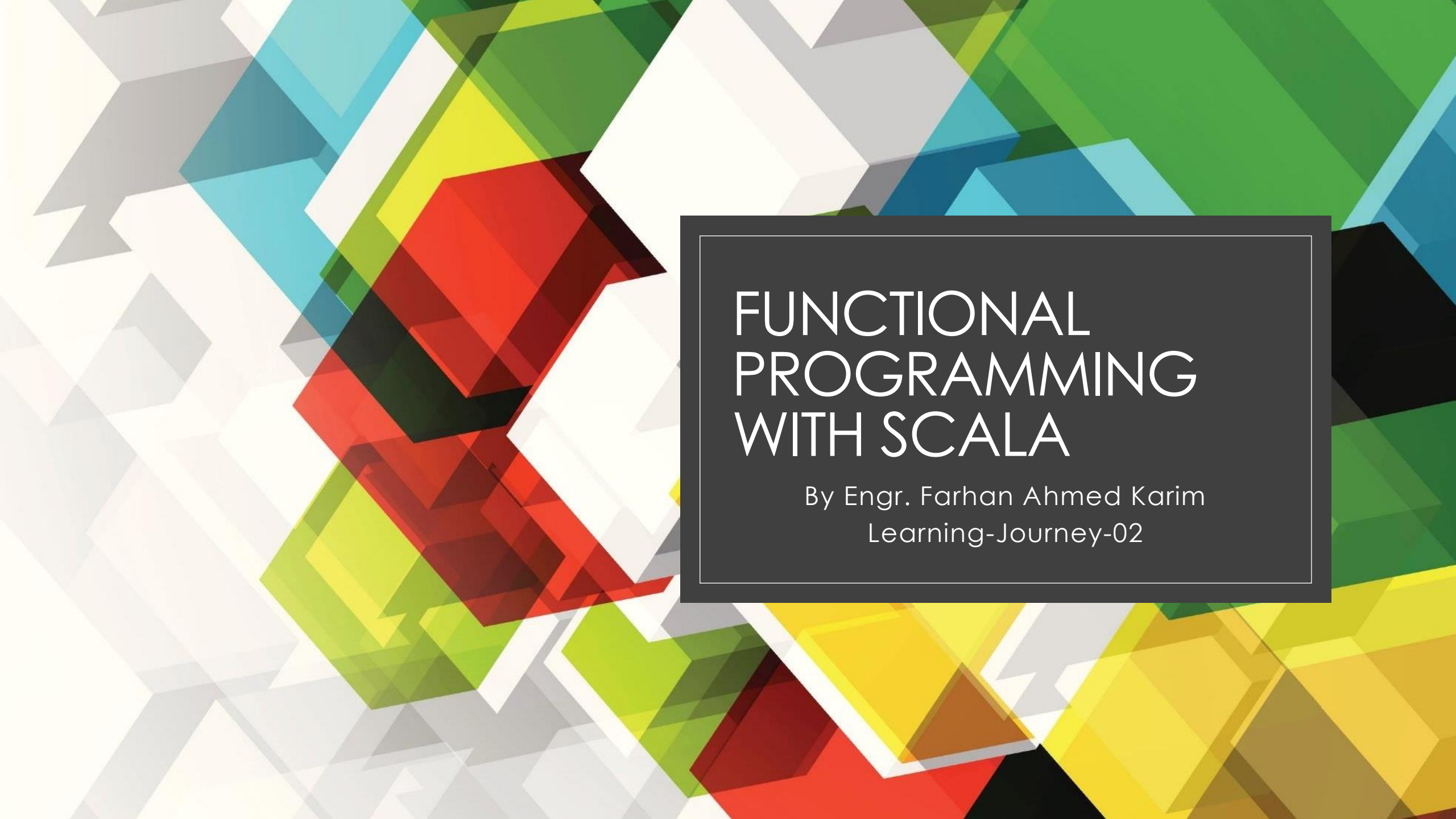

FUNCTIONAL PROGRAMMING WITH SCALA
By Engr. Farhan Ahmed Karim
Learning-Journey-02

