# Scala vs. Java

A Comparison

# Content

- Introduction
  - History/Technical Facts
  - Hello World
- Building Blocks
  - Scala Type System
  - Program Structures
- Scala Concepts
  - Language Concepts (not available in Java)

# Introduction

- History
- Technical Facts
- Hello World

# History

- 2001: Martin Odersky at EPFL Lausanne
  (Is one of the developers of the *Java Generics*.)
- 2003: First release
- 2006: Second release
- 2011: Typesafe founded
- 2018: Current Version is 2.12.6
- 2020: Scala 3.0 (Dotty compiler)

# Technical Facts

- JVM based language (compiles to Java class files)
- Statically typed with unified type system
- Object oriented/functional hybrid
- Interoperability with Java
- No checked exceptions
- Everything is an expression (almost)
- Typesafe stack: Scala/Akka/Play/Lagoom

# Hello World

*Scala*

```scala
package at.twinformatics.scalademo.scala

object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello, World!")
  }
}
```

*Java*

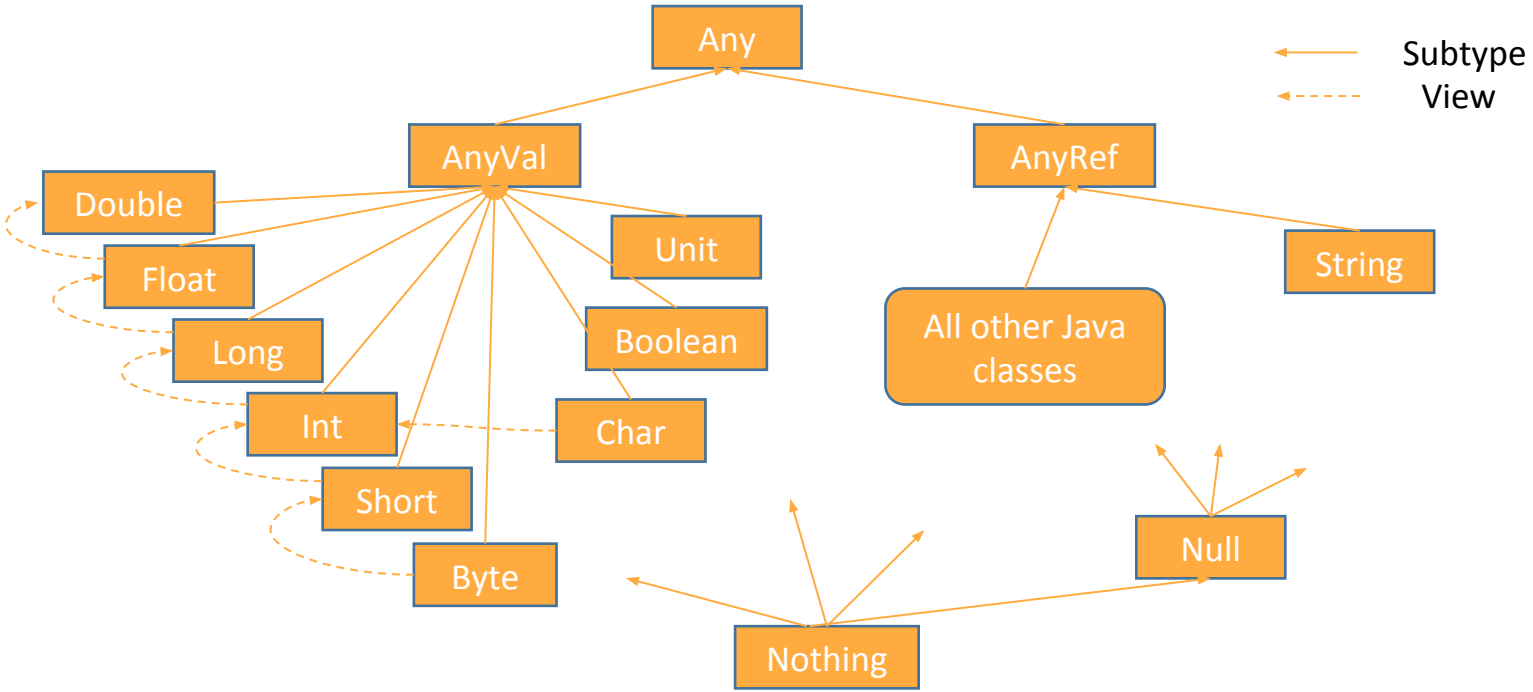
```java
package at.twinformatics.scalademo.java;

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

# Building Blocks

Structuring your Program

- Scala Type System
- Traits
- Classes
- Objects
- Case Classes
- Function Types
- Generics

# Scala Type System



# Nothing

