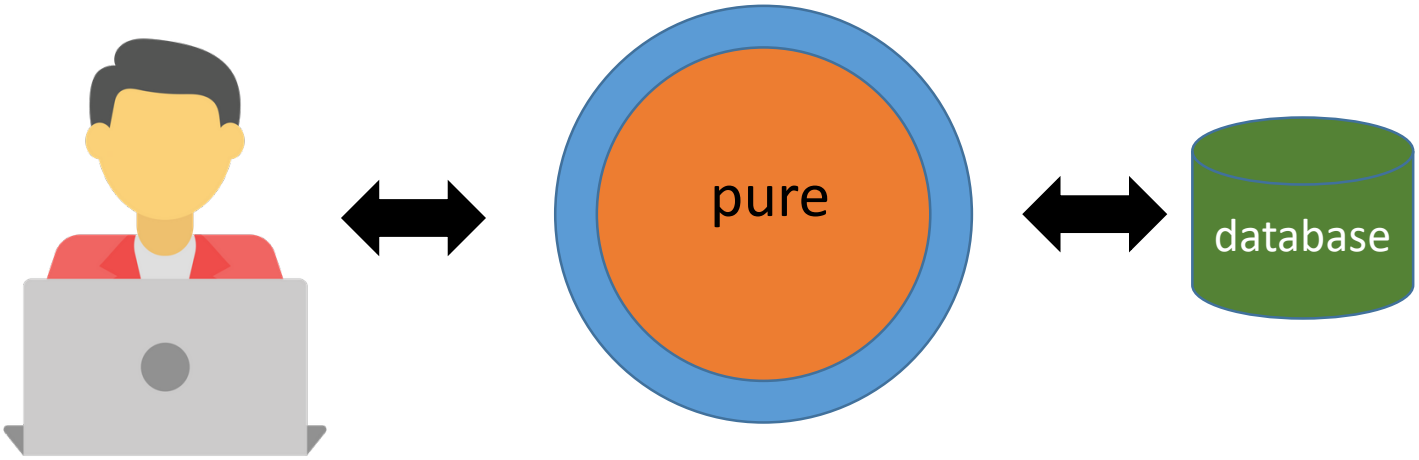
Parameter mutation!

```
// what is x here?
```

Real-world apps need side effects

We strive to limit side effects to certain parts of our code, keeping as much as possible purely functional.

