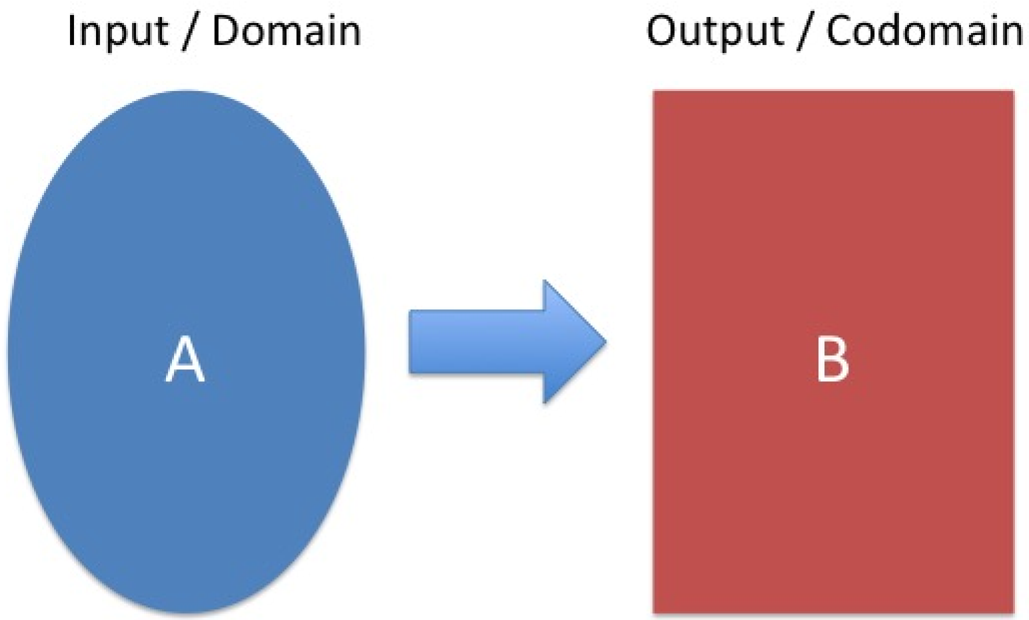
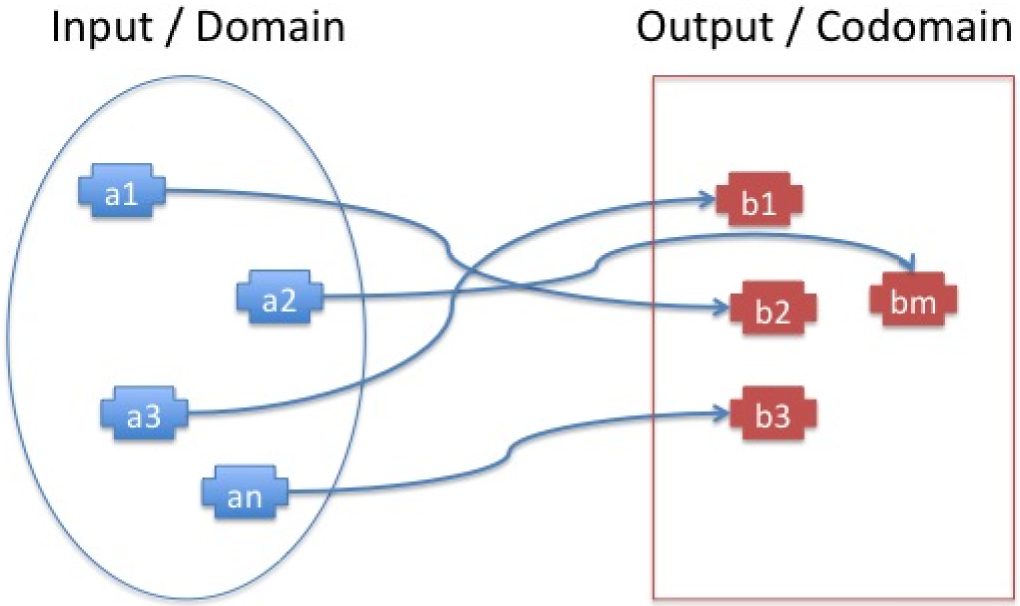
FOUNDATION