Programming Paradigms

- Programming Paradigm is a approach to programming
- It will be based on set of principles or theory
- Different ways of thinking.
- Different types
  - Imperative
  - Functional

Imperative Programming

- Imperative programming is a programming paradigm that uses statements that change a program's state.
- Example :
  - C#
  - C++
  - Java

Functional Programming

- A programming style that models computation as the evaluation of expression.
- Example:
- ● Haskell
- ● Erlang
- ● Lisp
- ● Clojure

History of Functional Programming

- Mathematical abstraction Lambda Calculus was the foundation for functional Programming
- Lambda calculus was developed in 1930s to be used for functional definition, functional application and recursion.
- It states that any computation can be achieved by sending the input to some black box, which produces its result.
- This black box is lambda functions, and in FP it's functions.
- LISP was the first functional programming language. It was defined by McCarthy in 1958

Functional Programming

Imperative vs Functional Programming

Characteristics	Imperative	Functional
State changes	Important	Non-existent
Order of execution	Important	Low importance
Primary flow control	Loops, conditions and method (function) calls	Function calls
Primary manipulation unit	Instances of structures or classes	Functions

Functional Programming

Characteristics of FP

- Immutable data
- Lazy evaluation
- No side effects
- Referential transparency
- Functions are first class citizens

Functional Programming

Functional programming: An Introduction

- Functional programming (FP) is based on a simple premise with far-reaching implications: we construct our programs using only pure functions—in other words, functions that have no **side effects**. What are side effects? A function has a side effect if it does something other than simply return a result, for example:

- Modifying a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

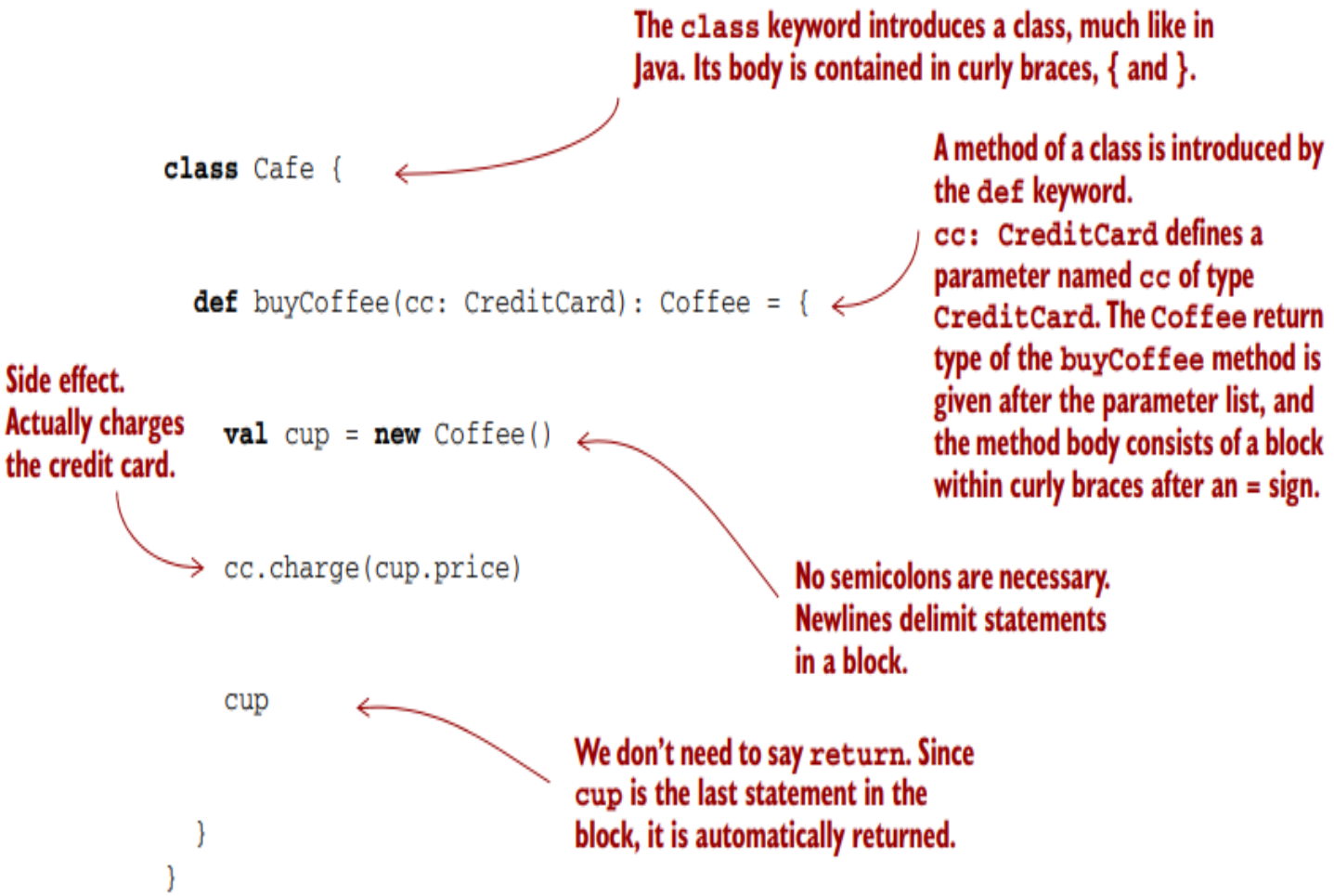
Question???

