SPARK with Scala

Module 1 begins

What is Scala?

- Scala is an acronym for “**Scalable Language”**. This means that Scala grows with you. You can play with it by typing one-line expressions and observing the results.

- But you can also rely on it for large mission critical systems, as many companies, including Twitter, LinkedIn, or Intel do with advent of frameworks like PLAY, AKKA etc.

- It can act both as a scripting language as well as a regular objected oriented language like java.

- Scala is a **pure-bred object-oriented language**. Conceptually, **every value is an object** and **every operation is a method-call**. The language supports advanced component architectures through **classes** and **traits.**

- Many traditional design patterns in other languages are already natively supported.
  - Examples being: Factory Patterns
    - like Companion Objects, Singleton Objects etc…
    - Lazy Initialisations etc.

- Even though its syntax is fairly conventional, Scala is also a **full-blown functional language**. It has everything you would expect, including first-class functions, a library **with efficient immutable data structures**, and **a general preference of immutability over mutation**.

- Scala **runs on the JVM**. **Java and Scala classes can be freely mixed**, no matter whether they reside in different projects or in the same.

Introduction to Scala for Spark

- Scala is a language which
  - Type Inferential
  - STATIC TYPING
  - TYPE SAFE: Mostly Throws errors at the time of compilation rather at run time unlike java
- others like:
  - PATTERN MATCHING,
  - CLEAN SYNTAX and API,
  - CURRYING,
  - PARTIAL FUNCTIONS and much more....
  - Tough to reach the ceiling: things become obscure for a programmer.

-gatives:

- --> SCALA is difficult, analogically it is similar to carrying the luggage on your shoudlers and climb, or learn to operate a crane which would do this job easier, but u need to invest time to learn operating crane which for some might be a difficult exercise. But this is worth an investment.

Functional Programming in Scala

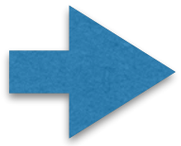
```
f2(f3:Int  => Int,x:Int):Int {f3(x+1}
```

```
f1(y:Int) => Int = {y*y}

printlf(f2(f1,4))
```

Function passed to a function

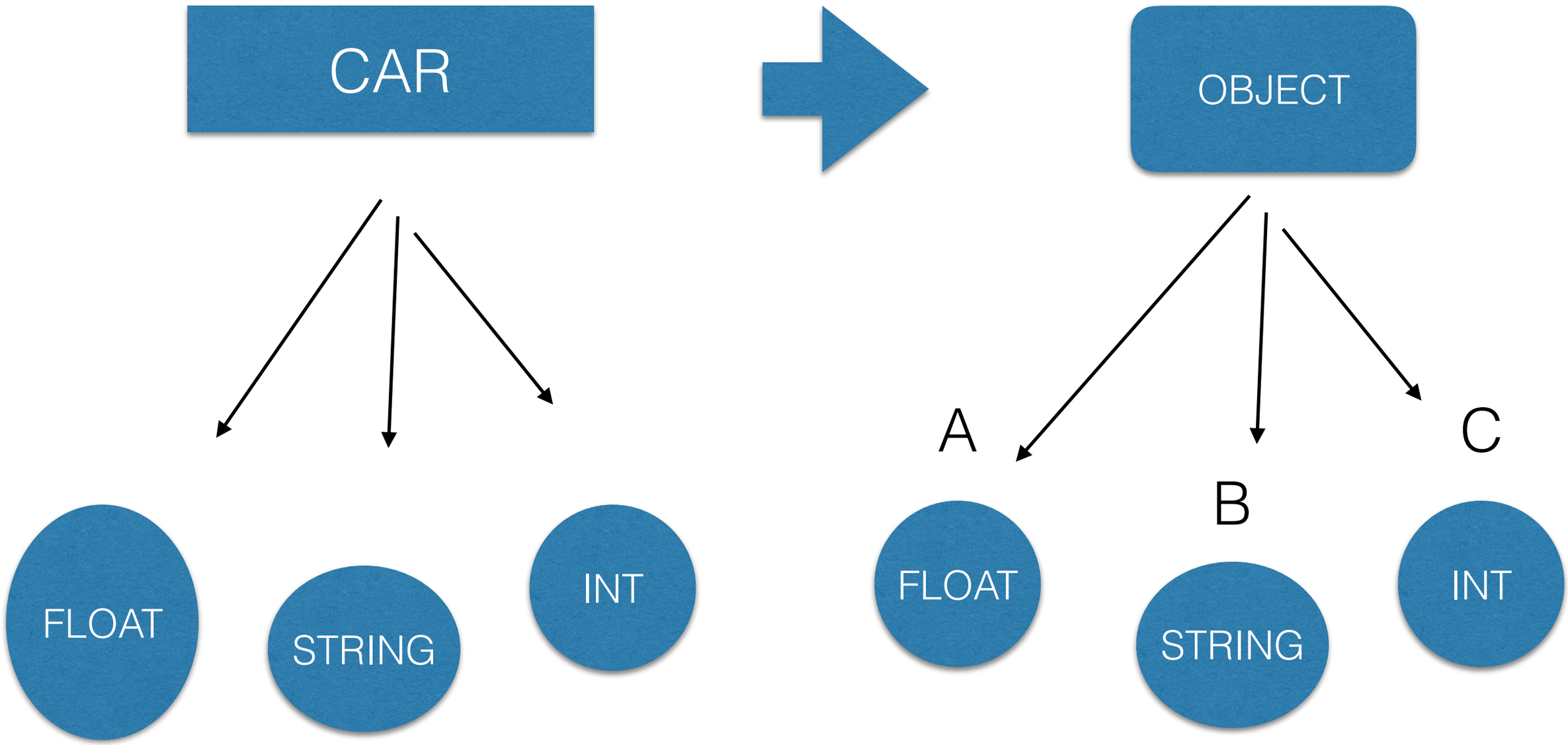
```
val a = Array(1,2,3,4,5)
a.Map(x => x +1))
```



```
scala> a.map(x => x + 1)
res2: Array[Int] = Array(2, 3, 4, 5, 6)
```