Function

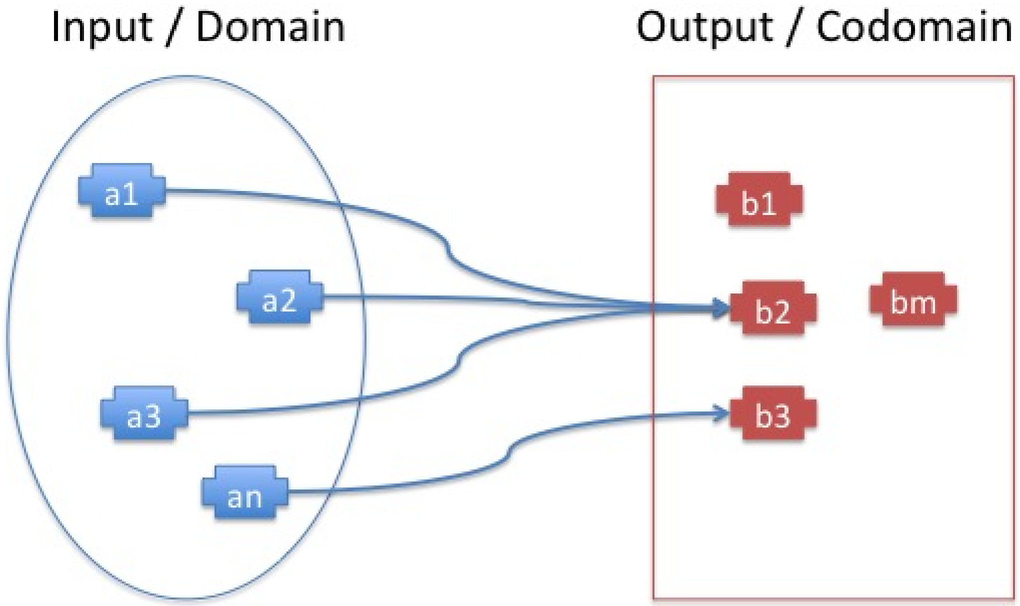
Function



Function is a mapping



Function is a mapping



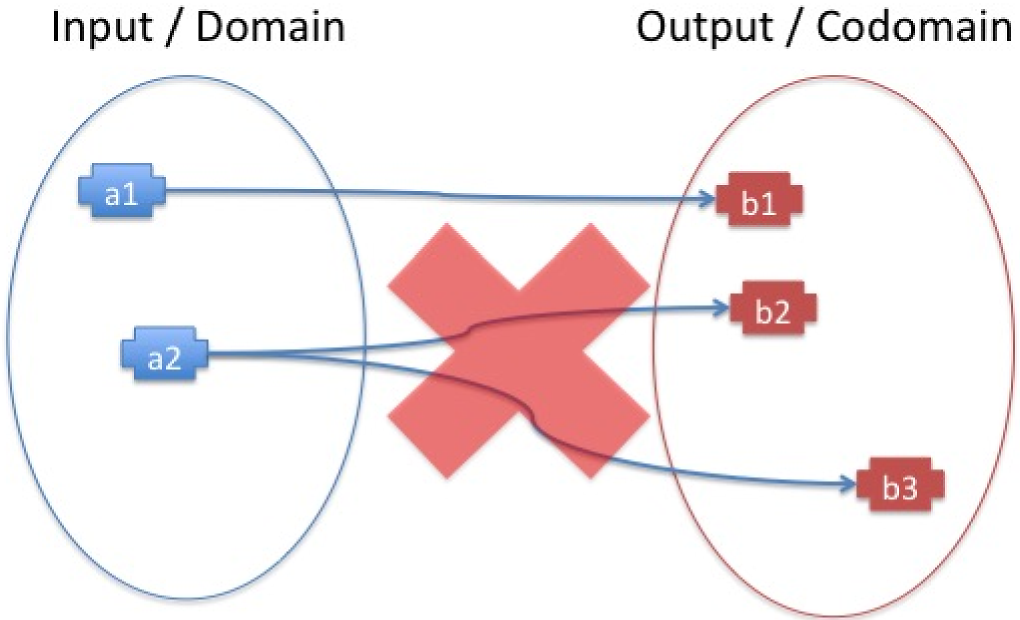
Programming function

!=

Maths function

Pure function

Nondeterministic

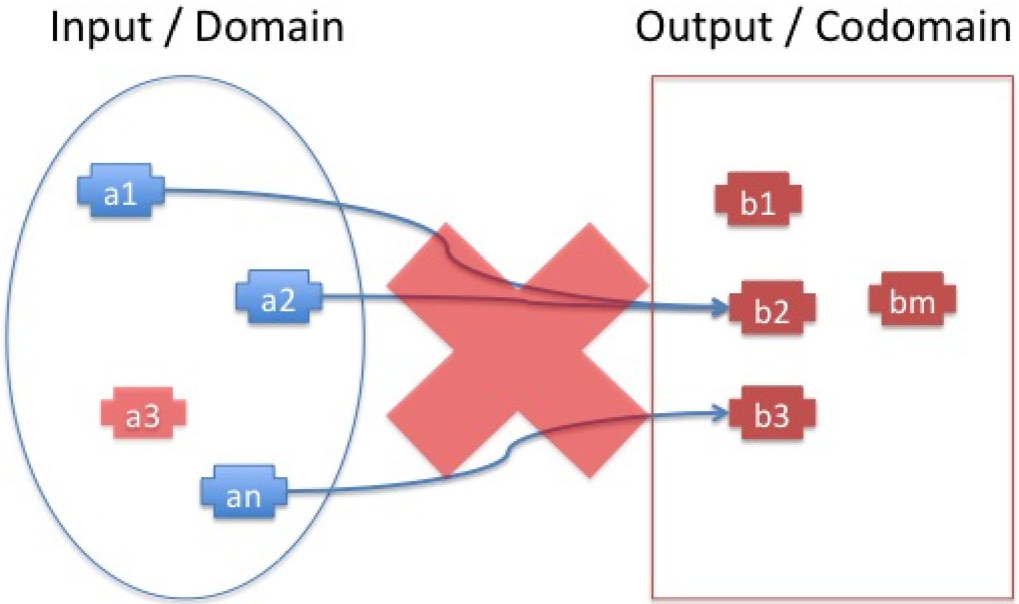