- What programming would be like without the ability to do these things, or with significant restrictions on when and how these actions can occur. It may be difficult to imagine.
- How is it even possible to write useful programs at all?
- If we can't reassign variables, how do we write simple programs like loops?
- What about working with data that changes, or handling errors without throwing exceptions? How
- Can we write programs that must perform I/O, like drawing to the screen or reading from a file?
- **The answer is** that functional programming is a restriction on how we write programs, but not on what programs we can express.

The benefits of FP: a simple example

- Let's look at an example that demonstrates some of the benefits of programming with pure functions.

- Note: Don't worry too much about following every detail about code . As long as you have a basic idea of what the code is doing, that's what's important.



A program with side effects

Suppose we're implementing a program to handle purchases at a coffee shop. We'll begin with a Scala program that uses side effects in its implementation (also called an impure program).

The benefits of FP: a simple example

- The line cc.charge(cup.price) is an example of a side effect. Charging a credit card involves some interaction with the outside world—suppose it requires contacting the credit card company via some web service, authorizing the transaction, charging the card, and (if successful) persisting some record of the transaction for later reference. But our function merely returns a Coffee and these other actions are happening on the side, hence the term "side effect."
- As a result of this side effect, the code is difficult to test. We don't want our tests to actually contact the credit card company and charge the card! This lack of testability is suggesting a design change:

Cntd.

- As a result of this side effect, the code is difficult to test. We don't want our tests to actually contact the credit card company and charge the card! This lack of testability is suggesting a design change: arguably, CreditCard shouldn't have any knowledge baked into it about how to contact the credit card company to actually execute a charge, nor should it have knowledge of how to persist a record of this charge in our internal systems. We can make the code more modular and testable by letting CreditCard be ignorant of these concerns and passing a Payments object into buyCoffee.