Lambda Function

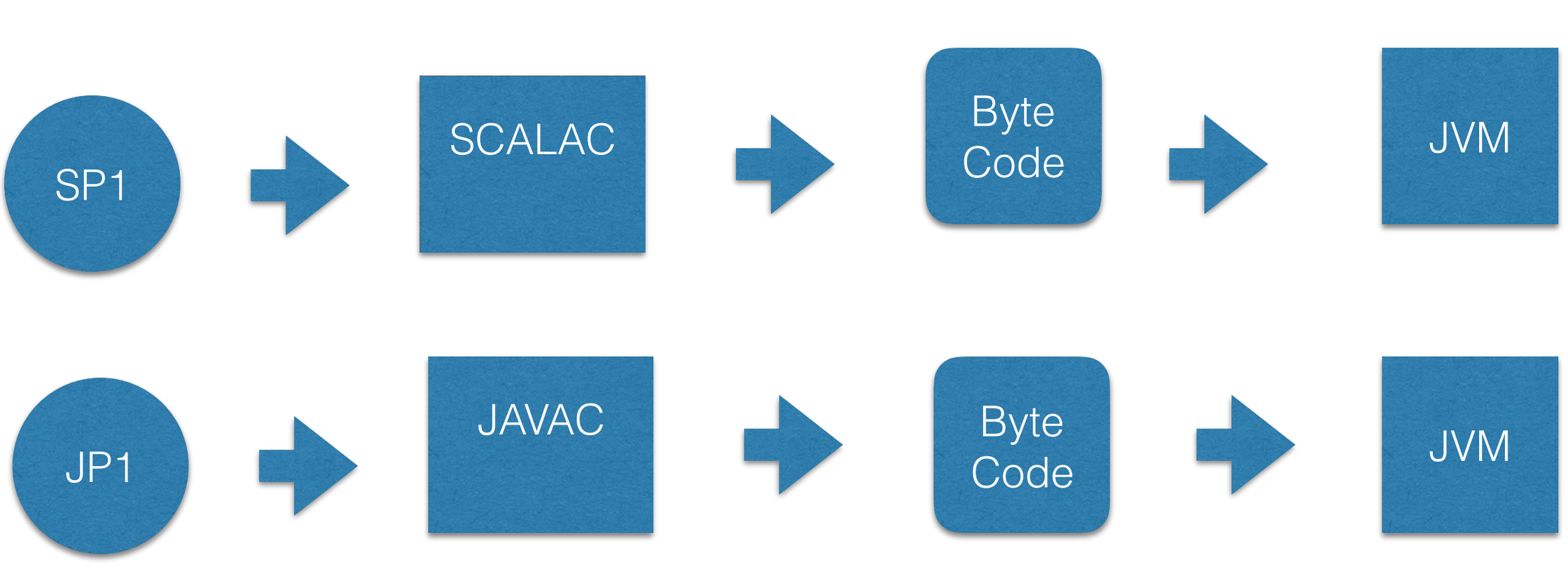
JAVA8 is Functional
Python is Functional - lambdas
SCALA


CAR
OBJECT
FLOAT
STRING
INT
A
FLOAT
B
STRING
C
INT

SCALA is Inherently JAVA
SCALAC is the scala compiler
SCALAC produces BYTE CODE
and SCALA Programs run in JVM

Similar to JAVAC

RUNTIME



Class1

F1(x:Int):Int = x + 1

Obj1

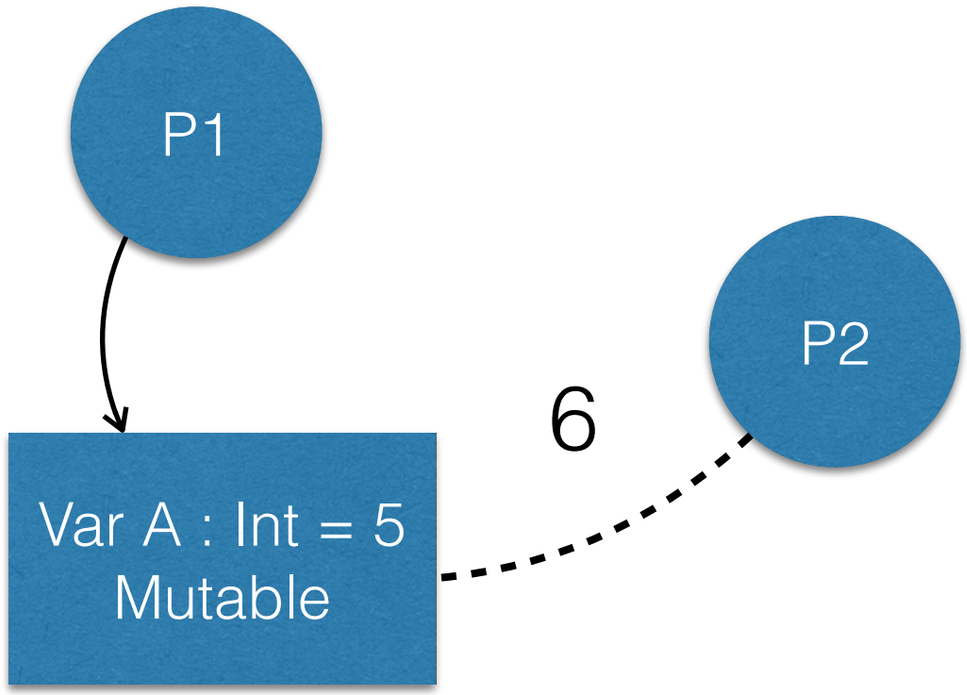
Obj1.F1(5)

