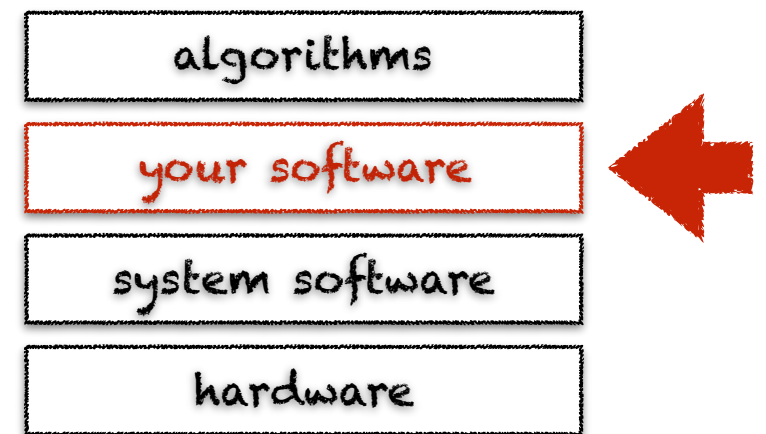




functional  
programming

# learning objectives



- ◆ learn the problem with side effects
- ◆ learn how functional programming solves it
- ◆ learn about high-order functions

# source of this lesson



## ***FUNCTIONAL PROGRAMMING IN SCALA***

***BY PAUL CHIUSANO &  
RÚNAR BJARNASON***

***MANNING, 2014***

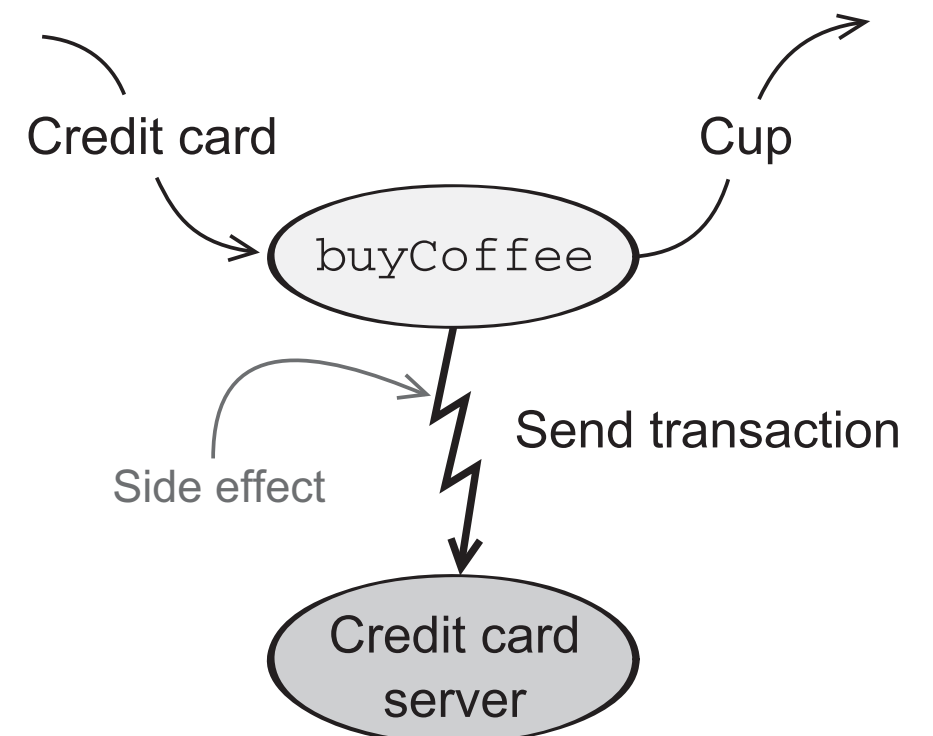
# side effects

a function has a **side effect** if it **modifies something outside itself**

charging a credit card modifies something outside buyCoffee, i.e., it requires **contacting the credit card company**