Nondeterministic

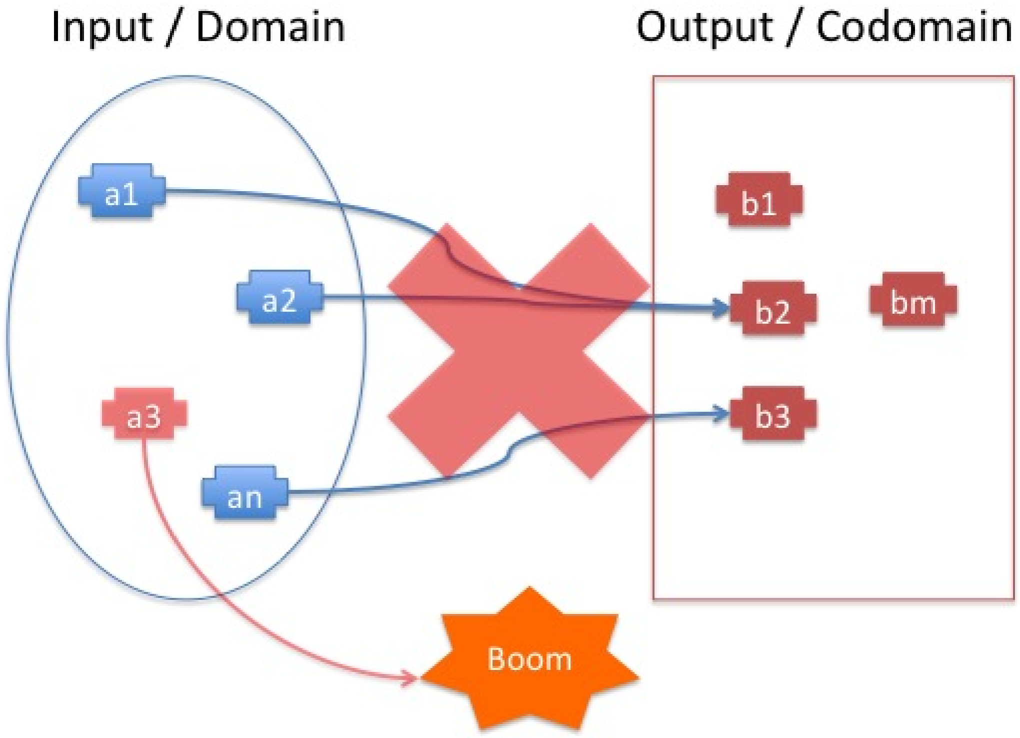
```scala
import scala.util.Random
```

```
scala> Random.nextInt(100)
res0: Int = 63

scala> Random.nextInt(100)
res1: Int = 30

scala> Random.nextInt(100)
res2: Int = 28
```