def F1(x:Int):Int = x + 1

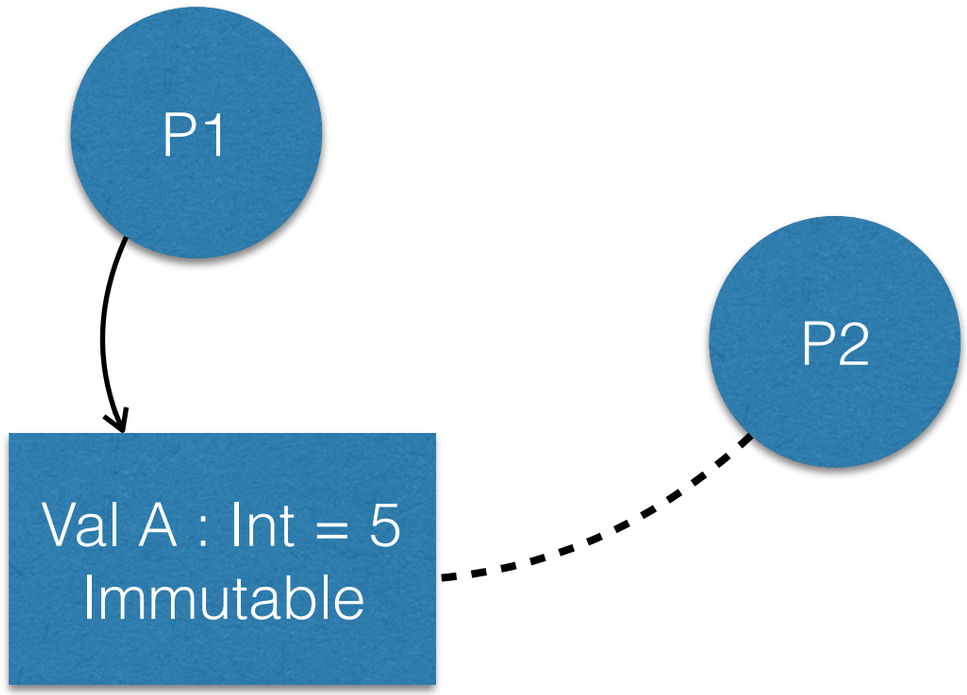
F1(5)