```
def toInt(s: String): Int = {
  if (s.nonEmpty) s.toInt
  else raiseError()
}

def raiseError(): Nothing = {
  throw new AssertionError("Got empty string.")
}

def systemExit(code: Int): Nothing = sys.exit(code)

case object Nil extends List[Nothing]
```

# Traits

*Scala*

```scala
trait Iterator[T] {
  def hasNext: Boolean
  def next: T
  def remove(): Unit = {
    throw new UnsupportedOperationException()
  }
}
```

*Java*

```java
public interface Iterator<T> {
  boolean hasNext();
  T next();
  default void remove() {
    throw new UnsupportedOperationException();
  }
}
```

# Classes

*Scala*

```scala
class Fraction(val num: Long, val denom: Long) {
  def *(other: Fraction): Fraction = new Fraction(
    num*other.num,
    denom*other.denom
  )
  override def toString: String = s"$num/$denom"
}

println(Fraction(10, 2).*(Fraction(5)))
println(Fraction(10, 2) * Fraction(5))
```

*Java*

```java
public class Fraction {
  private final long num;
  private final long denom;

public Fraction(final long num, final long denom) {
    this.num = num;
    this.denom = denom;
  }
  public long num() {
    return num;
  }
  public long denom() {
    return denom;
  }
  public Fraction mul(final Fraction other) {
    return new Fraction(
      num*other.num,
      denom*other.denom
    );
  }
  @Override
  public String toString() {
    return String.format("%d/%d", num, denom);
  }
}
```

# Objects

*Scala*

```
object Fraction {
  def apply(n: Long, d: Long): Fraction = {
    new Fraction(n, d)
  }
  def apply(v: Long): Fraction = Fraction(v, 1)
}
```

*Java*

```
public class Fraction {
  // ...
  public static Fraction of(long n, long d) {
    return new Fraction(n, d);
  }
  public static Fraction of(long v) {
    return of(v, 1);
  }
}
```

# Case Classes

*Scala*

```scala
case class Fraction(num: Long, denom: Long) {
  def *(other: Fraction): Fraction = Fraction(
    num*other.num,
    denom*other.denom
  )
}

object Fraction {
  def apply(v: Long): Fraction = Fraction(v, 1)
}
```

*Java*

```java
public class Fraction {
  // ...
  @Override
  public int hashCode() {
    return Long.hashCode(num)^Long.hashCode(denom);
  }
  @Override
  public boolean equals(final Object obj) {
    return obj == this ||
      obj instanceof Fraction &&
      num == ((Fraction)obj).num &&
      denom == ((Fraction)obj).denom;
  }
  @Override
  public String toString() {
    return String.format("%d/%d", num, denom);
  }
}
```

# Function Types

*Scala*

```scala
val f1a: (String => Int) = s => s.toInt
val f1b: (String => Int) = _.toInt
val f2: String => Int = Integer.parseInt
val f3 = (s: String) => s.toInt

val value1: Int = f1.apply("4")
val value2: Int = f1("3")
```

*Java*

```java
final Function<String, Integer> f1 =
    s -> Integer.parseInt(s);
final Function<String, Integer> f2 =
    Integer::parseInt;

final Integer value1 = f1.apply("4");
```

# Currying

```
object Currying extends App {

  def modN(n: Int)(x: Int) = (x%n) == 0

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(nums.filter(modN(2)))
  println(nums.filter(modN(3)))

  // List(2, 4, 6, 8)
  // List(3, 6)
}
```

# Generics

*Scala*

```
trait Iterator[+T] {
  def hasNext: Boolean
  def next: T
}

val strings: Iterator[String] = ...
val objects: Iterator[AnyRef] = strings


val integers: Iterator[Integer] = ...

def iterate(it: Iterator[Number]): Unit = ()
def iterate2[T <: Number](it: Iterator[T]): Unit = ()

iterate(integers)
iterate2(integers)
```

*Java*

```
interface Iterator<T> {
  boolean hasNext();
  T next();
}

Iterator<String> strings = ...;
Iterator<? extends Object> objects = strings;
Iterator<Object> objects = strings; //Won't compile