Partial function



Partial function



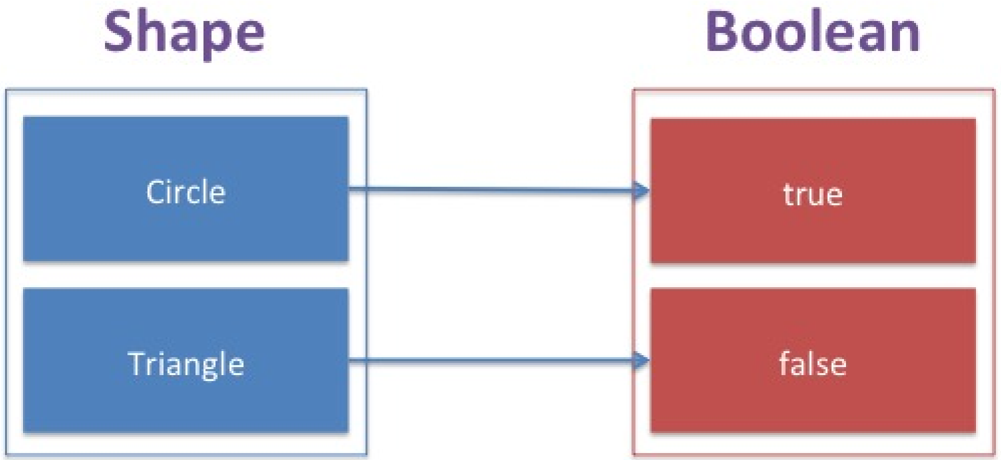
Partial function

```scala
def head(xs: List[Int]): Int = xs match {
  case Nil    => sys.error("Can't access head of empty List")
  case x :: _ => x
}
```

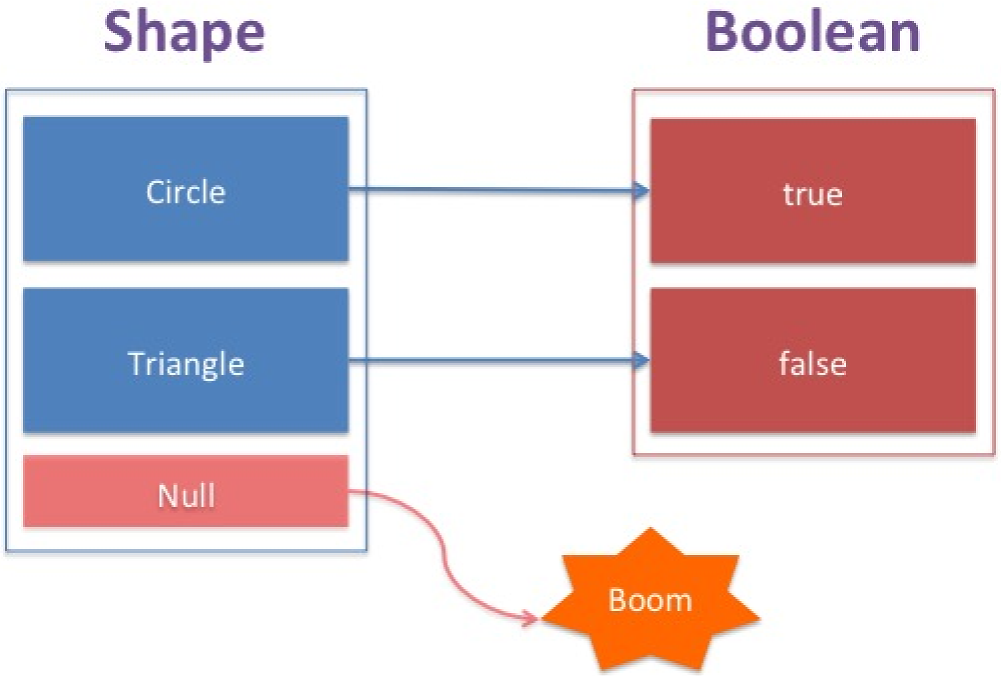
```
scala> head(List(1,2,3))
res3: Int = 1
```

```
scala> head(Nil)
java.lang.RuntimeException: Can't access head of empty List
  at scala.sys.package$.error(package.scala:30)
  at .head(<console>:14)
  ... 43 elided
```