SCALAC - Compiler

Converts human readable code into Byte Code

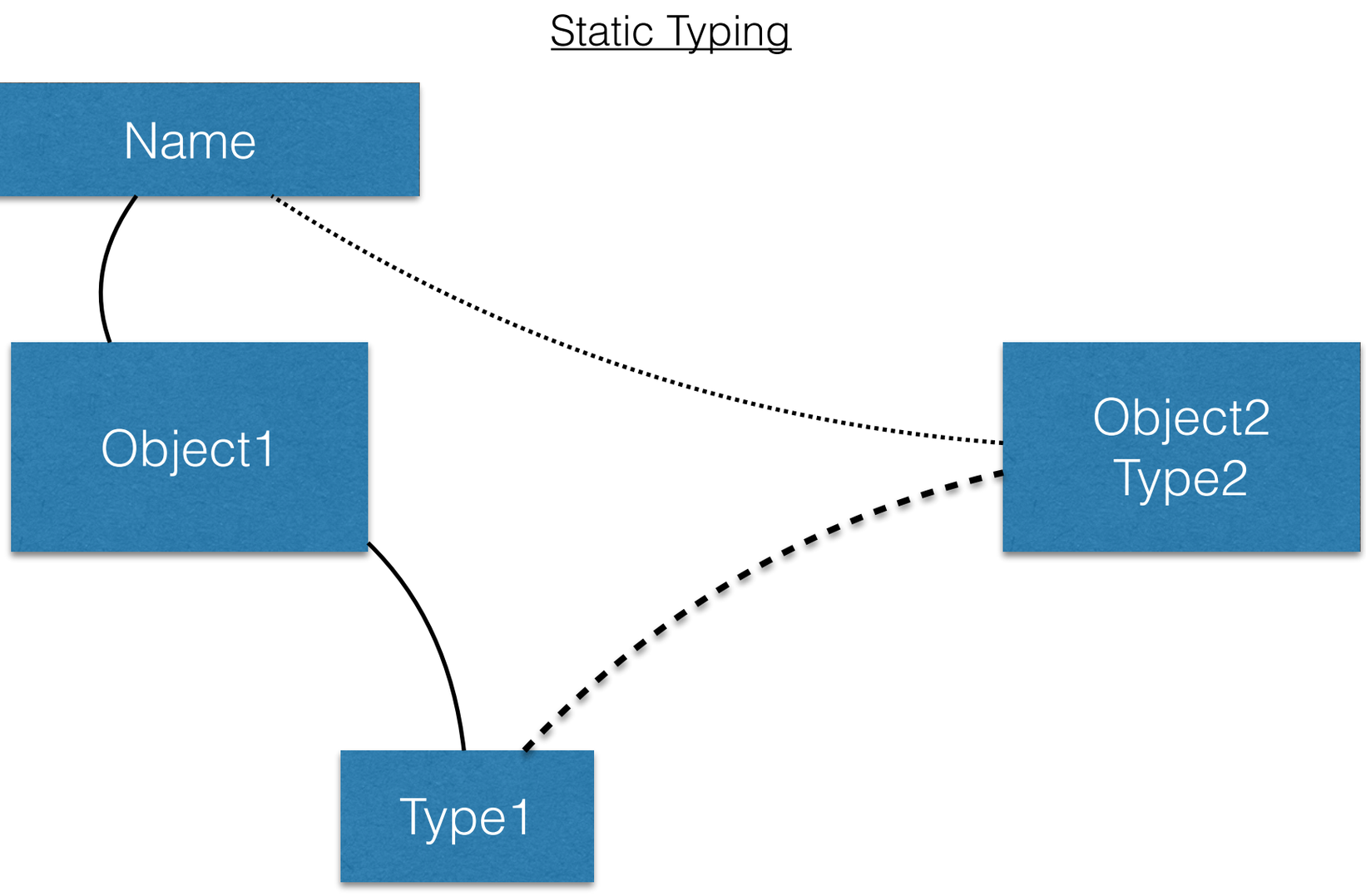


Scala prefers Immutability

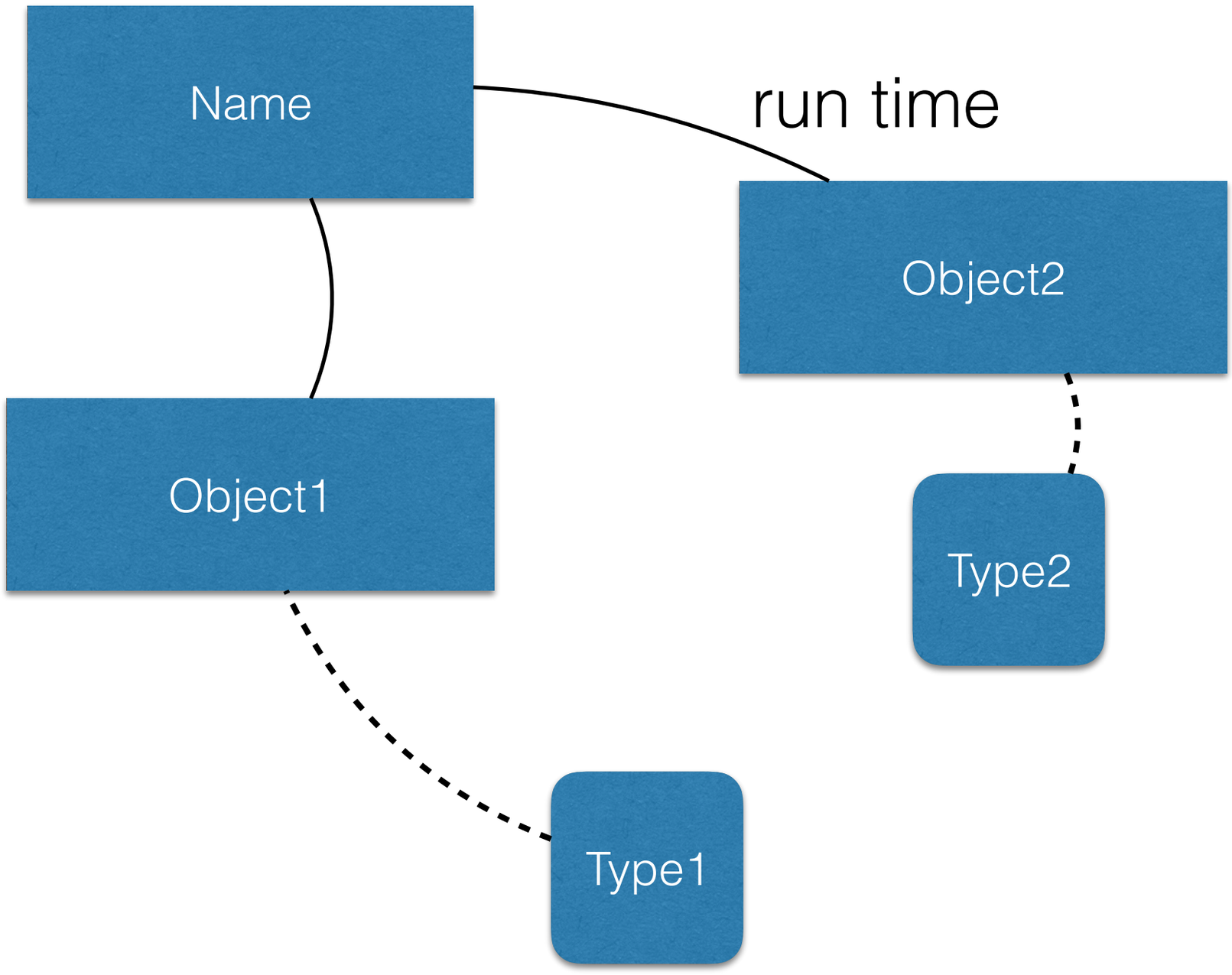


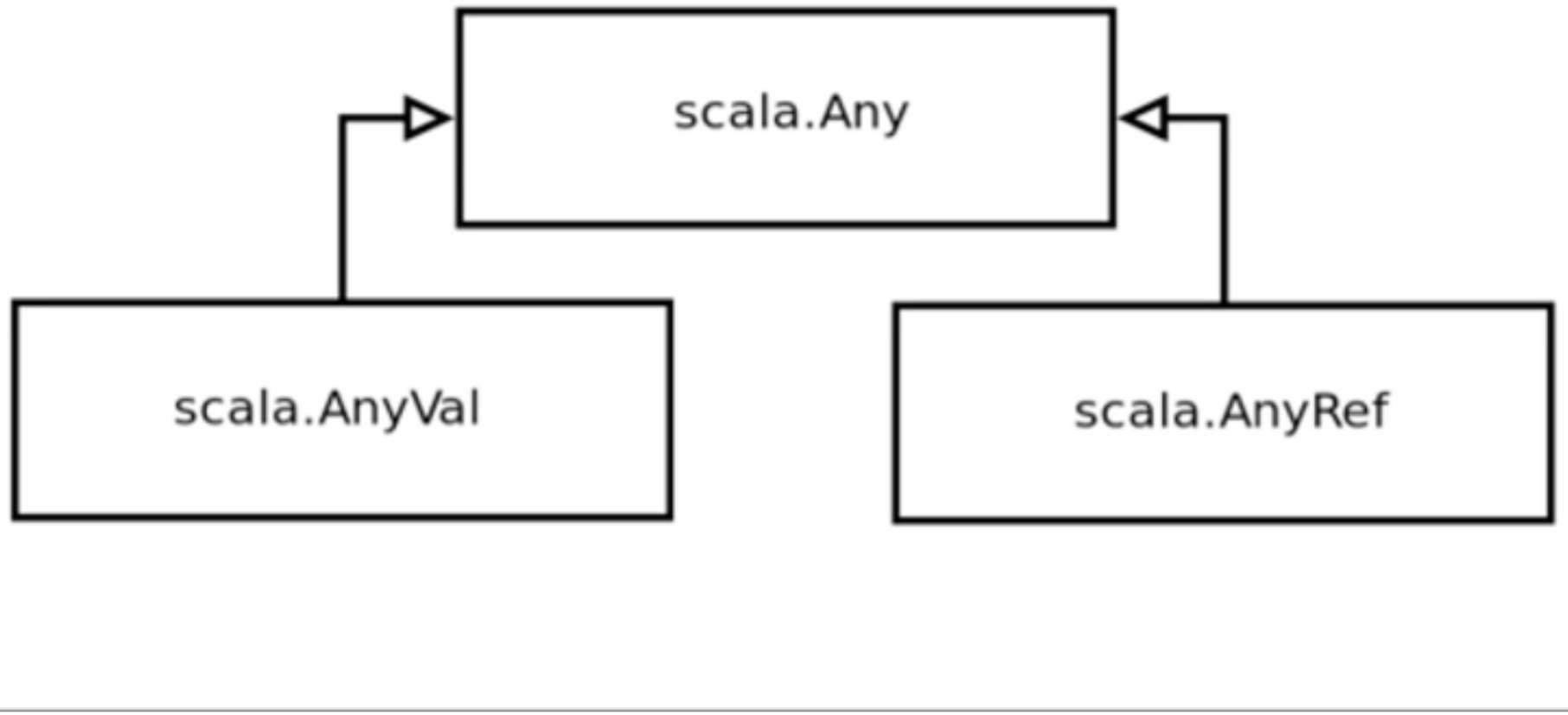
Types

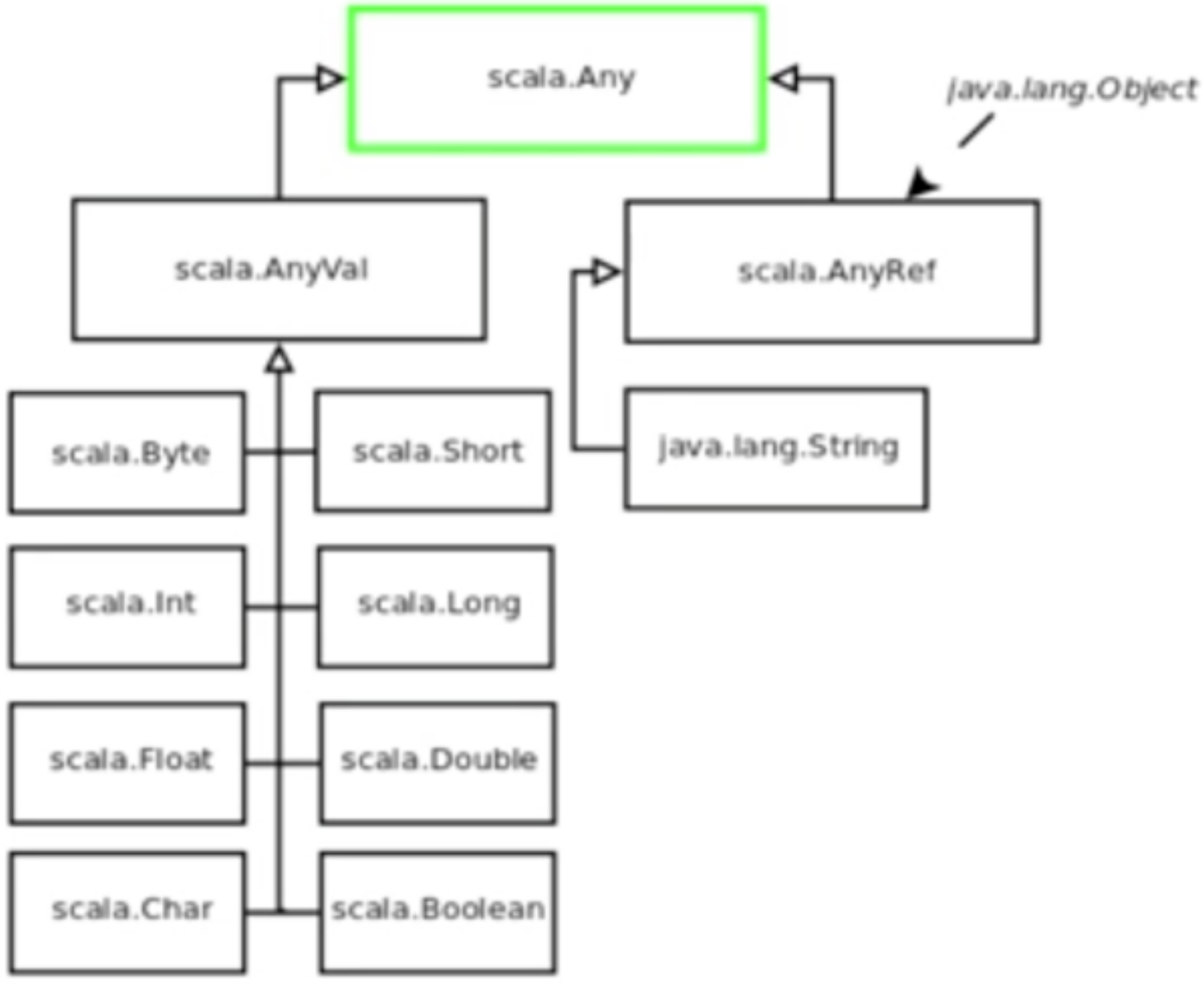
- Python, Ruby and Javascript - Dynamically Typed
- Scala, Java, C, C++ etc - Statically Typed



Dynamic Typed




scala.Any
scala.AnyVal
scala.AnyRef


scala.Any
java.lang.Object
scala.AnyVal
scala.AnyRef
scala.Byte
scala.Short
java.lang.String
scala.Int
scala.Long
scala.Float
scala.Double
scala.Char
scala.Boolean

VOID

UNIT

Type Inference