Iterator<Integer> integers = ...;

void iterate(Iterator<? extends Number> it) {}
<T extends Number> void iterate2(Iterator<T> it) {}

iterate(integers);
iterate2(integers);
```

# Scala Concepts

Not available in Java

- Lazy Values
- Default/Named Parameters
- Pattern Matching
- Algebraic Data Types
- Extractors
- Implicits
- Extension Methods
- Tail Recursion
- Value Types
- Structural Types
- Specialized Types
- Reified Types
- Dynamic Types
- Type Alias
- (Exception Handling)

# Lazy Values

*Scala*

```
object LazyValue {
  lazy val lazyValue: String = {
    println("Initialize lazy value")
    "Lazy value"
  }
  val eagerValue: String = {
    println("Initialize eager value")
    "Eager value"
  }
  def main(args: Array[String]): Unit = {
    println("Starting program")
    println(lazyValue)
    println(eagerValue)
  }
}
```

*Output*

```
Initialize eager value
Starting program
Initialize lazy value
Lazy value
Eager value
```

# Default/Named Parameter

*Scala*

```
object Parameters {
  case class Fraction(num: Long, denom: Long = 1)

  def print(s: String, times: Int = 1): Unit = {
    for (i <- 0 until times) println(s)
  }
  def main(args: Array[String]): Unit = {
    println(Fraction(10))
    print("Hello")
    print("World", 3)
    print("Nothing", times = 0)
  }
}
```

*Output*

```
Fraction(10,1)
Hello
World
World
World
```

# Pattern Matching

```
def describe(x: Any): String = x match {
  case 5 => "five"
  case true => "truth"
  case "hello" => "hi!"
  case Nil => "the empty list"
  case s: String => s"got string '$s'"
  case _: Number => "a number"
  case _ => "everything else"
}
```

# Algebraic Data Types

```
sealed abstract class Expr

final case class Var() extends Expr
final case class Const(v: Double) extends Expr
final case class Add(x: Expr, y: Expr) extends Expr
final case class Sub(x: Expr, y: Expr) extends Expr
final case class Mul(x: Expr, y: Expr) extends Expr
final case class Div(x: Expr, y: Expr) extends Expr

object eval {
  def apply(expr: Expr, value: Double): Double = expr match {
    case Var() => value
    case Const(x) => x
    case Add(x, y) => eval(x, value) + eval(y, value)
    case Sub(x, y) => eval(x, value) - eval(y, value)
    case Mul(x, y) => eval(x, value) * eval(y, value)
    case Div(x, y) => eval(x, value) / eval(y, value)
  }
}
```

# Extractor (1)

```scala
class Pair(val a: Int, val b: String)
object Pair {
  def apply(a: Int, b: String): Pair = new Pair(a, b)
  def unapply(ext: Pair): Option[(Int, String)] = {
    Some(ext.a, ext.b)
  }
}

object ExtractorApp extends App {
  def process(any: Any): Unit = any match {
    case Pair(a, b) => println(s"Pair $a and $b")
    case _ => println(s"Unknown: $any")
  }

  process(Pair(1, "2")) // Pair 1 and 2
  process("Some Value") // Unknown: Some Value
}
```

# Extractor (2)

```
object IntEntry {
  def unapply(s: String): Option[(String, Int)] = {
    val split: Array[String] = s.split(":")
    if (split.length == 2) {
      Try(split(1).toInt).toOption.map((split(0), _))
    } else {
      None
    }
  }
}
object ExtractorTest extends App {
  def process(any: Any): Unit = any match {
    case IntEntry(name, value) => println(s"Got Entry[$name, $value]")
    case _ => println(s"Unknown entry: $any")
  }

  process("count:12") // Got: Entry[count, 12]
  process("count:12.2") // Unknown entry: count:12.2
  process("invalid")  // Unknown entry: invalid
}
```

# Pattern Matching (2)

```
case class Point(x: Int, y: Int)
case class Line(start: Point, end: Point)

object ExtractorTest extends App {
  def process(any: Any): Unit = any match {
    case Point(x, y) if x < y => println(s"Point[$x, $y]")
    case Line(Point(x, _), Point(_, y)) => println(s"Start x: $x, End y: $y")
    case _ => println(s"Unknown: $any")
  }

  process(Point(1, 2)) // Point[1, 2]
  process(Line(Point(1, 2), Point(3, 4))) // Start x: 1, End y: 4
}
```

# Implicits

```
private val CHARS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
def randomString(length: Int)(implicit r: Random): String = {
    Stream.continually(CHARS.charAt(r.nextInt(CHARS.length)))
        .take(length)
        .mkString
}

def main(args: Array[String]): Unit = {
    println(randomString(10)(new Random()))

    implicit val random: Random = new Random(123)
    println(randomString(15))
}
```

# Extension Methods