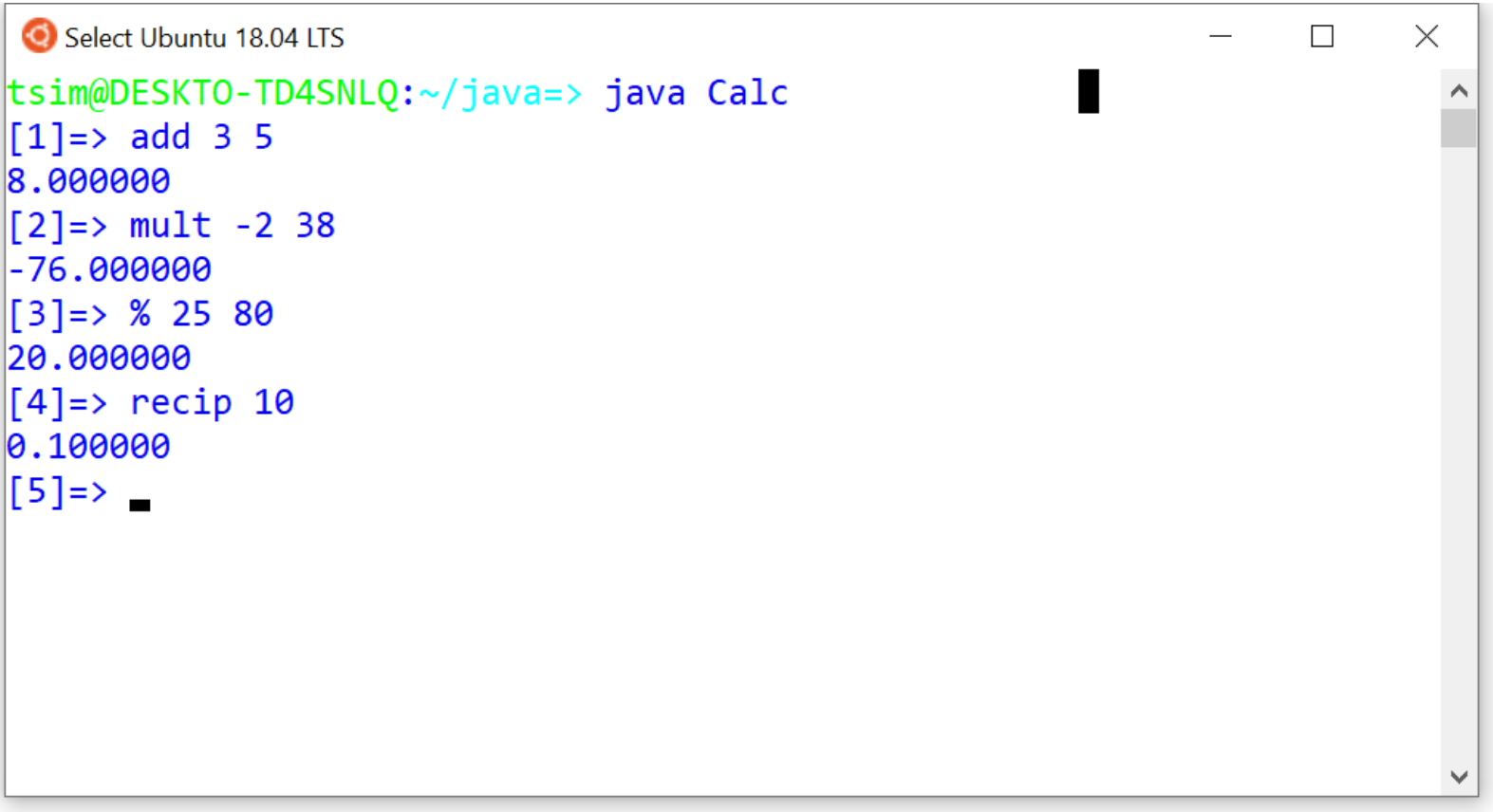




You have seen FP style!

```
new ArriveEvent(new Customer(2, 0.6),
   Arrays.asList(new Server(1, false, false, 1.0)))
   .execute()
   .execute()
   .execute()
```