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def add(x:Int, y:Int) = {
   if (x > 10) (x+y).toString
   else x + y
}


// Exiting paste mode, now interpreting.

add: (x: Int, y: Int)Any

scala>
```

Type Inference - Conclusion

CONCLUSION

- Types returned from a method are inferred
- Type inferencer will make the best judgment
- If types are different it will find a common ancestor

Val vs Vars

Lazy Vals

lazy val a = {println("evaluated");10 + pc}; var pc = 79

```
scala> lazy val a = {println("evaluated");10 + pc}; var pc = 79
a: Int = <lazy>
pc: Int = 79
```

```
scala> lazy val quotient = 40/divisor
quotient: Int = <lazy>

scala> println(quotient)
java.lang.ArithmeticException: / by zero
  at .quotient$lzycompute(<console>:12)
  at .quotient(<console>:12)
  ... 29 elided

scala> divisor = 2
divisor: Int = 2

scala> println(quotient)
20
```

Lazy Vals

CONCLUSION

- `lazy val` will not evaluated until referenced
- Any subsequent calls to the `val` will return the same value when initially called upon
- There is no such thing as a `lazy var`
- `lazy val` can be forgiving if an exception happens

For Comprehension with Yield

- Consider a FOR COMPREHENSION with yield as a FOR LOOP WITH BUFFER which results the same collection as input but with processed values as provided with YIELD.

