SCALA INTRO PART ONE

Brian Gordon

SigFig

WHY SCALA?

And not Haskell or F#

- Java has amazing support from open-source.
- Java has amazing tools.
- Java is widely used in the industry.
- Scala is completely interoperable with Java.
- Scala improves on Java.

HELLO WORLD COMPARISON

Java

```
package info.brian_gordon.scalademo;

public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello world!");
   }
}
```

Scala

```
package info.brian_gordon.scalademo

class HelloWorld {
    def main(args: Array[String]) = {
        println("Hello world!")
    }
}
```

IF YOU KNOW JAVA, YOU CAN WRITE SCALA TODAY

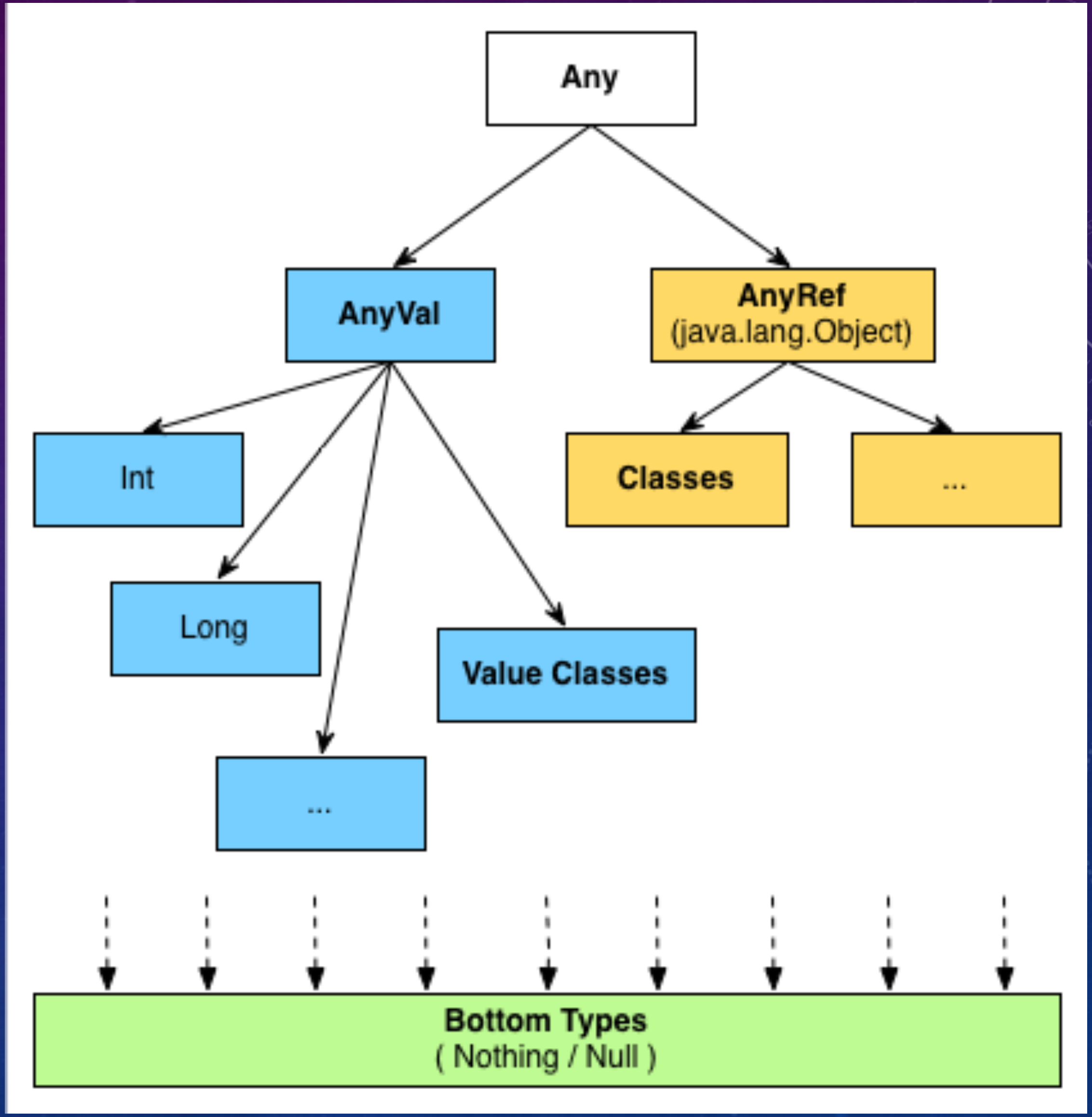
- Code goes in classes (usually)
- Classes have methods and fields
- Classes have constructors
- Fields can be final or not (caution!)
- Methods/fields can be public, private, etc
- References can be null (caution!)
- You can overload methods
- Classes have inheritance through “extends”
- You can polymorphically override methods (and fields)
- Generics (though variance is different)
- The memory model is the same (caution!)
- You can cast (don’t)
- Same primitives as Java
- Objects have equals/hashcode/toString like Java