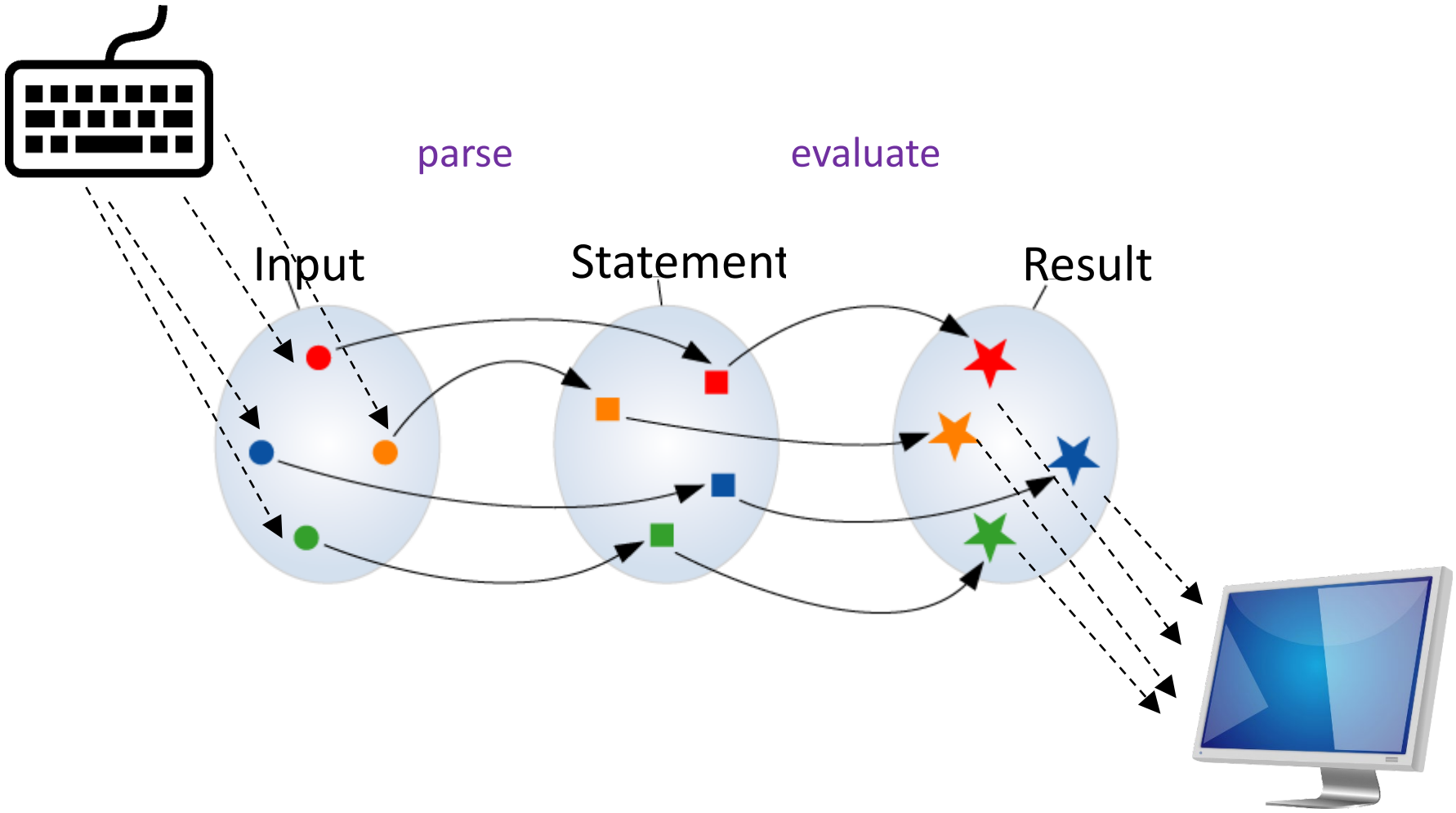
Command-line calculator revisited



Command-line Calculator revisited

```
public class Calc {
    public static void main(String[] args) {
        Statement.initialize();

        try {
            while (true) {
                Input input = Input.readInput();
                Statement statement = Statement.parse(input);
                Result result = statement.evaluate();
                result.display();
            }
        } catch (NoSuchElementException e) {
            //break out of while loop; end program
        }
    }
}
```

What is going on?



Hashtable: command – lambda entry

key	value
“add”	`x -> x[0] + x[1]`
“mult”	`x -> x[0] * x[1]`
“recip”	`x -> 1.0 / x[0]`
“%”	`x -> x[0] * x[1] / 100.0`

command arg1 arg2

```
add   3  5
mult  -8 20
recip 40
%     30  80
```

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Hashtable.html

Command-line Calculator revisited

```
 public Statement(String input) {
      Scanner sc = new Scanner(input);
      // Parse command
      String c = sc.next();
      if (!commandTable.containsKey(c)) {
          this.command = "err";
          this.message = String.format("Bad
command! %s", c);
          return; }
      this.command = c;
```