```
scala> for (i <- 1 to 5) yield i * 2
res11: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8
```

```
scala> val a = Array(1, 2, 3, 4, 5)
a: Array[Int] = Array(1, 2, 3, 4, 5)

scala> for (e <- a) yield e
res5: Array[Int] = Array(1, 2, 3, 4, 5)

scala> for (e <- a) yield e * 2
res6: Array[Int] = Array(2, 4, 6, 8, 10)

scala> for (e <- a) yield e % 2
res7: Array[Int] = Array(1, 0, 1, 0, 1)
```

```
scala> val a = Array(1, 2, 3, 4, 5)
a: Array[Int] = Array(1, 2, 3, 4, 5)

scala> for (e <- a if e > 2) yield e
res1: Array[Int] = Array(3, 4, 5)
```

Importance of Clean APIs

- Lot of importance has been given to maintain cleaner APIs in SCALA collections.
    - Learning implementation of a function available in a particular collection helps to implement the same function for any other collection, as many important functions are intentionally maintained across various collections.
    - This eases the life of a developer in a big way.

http://www.scala-lang.org/api/current/scala/collection/immutable/List.html

LISTS

```
scala> Nil
res2: scala.collection.immutable.Nil.type = List()

scala> 1 :: 2 :: 3 :: 4 :: 5 :: Nil
res3: List[Int] = List(1, 2, 3, 4, 5)

scala> 5 :: Nil
res4: List[Int] = List(5)

scala> Nil.::(5)
res5: List[Int] = List(5)

scala> 4 :: 5 :: Nil
res6: List[Int] = List(4, 5)

scala> 3 :: 4 :: 5 :: Nil
res7: List[Int] = List(3, 4, 5)

scala> 3 :: 4 :: 5 :: Nil
```

```
object Lists extends App {

  val a = List(1,2,3,4,5)
  val a2 = List.apply(1,2,3,4,5)
  val a3 = 1 :: 2 :: 3 :: 4 :: 5 :: Nil

  println(a.head) //1
  println(a.tail)


}
```

Mutable vs Immutable Collections - Map

- **Defining Immutable Vals:**
  - Immutable VALS follows below properties:
    - Can’t be re-assigned to a new Map Collection

```
scala> val a = Map("AL" -> "Alabama")
a: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama)

scala> val b = Map("AK" -> "Alaska")
b: scala.collection.immutable.Map[String,String] = Map(AK -> Alaska)

scala> a = b
<console>:13: error: reassignment to val
       a = b
         ^
```

  - Follows static typing principles that is Key, Value pairs cant be re-assigned to new types.
  - Overloaded functions like +=, -=, +, - etc. aren’t available for Immutable collections, hence, new Key, Values cant be assigned to an existing Map.

```
scala> a += ("AK" -> "Alaska"); println(a)
<console>:12: error: value += is not a member of scala.collection.immutable.Map[
String,String]
       a += ("AK" -> "Alaska");;
         ^
```

Mutable vs Immutable Collections - Map

- **Defining Immutable Vals continued:**
  - Immutable VALS follows below properties:
    - However, you can create new Maps from existing Maps by manipulating existing maps in the Map declaration.

```
scala> val c = a + ("AR" -> "Arkansas", "AZ" -> "Arizona")
c: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama, AR -> Arkansas, AZ -> Arizona)

scala> val d = c + ("AR" -> "banana")
d: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama, AR -> banana, AZ -> Arizona)

scala> val e = d - "AR"
e: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama, AZ -> Arizona)

scala> val f = e - "AZ" - "AL"
f: scala.collection.immutable.Map[String,String] = Map()
```

  - Being Immutable, you can’t re-assign an existing key to a new value.

```
scala> a("AR") = "foo"
<console>:13: error: value update is not a member of scala.collection.immutable.Map[String,String]
       a("AR") = "foo"
       ^
```

Defining Immutable Maps - as VARS

- **Defining Immutable Vars:**
  - Doing so has a dramatic difference on how you can treat the map. Though it seems VAR overrides the limitations of re-assignment in an Immutable Map, internally, the Variable name is pointed to a new copy of same MAP. Hence, it looks confusing and dramatic.

    - Can be re-assigned to a new Map Collection.

```
scala> var x = Map("AL" -> "Alabama")
x: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama)

scala> a
res2: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama)

scala> x = a
x: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama)
```

    - Follows static typing principles that is Key, Value pairs cant be re-assigned to new types.

```
scala> x("AL") = "foo"
<console>:13: error: value update is not a member of scala.collection.immutable.Map[String,String]
       x("AL") = "foo"
       ^
```

Defining Immutable Maps - as VARS

- **Defining Immutable Vars continued:**

- Overloaded functions like +=, -=, +, - etc. are available for Immutable VARS, hence, new Key, Values can be assigned to an existing Map**.**

```
// add one element
scala> x += ("AK" -> "Alaska"); println(x)
Map(AL -> Alabama, AK -> Alaska)

// add multiple elements
scala> x += ("AR" -> "Arkansas", "AZ" -> "Arizona"); println(x)
Map(AZ -> Arizona, AL -> Alabama, AR -> Arkansas, AK -> Alaska)

// add a tuple to a map (replacing the previous "AR" key)
scala> x += ("AR" -> "banana"); println(x)
Map(AZ -> Arizona, AL -> Alabama, AR -> banana, AK -> Alaska)

// remove an element
scala> x -= "AR"; println(x)
Map(AZ -> Arizona, AL -> Alabama, AK -> Alaska)

// remove multiple elements (uses varargs method)
scala> x -= ("AL", "AZ"); println(x)
Map(AK -> Alaska)

// reassign the map that 'x' points to
scala> x = Map("CO" -> "Colorado")
x: scala.collection.immutable.Map[String,String] = Map(x -> Colorado)
```

Mutable vs Immutable Collections - Map

- **Defining Mutable Vals:**
  - Immutable VALS follows below properties:
    - Can’t be re-assigned to a new Map Collection, being a VAL

```
scala> a
res5: scala.collection.immutable.Map[String,String] = Map(AL -> Alabama)

scala> val MutMapa = scala.collection.mutable.Map("BM" -> "Mumbai", "DL" -> "Delhi")
MutMapa: scala.collection.mutable.Map[String,String] = Map(DL -> Delhi, BM -> Mumbai)

scala> MutMapa = a
<console>:13: error: reassignment to val
       MutMapa = a
                 ^
```

    - Follows static typing principles that is Key, Value pairs cant be re-assigned to new types.
    - Overloaded functions like +=, -=, +, - etc. are available for Mutable collections, hence, new Key, Values cant be assigned to an existing Map.