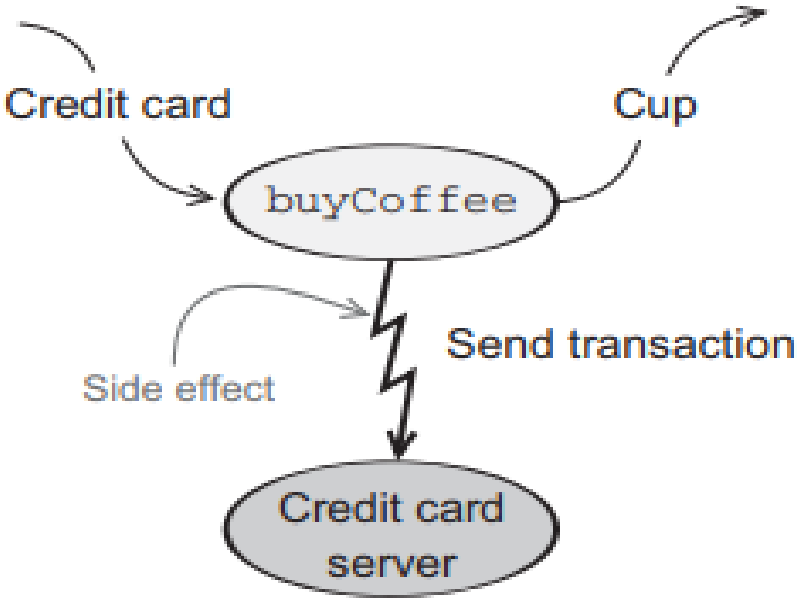
```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

Cntd.

- Though side effects still occur when we call p.charge(cc, cup.price), we have at least regained some testability.
- Payments can be an interface, and we can write a mock implementation of this interface that is suitable for testing. But that isn't ideal either. We're forced to make Payments an interface, when a concrete class may have been fine otherwise, and any mock implementation will be awkward to use.
- For example, it might contain some internal state that we'll have to inspect after the call to buyCoffee, and our test will have to make sure this state has been appropriately modified (mutated) by the call to charge.
- We can use a mock framework or similar to handle this detail for us, but this all feels like overkill if we just want to test that buyCoffee creates a charge equal to the price of a cup of coffee.

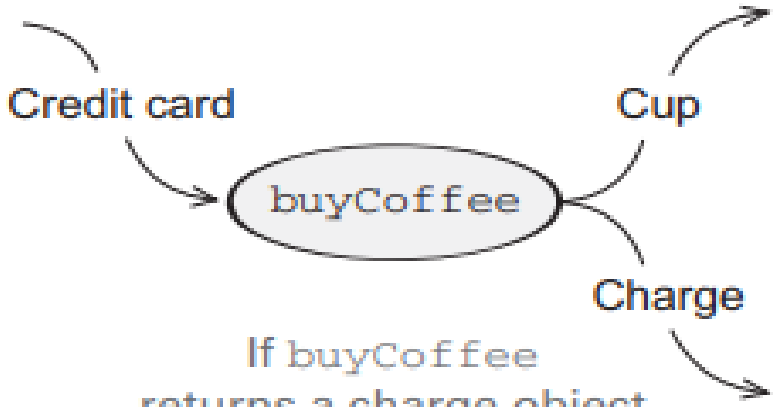
A call to buyCoffee

With a side effect



Can't test `buyCoffee` without credit card server. Can't combine two transactions into one.

Without a side effect



If `buyCoffee` returns a charge object instead of performing a side effect, a caller can easily combine several charges into one transaction. (and can easily test the `buyCoffee` function without needing a payment processor).

A functional solution: removing the side effects

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    (cup, Charge(cc, cup.price))
  }
}
```

buyCoffee now returns a pair of a Coffee and a Charge, indicated with the type (Coffee, Charge). Whatever system processes payments is not involved at all here.

To create a pair, we put the cup and Charge in parentheses separated by a comma.

The benefits of FP: a simple example

A case class has one primary constructor whose argument list comes after the class name (here, `Charge`). The parameters in this list become public, unmodifiable (immutable) fields of the `class` and can be accessed using the usual object-oriented dot notation, as in `other.cc`.

```
case class Charge(cc: CreditCard, amount: Double) {

  def combine(other: Charge): Charge =

    if (cc == other.cc)

      Charge(cc, amount + other.amount)

    else

      throw new Exception("Can't combine charges to different cards")

}
```

An `if` expression has the same syntax as in Java, but it also returns a value equal to the result of whichever branch is taken. If `cc == other.cc`, then `combine` will return `Charge(..)`; otherwise the exception in the `else` branch will be thrown.