Parsing arguments

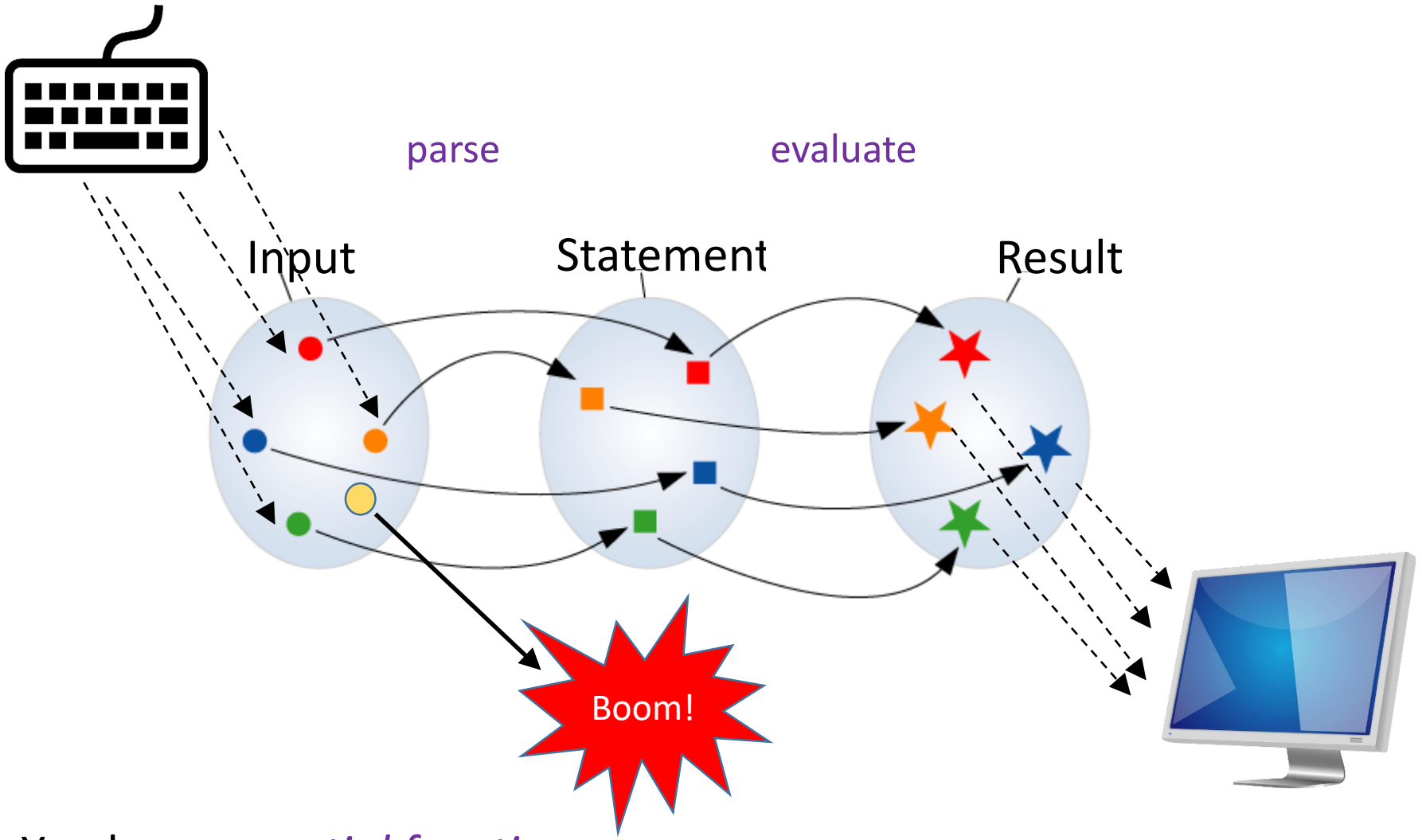
```
        int numberArgs = argsTable.get(this.command);
        this.arguments = new Double[numberArgs];
        try {
            for (int i = 0; i < numberArgs; i++) {
              this.arguments[i] = sc.nextDouble(); }
            if (sc.hasNext()) {
              this.command = "err";
              this.message = "Too many arguments!"; }
        } catch (InputMismatchException e) {
            this.command = "err";
            this.message = "Bad arguments!";
        } catch (NoSuchElementException e) {
            this.command = "err";
            this.message = "Too few arguments!"; }
}
```

Evaluating the statement

```
 public Result evaluate() {
      if (!this.command.equals("err")) {
          double r =
commandTable.get(this.command).apply(this.argume
nts);
          this.message = String.format("%f", r);
      }
      return new Result(this.message);
}
```

Notice how much code is needed to detect and handle exceptions. Also, how it is intertwined with normal processing.

What is going on?



You have a *partial function*

How Functors can help

Functor is a design pattern (ie. best coding practice):

- Immutable generic box (ie. parametrized type)
- Constructor: (different names)
- Method: map

Functor is a box (aka *context*) to contain a thing, which then can be manipulated safely.

Restores the partial function back into a pure one.

Revised Statement class

```
private final String command;
private final Double[] arguments;
private final Scanner scanner;

public Statement(Input input) {
    this(new Scanner(input.getLine()), null, null); }


private Statement(Scanner sc, String c, Double[] args) {
    this.scanner = sc;
    this.command = c;
    this.arguments = args; }


public Statement parseCommand() {
    String c = scanner.next();
    if (!commandTable.containsKey(c))
       throw new RuntimeException(c);//unknown command
    return new Statement(scanner, c, null); }
```

Revised Statement class

```
public Statement parseArguments() {
    int numberArgs = argsTable.get(this.command);
    Double[] args = new Double[numberArgs];
    for (int i = 0; i < numberArgs; i++) {
       args[i] = scanner.nextDouble();
    }

    if (scanner.hasNext()) //too many arguments
       throw new IllegalArgumentException(
       String.format("Too many arguments: %s",
       scanner.nextLine()));
    return new Statement(scanner, this.command, args);
}
```

When we detect error, just throw exception.

No need to catch exceptions. Let the functor handle exceptions.

Revised Statement class

```
public Result evaluate() {
   double r = commandTable.get(this.command)
                .apply(this.arguments);
   return new Result(String.format("%f", r));
}
```

Evaluate no longer has to check for errors.