```
scala> MutMapa += ("KL" -> "Kolkata")
res12: MutMapa.type = Map(KL -> Kolkata, DL -> Delhi, BM -> Mumbai)
```

    - Being Mutable, you can re-assign an existing key to a new value.

```
scala> MutMapa("BM") -> "Bombay"; println(MutMapa)
Map(DL -> Delhi, BM -> Mumbai)
```

Mutable vs Immutable Collections - Map

- **Defining Mutable Vars:**
  - Mutable VARS follows below properties:
    - Can be re-assigned to a new Map Collection, being a VAR

```
scala> var MutMapb = scala.collection.mutable.Map("HYD" -> "Hyderabad", "CMB" -> "Coimbatore")
MutMapb: scala.collection.mutable.Map[String,String] = Map(HYD -> Hyderabad, CMB -> Coimbatore)

scala> MutMapa
res15: scala.collection.mutable.Map[String,String] = Map(KL -> Kolkata, DL -> Delhi, BM -> Mumbai)

scala> MutMapb =MutMapa
MutMapb: scala.collection.mutable.Map[String,String] = Map(KL -> Kolkata, DL -> Delhi, BM -> Mumbai)
```

    - Follows static typing principles that is Key, Value pairs cant be re-assigned to new types.

    - Overloaded functions like +=, -=, +, - etc. are available for Mutable collections, hence, new Key, Values cant be assigned to an existing Map.

```
scala> MutMapb += ("AHM" -> "Ahemadabad")
res27: scala.collection.mutable.Map[String,String] = Map(KL -> Kolkata, Pune -> PN, Kerala -> KL, DL -> Delhi, AHM -> Ahemadabad, BM -> Mumbai)
```

    - Being Mutable, you can re-assign an existing key to a new value.

```
scala> MutMapb("AHM") -> "AHEMADABAD"
res28: (String, String) = (Ahemadabad,AHEMADABAD)

scala> MutMapb
res29: scala.collection.mutable.Map[String,String] = Map(KL -> Kolkata, Pune -> PN, Kerala -> KL, DL -> Delhi, AHM -> Ahemadabad, BM -> Mumbai)
```

Natively supported Design Patterns

1) ***Factory method allows to:***
- merge complex object creation code,
- select which class to instantiate,
- cache objects,
- coordinate access to shared resources.

Eg: Companion Objects

In Java, we use *new* **operator** to instantiate a class, by invoking its constructor. To implement the pattern, we rely on ordinary methods. Also, because we can't define static methods in interface, we have to use a separate factory class:

2) ***Lazy Initialisation:***

Lazy initialization is a special case of lazy evaluation strategy. It's a technique that initialises a value (or an object) on its first access.

Lazy initialisation allows to defer (or avoid) some expensive computation.
A typical Java implementation uses null value to indicate uninitialised state. However, if null is a valid final value, then a separate flag is needed to indicate whether the initialisation process has taken place.

In multi-threaded code, access to the flag must be synchronised to guard against a race condition. Efficient synchronisation may employ double-checked locking, which complicates code even further:

Eg. of Factory Pattern implemented in JAVA vs SCALA

Java

```java
public interface Animal {}

private class Dog implements Animal {}

private class Cat implements Animal {}

public class AnimalFactory {
    public static Animal createAnimal(String kind) {
        if ("cat".equals(kind)) return new Cat();
        if ("dog".equals(kind)) return new Dog();
        throw new IllegalArgumentException();
    }
}

AnimalFactory.createAnimal("dog");
```

In addition to constructors, Scala provides a special syntactic construct, that looks similar to constructor invocation, but it's actually a convenient factory **method**:

Scala

```scala
trait Animal
private class Dog extends Animal
private class Cat extends Animal

object Animal {
  def apply(kind: String) = kind match {
    case "dog" => new Dog()
    case "cat" => new Cat()
  }
}

Animal("dog")
```

The factory method is defined in so-called "companion object" – a special singleton object

Eg. of Lazy Initialisation Pattern implemented in JAVA vs SCALA

Java

```java
private volatile Component component;

public Component getComponent() {
    Component result = component;
    if (result == null) {
        synchronized(this) {
            result = component;
            if (result == null) {
                component = result = new Component();
            }
        }
    }
    return result;
}
```

Scala offers a clean built-in syntax to define lazy values:

Scala

```scala
lazy val x = {
  print("(computing x) ")
  42
}

print("x = ")
println(x)

// x = (computing x) 42
```

Lazy values in Scala can hold *null* values. Access to lazy value is thread-safe.