Null



Null



Null

```scala
sealed trait Shape

case class Circle(radius: Int) extends Shape
case class Rectangle(width: Int, height: Int) extends Shape

def isCircle(x: Shape): Boolean = x match {
  case Circle(_)       => true
  case Rectangle(_, _) => false
}
```

```
scala> isCircle(Circle(5))
res5: Boolean = true

scala> isCircle(Rectangle(5, 10))
res6: Boolean = false
```

```
scala> isCircle(null)
scala.MatchError: null
  at .isCircle(<console>:18)
  ... 43 elided
```

Reflection

```scala
def foo[A](a: A): A = a match {
  case x: Int    => (x + 1).asInstanceOf[A]
  case x: String => x.reverse.asInstanceOf[A]
  case _         => a
}
```

```
scala> foo(5)
res8: Int = 6

scala> foo("Hello")
res9: String = olleH

scala> foo(true)
res10: Boolean = true
```

Reflection

```scala
def foo[A](a: A): Int = a match {
  case _: List[Int]    => 0
  case _: List[String] => 1
  case _               => 2
}
```

```
scala> foo(List(1,2,3))
res11: Int = 0

scala> foo(List("abc"))
res12: Int = 0
```

Without Reflection

```
def foo[A](a: A): A = ???
```

Without Reflection

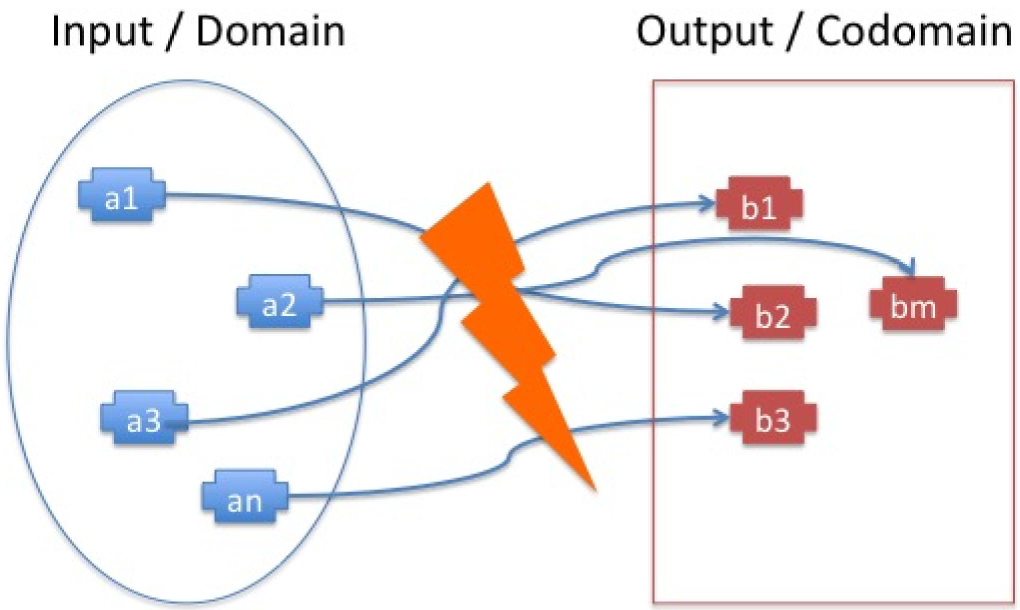
```
def foo[A](a: A): A = a
```

Reflection

```scala
def bar(a: Any): Any = a match {
  case x: Int    => x + 1
  case x: String => x.reverse
  case _         => a
}
```

```scala
trait MyInterface
case class Impl1(value: Int) extends MyInterface
case class Impl2(value: String) extends MyInterface

def bar(a: MyInterface): Int = a match {
  case x: Impl1 => x.value + 1
  case x: Impl2 => x.value.size
  case _        => 0
}
```