Revised REPL

```
public class CalcM {
  public static void main(String[] args) {
    Statement.initialize();
    try {
      while (true) {
        Sandbox.make(Input.readInput())
          .map(Statement::new)
          .map(Statement::parseCommand, "Bad command!")
          .map(Statement::parseArguments, "Bad
            arguments!")
          .map(Statement::evaluate, "Bad result!")
          .consume(Result::display); }
    } catch (NoSuchElementException e) {
          //break out of while loop; end program
} } }
```

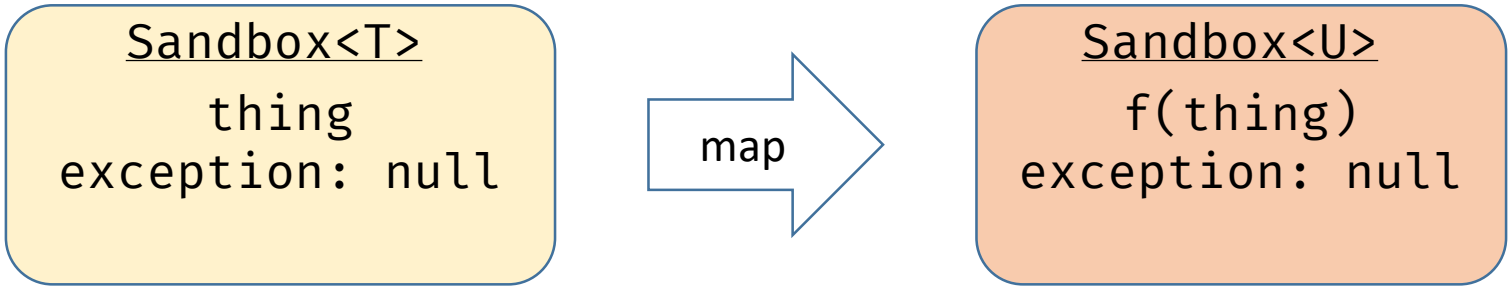
What does the Sandbox look like?

```
public final class Sandbox<T> {
    private final T thing;
    private final Exception exception;

    private Sandbox(T thing, Exception ex) {
        this.thing = thing;
        this.exception = ex; }

    public static <T> Sandbox<T> make(T thing) {
        return new
         Sandbox<T>(Objects.requireNonNull(thing),
             null);}
```

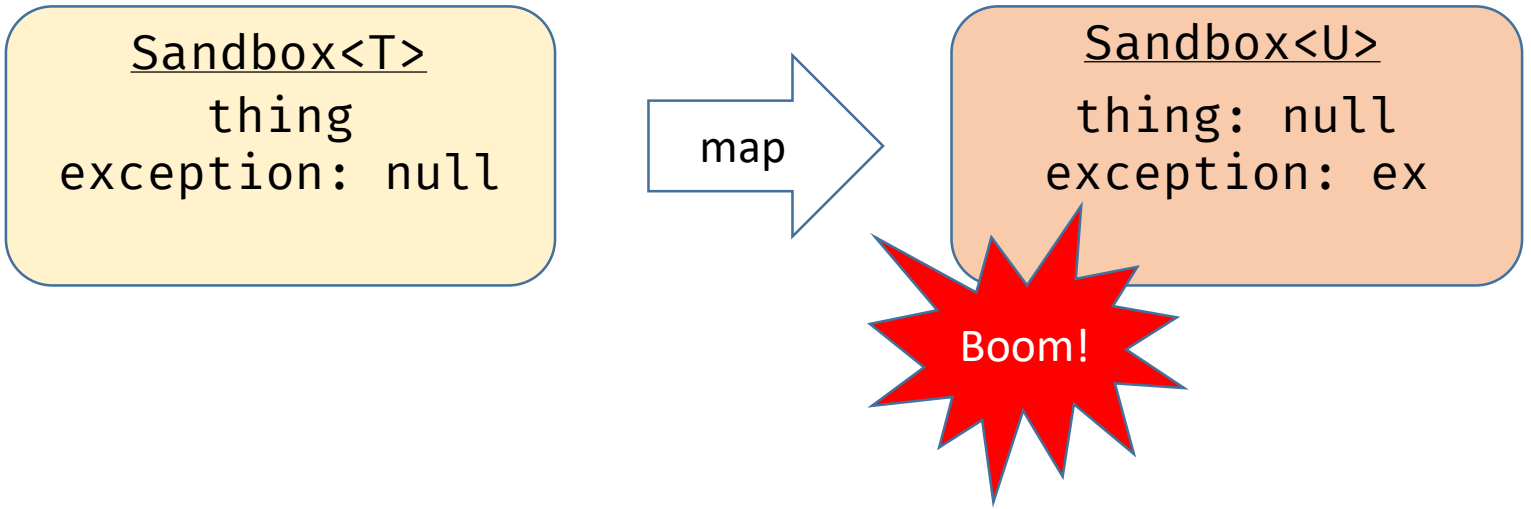
```
public <U> Sandbox<U> map(Function<T,U> f) {
        return map(f, ""); }


public <U> Sandbox<U> map(Function<T,U> f,
      String errorMessage) {
  if (isEmpty())
    return new Sandbox<U>(null, this.exception);
  try {
    return new Sandbox<U>(f.apply(thing), null);
  }
  catch (Exception ex) {
    return new Sandbox<U>(null,
            new Exception(errorMessage, ex));
  } }
```