A `case class` can be created without the keyword `new`. We just use the class name followed by the list of arguments for its primary constructor.

The syntax for throwing exceptions is the same as in Java and many other languages. We'll discuss more functional ways of handling error conditions in a later chapter.

Exactly what is a (pure) function?

- We said earlier that FP means programming with pure functions, and a pure function is one that lacks side effects. In our discussion of the coffee shop example, we worked off an informal notion of side effects and purity. Here we'll formalize this notion, to pinpoint more precisely what it means to program functionally. This will also give us additional insight into one of the benefits of functional programming: pure functions are easier to reason about.

Cntd.

- You should be familiar with a lot of pure functions already. Consider the addition (+) function on integers. It takes two integer values and returns an integer value. For any two given integer values, it will always return the same integer value. Another example is the length function of a String in Java, Scala, and many other languages where strings can't be modified (are immutable). For any given string, the same length is always returned and nothing else occurs.

Referential Transparency

- We can formalize this idea of pure functions using the concept of referential transparency (RT).
- This is a property of expressions in general and not just functions. For the purposes of our discussion, consider an expression to be any part of a program that can be evaluated to a result—anything that you could type into the Scala interpreter and get an answer.
- For example, 2 + 3 is an expression that applies the pure function + to the values 2 and 3 (which are also expressions). This has no side effect. The evaluation of this expression results in the same value 5 every time. In fact, if we saw 2 + 3 in a program we could simply replace it with the value 5 and it wouldn't change a thing about the meaning of our program

Cntd.

- This is all it means for an expression to be referentially transparent—in any program, the expression can be replaced by its result without changing the meaning of the program. And we say that a function is pure if calling it with RT arguments is also RT.

Referential transparency, purity, and the substitution model

- Let's look at two more examples—one where all expressions are RT and can be reasoned about using the substitution model, and one where some expressions violate RT. There's nothing complicated here; we're just formalizing something you likely already understand

```
scala> val x = "Hello, World"
x: java.lang.String = Hello, World

scala> val r1 = x.reverse
r1: String = dlroW ,olleH

scala> val r2 = x.reverse
r2: String = dlroW ,olleH
```

r1 and r2 are the same.

Suppose we replace all occurrences of the term x with the expression referenced by x (its *definition*), as follows:

```
scala> val r1 = "Hello, World".reverse
r1: String = dlroW ,olleH

scala> val r2 = "Hello, World".reverse
r2: String = dlroW ,olleH
```

r1 and r2 are still the same.

Cntd

- This transformation doesn't affect the outcome. The values of r1 and r2 are the same as before, so x was referentially transparent. What's more, r1 and r2 are referentially transparent as well, so if they appeared in some other part of a larger program, they could in turn be replaced with their values throughout and it would have no effect on the program.

- Now let's look at a function that is not referentially transparent. Consider the append function on the java.lang.StringBuilder class. This function operates on the StringBuilder in place. The previous state of the StringBuilder is destroyed after a call to append. Let's try this out:

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val y = x.append(", World")
y: java.lang.StringBuilder = Hello, World

scala> val r1 = y.toString
r1: java.lang.String = Hello, World

scala> val r2 = y.toString
r2: java.lang.String = Hello, World
```

r1 and r2 are the same.

Cntd.

- So far so good. Now let's see how this side effect breaks RT. Suppose we substitute the call to append like we did earlier, replacing all occurrences of y with the expression referenced by y:

```
scala> val x = new StringBuilder("Hello")
x: java.lang.StringBuilder = Hello

scala> val r1 = x.append(", World").toString
r1: java.lang.String = Hello, World

scala> val r2 = x.append(", World").toString
r2: java.lang.String = Hello, World, World
```

r1 and r2 are no longer the same.

This transformation of the program results in a different outcome. We therefore conclude that StringBuilder.append is not a pure function.

End of Lecture-01