Side effect



Side effect

```scala
def println(message: String): Unit = ...
```

Side effect

```scala
def println(message: String): Unit = ...
```

```
scala> val x = println("Hello")
Hello
x: Unit = ()
```

Side effect

```
scala> scala.io.Source.fromURL("http://google.com").take(100).mkString
res13: String = <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><meta content
```

Side effect

```scala
var x: Int = 0

def count(): Int = {
  x = x + 1
  x
}
```

```
scala> count()
res14: Int = 1

scala> count()
res15: Int = 2

scala> count()
res16: Int = 3
```

Pure Function

- deterministic (not nondeterministic)
- total (not partial)
- no mutation
- no exception
- no null
- no reflection
- no side effect

Scalazzi subset

- deterministic (not nondeterministic)
- total (not partial)
- no mutation
- no exception
- no null
- no reflection
- no side effect

Exercises

Why pure function?

Referential transparency

Referential transparency

A function is referentially transparent if we can replace all the calls of the function by its output without changing the behaviour of the program.

Referential transparency

A function is referentially transparent if we can replace all the calls of the function by its output without changing the behaviour of the program.

```
bar(foo(42), foo(42))
```

```
val x = foo(42)
bar(x, x)
```

Referentially transparent

```scala
var countCall = 0

def isEven(x: Int): Boolean = {
  countCall += 1
  x % 2 == 0
}
```

Referentially transparent

```scala
var countCall = 0

def isEven(x: Int): Boolean = {
  countCall += 1
  x % 2 == 0
}
```

```scala
scala> val five = isEven(5)
five: Boolean = false

scala> val six  = isEven(6)
six: Boolean = true

scala> countCall
res17: Int = 2
```

Referentially transparent

```scala
var countCall = 0

def isEven(x: Int): Boolean = {
  countCall += 1
  x % 2 == 0
}
```

```scala
scala> val five = isEven(5)
five: Boolean = false

scala> val six  = isEven(6)
six: Boolean = true

scala> countCall
res17: Int = 2
```

```scala
forAll((x: Int) => !isEven(x + 1) == isEvent(x))
```

Referentially transparent

```scala
var countCall = 0

def isEven(x: Int): Boolean = {
  countCall += 1
  x % 2 == 0
}
```

```scala
scala> val five = isEven(5)
five: Boolean = false

scala> val six  = !isEven(5) // isEven(6)
six: Boolean = true

scala> countCall
res18: Int = 2
```

```scala
forAll((x: Int) => !isEven(x + 1) == isEvent(x))
```

Referentially transparent

```scala
var countCall = 0

def isEven(x: Int): Boolean = {
  countCall += 1
  x % 2 == 0
}
```

```scala
scala> val five = isEven(5)
five: Boolean = false

scala> val six  = !five // isEven(6)
six: Boolean = true

scala> countCall
res19: Int = 1
```

```scala
forAll((x: Int) => !isEven(x + 1) == isEvent(x))
```

Referentially transparent

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def doSomethingExpensive(x: Int): Future[Int] =
  Future { ??? }

for {
  x <- doSomethingExpensive(5)
  y <- doSomethingExpensive(8) // one after the other
} yield x + y
```

Referentially transparent

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def doSomethingExpensive(x: Int): Future[Int] =
  Future { ??? }

for {
  x <- doSomethingExpensive(5)
  y <- doSomethingExpensive(8) // one after the other
} yield x + y
```

```scala
val fx = doSomethingExpensive(5)
val fy = doSomethingExpensive(8) // in parallel

for {
  x <- fx
  y <- fy
} yield x + y
```

Referentially transparent

```scala
def confirm(id: OrderId): Order = {
  val order  = getOrder(id)
  val status = updateStatus(id, Confirmed)
  val log    = audit.info(s"Order $id confirmed")
  val user   = getUser(order.userId)
  order
}
```

Referentially transparent

```
def confirm(id: OrderId): Order = {
  val order  = getOrder(id)
  val status = updateStatus(id, Confirmed)
  val log    = audit.info(s"Order $id confirmed")
  // val user   = getUser(order.userId)
  order
}
```

Referentially transparent

```
def confirm(id: OrderId): Order = {
  val order  = getOrder(id)
  // val status = updateStatus(id, Confirmed)
  // val log    = audit.info(s"Order $id confirmed")
  // val user   = getUser(order.userId)
  order
}
```