PASS BY NAME

```
def log(message: => String) =
    if (loggingEnabled) {
        logFile.append(message)
    }
```

TYPE INFERENCE

TYPE HIERARCHY



Source: https://ktoso.github.io/scala-types-of-types/

CLASSES

- There's one primary constructor for each class.
- You may define auxiliary constructors, but they must call either the primary constructor or another auxiliary constructor as their first action (like Java). This is not common.
- Primary constructor arguments are in scope within the class body.
- Primary constructor arguments can be automatically made into fields.

VISIBILITY MODIFIERS

- public – the default
- protected – subclasses only, unlike Java
- private[package]
- private[this]

GENERICS

```
class Holder[T](val item: T)

new Holder(3).item + 5
```

```
public class Holder<T> {
    public final T item;
    public Holder(T item) {
        this.item = item;
    }
}

new Holder<>(3).item + 5
```

TYPE VARIANCE POP QUIZ

```
List<List<? extends Number>> a = null;
List<List<Integer>> b = null;
a = b;
```

TYPE VARIANCE POP QUIZ

Incompatible types.

Required:

```
java.util.List<java.util.List<? extends java.lang.Number>>
```

Found:

```
java.util.List<java.util.List<java.lang.Integer>>
```

WHY DOES IT FAIL TO COMPILE?

```
List<List<? extends Number>> a = null;
List<List<Integer>> b = new ArrayList<>();
a = b; // Say this succeeds.

a.add(Lists.newArrayList(2.718));
Integer c = b.get(0).get(0);

// c is an Integer containing 2.718
```

HOW DO WE GET IT TO COMPILE?

```
List<List<? extends Number>> a = null;
List<List<Integer>> b = null;
a = b; // Error
```

```
List<? extends List<? extends Number>> c = null;
List<List<Integer>> d = null;
c = d; // OK
```

http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeArguments.html#FAQ104

“I am completely and totally humbled. Laid low. I realize now that I am simply not smart at all. I made the mistake of thinking that I could understand generics. I simply cannot. I just can't. This is really depressing. It is the first time that I've ever not been able to understand something related to computers, in any domain, anywhere, period.”

“

We simply cannot afford another *wildcards*.

”

—Joshua Bloch, 2007

USE-SITE VARIANCE

```
interface Consumer<T> {
    void consumeList(List<T> values);
}

interface Producer<T> {
    List<T> produceList();
}

public class PECS<T> {
    Consumer<? super T> consumer;
    Producer<? extends T> producer;
}
```

PECS

Producers: `extend`

Consumers: `super`

- Key insight: if we document these on the interface’s type parameters, things can “just work.”