```
final class RichInt(value: Int) {
  def isPrime: Boolean = {
     BigInt(value).isProbablePrime(20)
  }
}

object RichInt {
  implicit def toRichInt(value: Int): RichInt = new RichInt(value)
}

object PrimeTest extends App {
  import RichInt.toRichInt

  println(2.isPrime)  // true
  println(1323123.isPrime)  // false
}
```

# Tail Recursion

*Scala*

```scala
def fibrec(n: Int): Int = n match {
  case 0 | 1 => n
  case _ => fibrec(n - 1) + fibrec(n - 2)
}

@tailrec
def fib(x: Long, prev: Long = 0, next: Long = 1): Long = x match {
  case 0 => prev
  case 1 => next
  case _ => fib(x - 1, next, next + prev)
}

def main(args: Array[String]): Unit = {
  println(fibrec(10))
  println(fib(10))
}
```

*Java*

```java
public static long fib(long x, long prev, long next) {
  while(true) {
    long var8;
    if (0L == x) {
      var8 = prev;
    } else {
      if (1L != x) {
        long var10000 = x - 1L;
        long var10001 = next;
        next += prev;
        prev = var10001;
        x = var10000;
        continue;
      }
      var8 = next;
    }
    return var8;
  }
}
```

# Value Types

```
object ValueTypes extends App {

  implicit class RichInt(val value: Int) extends AnyVal {
    def isPrime: Boolean = {
      BigInt(value).isProbablePrime(20)
    }
  }

  println(2.isPrime)  // true
  println(1323123.isPrime)  // false

  // Actual call
  // RichInt.isPrime(2)
}
```

# Structural Types

```
def using[C <: {def close(): Unit}, T](closeable: C)(block: C => T) = {
  try {
    block(closeable)
  } finally {
    try {
      closeable.close()
    } catch {
      case ignore: Exception =>
    }
  }
}
```

```
val msg = using(new ByteArrayInputStream("hello".getBytes)) { in =>
  Source.fromInputStream(in).mkString
}
```

# Specialized Types

```
case class Complex[@specialized(Int, Double)T](re: T, im: T)
```

- Generated classes

```
Complex.class
Complex$.class
Complex$mcD$sp.class
Complex$mcI$sp.class
```

# Reified Types

```
object Reified extends App {

  def array[T <: AnyRef](length: Int)(implicit  tag: ClassTag[T]): Array[T] = {
    val a = new Array[T](length)
    for (i <- 0 until length) {
      a(i) = tag.runtimeClass.newInstance().asInstanceOf[T]
    }
    a
  }

  val a1: Array[String] = new Array(5)
  val a2: Array[String] = array(5)

  println(a1.mkString("[", ",", "]"))
  println(a2.mkString("[", ",", "]"))

  // [null,null,null,null,null]
  // [,,,,]
}
```

# Dynamic Types

```
object Arithmetic extends Dynamic {
  def applyDynamic(methodName: String)(args: Int*): Int = {
    methodName match {
      case "add" => args.sum
      case "mul" => args.product
      case _ => throw new UnsupportedOperationException()
    }
  }
}

println(Arithmetic.add(1, 2, 3, 4)) // 10
println(Arithmetic.mul(1, 2, 3, 4)) // 24
println(Arithmetic.div(4, 2))       // Exception
```

# Type Alias

```
type Property = Map[String, String]
type IntFunction[+T] = Int => T
type Consumer[-T] = T => Unit
type Mapper[-T, +R] = T => R
type StringMapper[+R] = Mapper[String, R]

import java.lang.{Double => jdouble}
import java.util.function.{Function => JFunction}
```

# Exception Handling

*Scala*

```
def readText1(file: Path): Option[String] = {
  try {
    Some(new String(readAllBytes(file)))
  } catch {
    case e: IOException => None
  }
}

def readText2(file: Path): Option[String] = {
  try {
    Some(new String(readAllBytes(file)))
  } catch {
    handler(false)
  }
}

def handler(ignoreNull:  Boolean):
PartialFunction[Throwable, Option[String]] = {
  case e: NullPointerException if ignoreNull => None
  case _: IOException => None
}
```

*Java*

```
Optional<String> readText(Path file) {
  try {
    return Optional.of(new String(readAllBytes(file)));
  } catch (IOException e) {
    return Optional.empty();
  }
}
```