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

this makes it **difficult to test** function buyCoffee





solution

pure  
functions



# pure functions

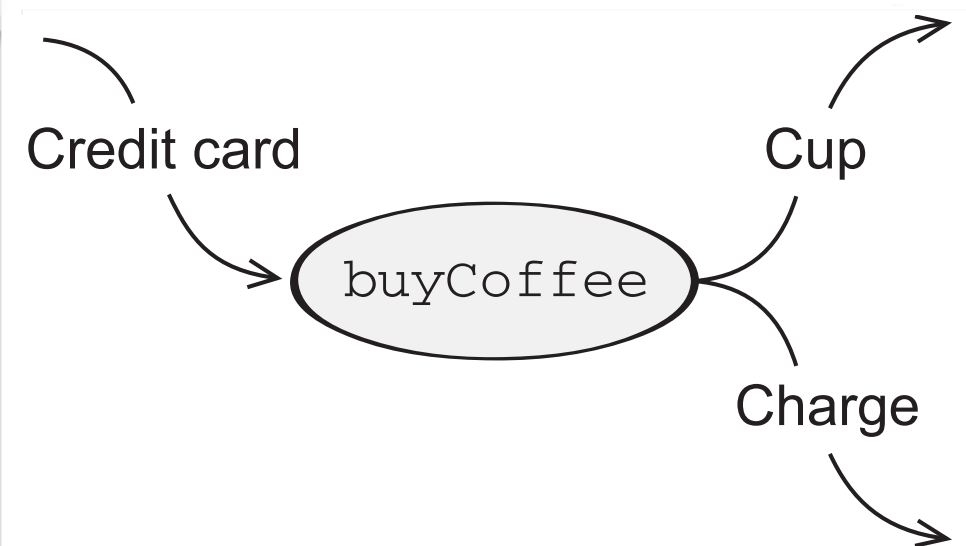
only return values and have no side effects

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

now buyCoffee simply returns a tuple consisting of a Coffee and a Charge

```
case class Charge(cc: CreditCard, amount: Double) {  
  def combine(other: Charge): Charge =  
    if (cc == other.cc)  
      return Charge(cc, amount + other.amount)  
    else throw new Exception("Charges on different credit cards cannot be combined")  
}
```

class Charge reifies the action of charging the credit card and returns it as an object



# referential integrity

an expression  $e$  is **referentially transparent** if, for all programs  $p$ , all occurrences of  $e$  in  $p$  can be **substituted by the result of evaluating  $e$  without affecting the semantics of  $p$**

a function  $f$  is pure if the expression  $f(x)$  is **referentially transparent** for all referentially transparent  $x$

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

```
...  
val coffee = buyCoffee(myCreditCard)  
...
```



```
...  
val coffee = new Coffee()  
...
```

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

```
...  
val (coffee, charge) = buyCoffee(myCreditCard)  
...
```



```
...  
var cup = new Coffee()  
val (coffee, charge) = (cup, Charge(myCreditCard, cup.price))  
...
```

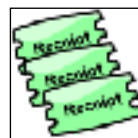


# functional reuse

pure functions contribute to code reuse  
because they can be easily composed

```
class Cafe {  
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = { ... }  
  
  def buyCoffeees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {  
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))  
    val (coffees, charges) = purchases.unzip  
    return (coffees, charges.reduce((c1, c2) => c1.combine(c2)))  
  }  
}
```

assuming  $n = 3$



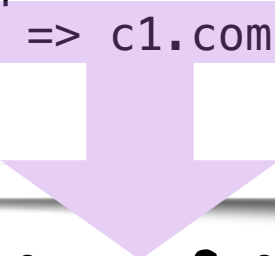
what is this?



# high-order functions

take a function as parameter or  
returns function as its results

```
class Cafe {  
  def buyCoffee(cc: CreditCard): Coffee = { ... }  
  
  def buyCoffeees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {  
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))  
    val (coffees, charges) = purchases.unzip  
    return (coffees, charges.reduce((c1,c2) => c1.combine(c2)))  
  }  
}
```



this is a parameter of type function  
(Charge, Charge) => Charge

high-order functions are also called functionals or functors

this concept comes from lambda calculus, a formal system  
in mathematical logic for expressing computation

# back to matrices

```
class DenseMatrix(private val rows: Int, private val cols: Int) extends AbstractMatrix {  
    ...  
  
    private val matrix = Array.ofDim[Int](rows, cols)  
  
    def this(rows: Int, cols: Int, init: (Int,Int) => Int) {  
        this(rows,cols)  
        for (i <- 0 to rows - 1; j <- 0 to cols - 1)  
            this.matrix(i)(j) = init(i,j)  
        }  
    ...  
}
```

```
var m = new DenseMatrix(3, 3, (i,j) => i + j)  
m.print
```

DenseMatrix

	0	1	2	
	1	2	3	
	2	3	4	

```
m = new DenseMatrix(3, 3, (i,j) => (i + 1) * 2)  
m.print
```

DenseMatrix

	2	2	2	
	4	4	4	
	6	6	6	

```
m = new DenseMatrix(3, 3, (i,j) => (j + 1) * 2)  
m.print
```

DenseMatrix

	2	4	6	
	2	4	6	
	2	4	6	

```
m = new DenseMatrix(3, 3, (i,j) => 7)  
m.print
```

DenseMatrix

	7	7	7	
	7	7	7	
	7	7	7	

```
m = new DenseMatrix(3, 3, (i,j) => scala.util.Random.nextInt(5))  
m.print
```

DenseMatrix

	1	3	3	
	2	2	3	
	2	1	2	