Referentially transparent

means

local reasoning

Referentially transparent

means

fearless refactoring

Testing

```scala
var counter: Long = 1

def foo(x: Int, b: Boolean): Long = {
  counter += x
  if(b) counter *= 2
  else counter = 0
  counter
}
```

Testing

```scala
var counter: Long = 1

def foo(x: Int, b: Boolean): Long = {
  counter += x
  if(b) counter *= 2
  else counter = 0
  counter
}
```

```
scala> foo(5,  true) == 12L
res22: Boolean = true

scala> foo(5, false) == 0L
res23: Boolean = true

scala> foo(5,  true) == 10L
res24: Boolean = true
```

Testing

```
def foo(x: Int, b: Boolean): Long = ???
```

Testing

```scala
def foo(x: Int, b: Boolean): Long = ???
```

```scala
foo(5,  true) == 12L

foo(5, false) == 0L
```

Caching

```scala
def expensiveFunc(x: Long): Boolean = ???
```

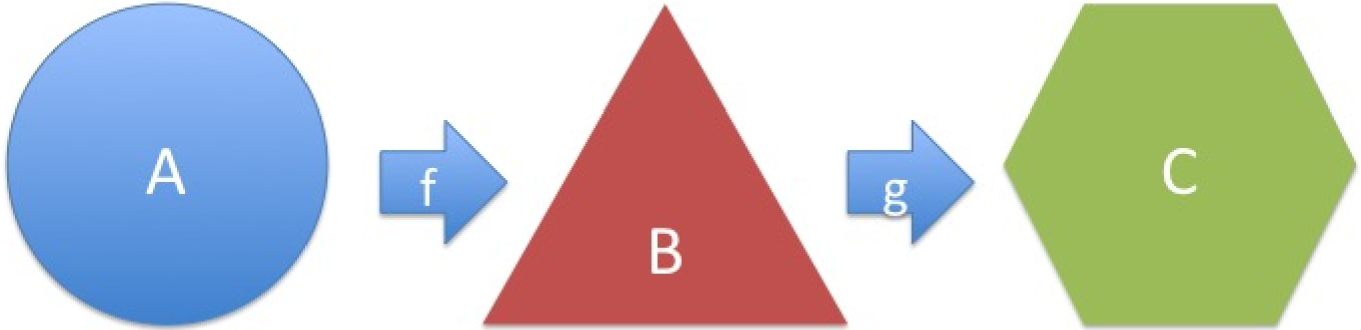
Caching

```scala
def expensiveFunc(x: Long): Boolean = ???
```

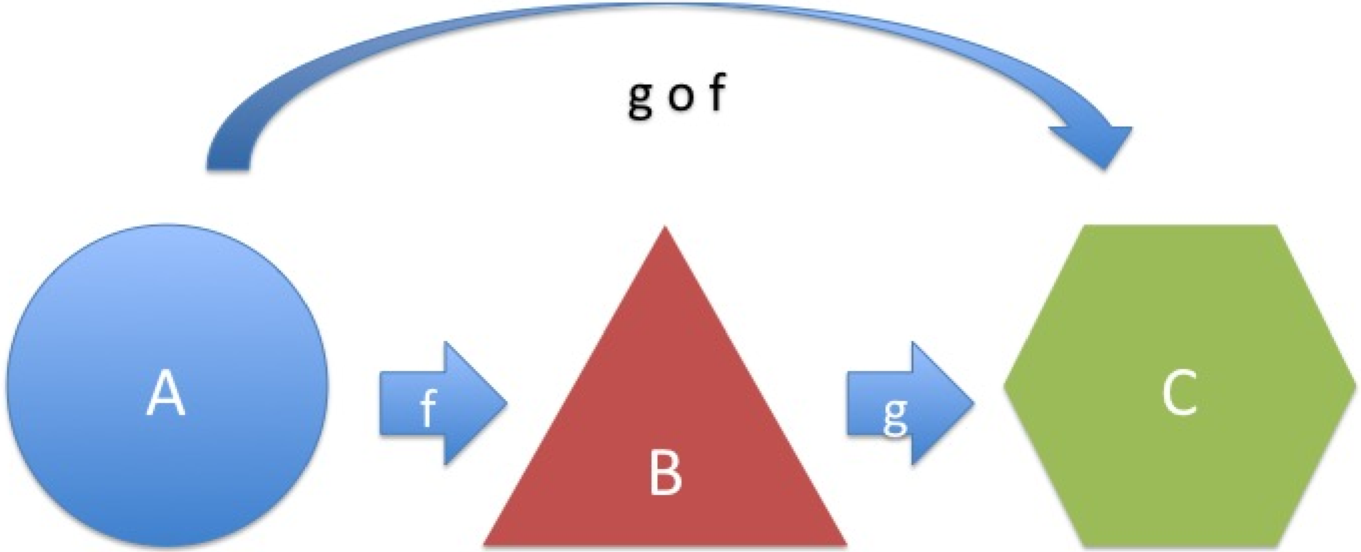
```scala
def memoize[A, B](f: A => B): A => B = ???
```

```scala
val cachedExpensiveFunc: Long => Boolean =
  memoize(expensiveFunc)
```