DECLARATION-SITE VARIANCE

```
package java.util;
public interface List<E> {
   boolean add(E e);
   E get(int index);
}
```

DECLARATION-SITE VARIANCE

- `class ArrayBuffer[A]`
- `class List[+A]`

DECLARATION-SITE VARIANCE

```
package scala.collection.immutable
class List[+A] {
   def head: A
   def tail: List[A]
   def prepend[B >: A] (x: B): List[B]
}
```

OBJECTS

```
class SecretHolder(secret: String) {
    private def printSecret() = println(secret)
}
object SecretHolder {
    def betraySecret(holder: SecretHolder) =
        holder.printSecret()
}
```

CASE CLASSES

```
case class Person(name: String, age: Int, height: Int)
val harold = Person("Harold", 42, 183)
```

- Auto hashcode/equals/toString
- Auto apply/unapply

TUPLES

```
(1, 2, 3)

("a", 7, List)._2

def printFirst(input: (Int, String)) = {
    val (first, second) = input
    println(first)
}
```

PATTERN MATCHING

```
case class Person(name: String, age: Int, height: Int)

val Person(haroldsName, haroldsAge, haroldsHeight) = Person("Harold", 42, 183)
```

Un apply

Apply

PATTERN MATCHING

```
class MyPerson(val name: String, val age: Int, val height: Int)
object MyPerson {
    def apply(name: String, age: Int, height: Int) =
        new MyPerson(name, age, height)
    def unapply(person: MyPerson) =
        Some((person.name, person.age, person.height))
}

val MyPerson(geraldsName, geraldsAge, geraldsHeight) =
    MyPerson("Gerald", 71, 160)
```

PATTERN MATCHING

```
val haroldsAge = harold match {
    case Person(name, age, height) => age
}

val haroldsAge = harold match {
    case Person(_, age, _) => age
}
```

```scala
object MultipleOfTwo {
    def unapply(number: Int) = {
        if (number % 2 == 0) Some(number / 2)
        else None
    }
}


object MultipleOfThree {
    def unapply(number: Int) = {
        if (number % 3 == 0) Some(number / 3)
        else None
    }
}


7 match {
    case MultipleOfTwo(factor) => factor
    case MultipleOfThree(factor) => factor
}
```

APPLY

```
class Multiplier(x: Int) {
    def apply(y: Int) = x * y
}

val doubler = new Multiplier(2)
doubler(7)
doubler(20)
```

OPERATORS

```
BigDecimal(5.0).+(BigDecimal(4.9))
BigDecimal(5.0) + BigDecimal(4.9)
```

RIGHT-ASSOCIATIVE OPERATORS

- If your operator ends in a colon, it will be right-associative.

```
val numbers = List(11, 7, 5, 3, 2)
13 :: numbers
```

SCALA LISTS

- Every list has a head and a tail.
- Nil is the empty list.
- You can build up a list from Nil by appending elements to the head with Cons.
- Completely immutable. If you want to change the list, you have to create a new one (though since lists are immutable you can share sublists!)

“

Referential transparency

”

An expression can be replaced by its value without changing the semantics of the program.

- Methods should return values, not mutate state
- Use immutable collections
- Use final variables

FOR LOOP COMPARISON

```java
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}

for (String arg : args) {
    System.out.println(arg);
}
```

```scala
for (i <- 1 to 10) {
    println(i)
}

(1 to 10).map(i => println(i))

(1 to 10).map(println)
```

SUMMING A LIST OF INTEGERS

```
def sum1(numbers: List[Int]): Int = {
    var sum = 0
    for (number <- numbers) {
        sum += number
    }
    sum
}
```

SUMMING A LIST OF INTEGERS

```
def sum2(numbers: List[Int]): Int =
    numbers match {
        case (head :: tail) => head + sum2(tail)
        case _ => 0
    }
```

SUMMING A LIST OF INTEGERS

```
@tailrec
def sum3(numbers: List[Int], accumulator: Int = 0): Int =
    numbers match {
        case head :: tail => sum3(tail, accumulator + head)
        case _ => accumulator
    }
```

SUMMING A LIST OF INTEGERS

```
def sum4(numbers: List[Int]): Int =
    numbers.fold(0)((acc, cur) => acc + cur)
```

SUMMING A LIST OF INTEGERS

```
def sum4(numbers: List[Int]): Int =
    numbers.fold(0)((acc, cur) => acc + cur)

def sum5(numbers: List[Int]): Int =
    numbers.fold(0)(_ + _)
```

SUMMING A LIST OF INTEGERS

```
def sum6(numbers: List[Int]): Int =
    numbers.sum
```

JSON AST EXAMPLE