The action of `map`

Normal operation



Exception is caught



```
public void consume(Consumer<T> eat) {
    if (isEmpty() && hasNoException())
        return;
    if (isEmpty() && !hasNoException())
        handleException();
    if (!isEmpty() && hasNoException())
            eat.accept(thing);
}
```

`consume` ends the processing pipeline: the `eat` method accepts the object and returns nothing.

```
private void handleException() {

   System.err.printf(this.exception.getMessage());
   Throwable t = this.exception.getCause();
   if (t != null) {
      String msg = t.getMessage();
      if (msg != null)
         System.err.printf(": %s", msg);
    }
    System.err.println(""); }
```

`handleException` prints the exception message to `System.err`, including the `cause of` the exception, if any.

The revised calculator code is available in Luminus. Do take a look, and compare it with the first version.

Functor laws

A functor is a generic type that contains a thing (aka *payload*), and provides a constructor (`make`), and a `map` method that transforms the payload. These must satisfy 2 laws:

1. Identity:
   ```
   make(t).map(x -> x).equals(make(t))
   ```
2. Associativity:
   ```
   make(t).map(f).map(g).equals(
       make(t).map(f.andThen.g))
   ```

Functors in Java

Optional<T>:

Separates logic for handling `null` values from normal processing.

Constructor: `of`, Method: `map`

Stream<T>:

Provides lazily evaluated lists

Constructor: `of`, Method: `map`

ArrayList<T>:

Stores multiple values which are accessible by a numeric index (via the `get` method).

Constructor: `Arrays.asList`

But missing a `map` method.

Easy to provide a `map` for `ArrayList`

```
<T,U> ArrayList<U> map(ArrayList<T> list,
                       Function<T,U> f) {
    ArrayList<U> result = new ArrayList<>();
    for (T item: list)
        result.add(item);
    return result;
}
```

Interface Function<T,U>

Surprise! This is also a functor.

Think of it as an `ArrayList` whose index is of type `T`, and payload is of type `U`.

eg. The `boolStringFunc` stores "`hello`" at index `true`, and "`goodbye`" at index `false`.

```
Function<Boolean, String> boolStringFunc =
   v -> v ? "hello" : "goodbye";
```

The Constructor is Java's assignment of lambda expression, and the `map` method is `andThen`

```
boolStringFunc.andThen(String::length)
```

This is a `Function<Boolean, Integer>` that stores 5 at index `true`, and 7 at index `false`.

Monads

What happens if the Sandbox `map` method is given `f` with signature: `Function<T, Sandbox<U>>` ?

Then, `make(t).map(f)` returns `Sandbox<Sandbox<U>>`

That is, the transformed payload is now wrapped in a Sandbox inside another Sandbox. This is troublesome!

Thus, it is often useful to have a `flatmap` method, which "flattens" or unwraps one of the box. So:

`make(t).flatmap(f)` returns `Sandbox<U>`

The data type is then called a Monad.

Monads

A Monad is a parametrized type that contains a thing, along with a constructor (`of`, aka `unit`), and a method `flatmap` (aka `bind`).

`Optional<T>` and `Stream<T>` are also monads.

Monads must obey 3 laws. Please look them up:

https://medium.com/@afcastano/monads-for-java-developers-part-1-the-optional-monad-aa6e797b8a6e

Lecture Summary

Functional Programming is a style that emphasizes pure functions, and declarative coding.

FP makes code easier to reason about, test, debug, optimize, and parallelize.

Functors are generic boxes (context) that contain values (payload) and provide useful abstractions for side effects.

Monads are also common in FP.