Function composition

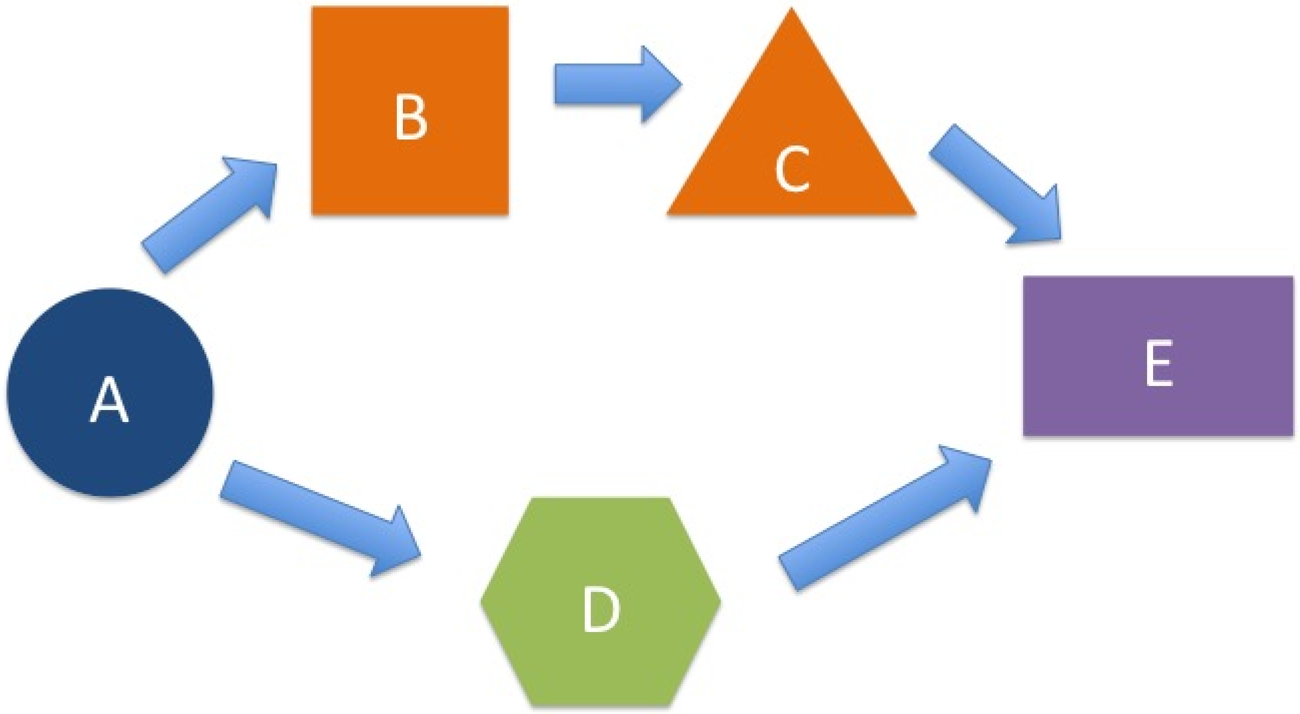


Function composition



Function composition

Graph



Downsides of pure function

- We can't **DO** anything

Downsides of pure function

- We can't **DO** anything
- We have to re-learn almost everything e.g. error handling, state, data structure

Downsides of pure function

- We can't **DO** anything
- We have to re-learn almost everything e.g. error handling, state, data structure
- Some things used to be easy e.g. logging

A => F[B]

Module 2: Type