Scala Basics

J. Heinzelreiter

Version 2.0

Optional Braces (Scala 3)

- In Scala 3 braces are optional in many cases.

```scala
trait A {
  def func(): Unit
}

class C(x: Int) extends A {
  def func(): Unit = {
    var i = 0;
    while (i < 10) {
      println(i)
      i += 1
    }
    println("done")
  }
}
```

```scala
trait A:
 def func(): Unit

class C(x: Int) extends A:
  def func(): Unit =
    val i=0
    while i<10 do
      println(i)
      i += 1
    println("done")
end C // optional
```

- Tabs and spaces may be mixed, but prefixes must be comparable
  - 2 tabs + 2 spaces and 2 tabs + 4 spaces are comparable
  - 2 tabs + 2 spaces and 3 tabs + 2 spaces are not comparable

Variable definition and function declaration

- The variable's type is placed after variable's name:

```
var msg : String = "hello"
```

- Often the compiler can infer the type of a variable:

```
var msg = "hello"
```

- Definition of functions (methods)

```
def min(x: Int, y: Int): Int = if x<y then x else y
```

  - The result type is placed at the end of the function heading.
  - Often the result type can also be automatically inferred:

```
def min(x: Int, y: Int) = if x<y then x else y
```

  - `Unit` is the result type of functions that return no value:

```
def trace(s: String): Unit = println(s)
```

```
def trace(s: String) = println(s) // result type inferred
```

Variables – `val` vs. `var`

- Variables declared as `val`:
  - Once initialized, one cannot reassign a new value.
  - Similar to a final variable in Java.

```
val pi = 3.14159
pi = 2.7183 // syntax error
```

- Variables declared as `var`:
  - The variable's value can always be changed.

```
var x = 1.0
x = 2.0 // ok
```

Functions

- In Scala *functions* are *first class* citizens of the type system.
  - There are *function literals:*

```
val sum = (a: Int, b: Int) => a + b
```

  - Functions have a *type:*

```
val sum: (Int, Int) => Int = (a: Int, b: Int) => a + b
```

  - Functions can be passed around as *values*:

```
def applyFunc(f: (Int, Int) => Int, a: Int, b: Int) = f(a, b)
```

- Function Literals
  - *Target typing:* Compiler can deduce type of function arguments

```
applyFunc((a, b) => a*b, 3, 4)
```

  - Placeholder syntax:

```
applyFunc(_*_, 3, 4)
```

Currying

- Functions can have *multiple argument lists*:

```
def product(a: Int)(b: Int) = a * b
val p = product(3)(4)
```

  - `product` is split into two internal functions.
  - The function `product(a)` returns a function.
  - One can apply parameter list `(b)` to this function.
- Arguments can be applied partially:

```
val twice : Int => Int = product(2)
val t = twice(5)
```

  - One can get a reference to the second function.
- Application
  - Creating new control structures.
  - Passing implicit values.

Call by Name

- Sometimes arguments must not be evaluated when a function is called.
- Possible solution: Passing a function instead of a value:

```
def myAssert(predicate: () => Boolean) =
  if (assertionsEnabled && !predicate())
    throw new AssertionError
myAssert(() => ref != null)
```

- Scala has a better solution: *by-name parameters*

```
def myAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
myAssert(ref != null)
```

  - Parameter is only evaluated when it is referenced in the function.

Custom Control Structures

- Methods taking functions as parameters can be used to create new control structures:

```
@tailrec
def myWhile(testCondition: => Boolean)(codeBlock: => Unit) =
  if (testCondition)
    codeBlock;
    myWhile(testCondition)(codeBlock);
```

```
var i = 0;
myWhile (i<5) { println(i); i += 1 }
```

- There is an alternative syntax for calling methods with one parameter:

```
singleArgumentMethod(argument)
singleArgumentMethod { argument }
```

Partial Functions

- A partial function is a function that is not defined for all argument values:

```
val sqrt = new PartialFunction[Double, Double] =
  def apply(x: Double) = Math.sqrt(x)
  def isDefinedAt(x: Double) = x >= 0

val s = sqrt(4);             // same as sqrt.apply(2)
val d = sqrt.isDefinedAt(-4) // returns false
```

- There is a special syntax for creating partial functions:

```
val sqrt : PartialFunction[Double, Double] = {
  case x : Double if x>=0 => Math.sqrt(x)
}
val s = sqrt(-4)             // throws scala.MatchError
val d = sqrt.isDefinedAt(-4) // returns false
```

- A partial function is a subtype of a (total) function

```
val f: Double -> Double = sqrt
```

Implicit Parameters

- If a function definition declares an implicit parameter list

```
def f(a: A)(implicit b: B)
```

  - the compiler automatically inserts values that are marked implicit (and are in scope).

```
implicit val bVal = new B()
f(a) // → f(a)(bVal)
```

  - Parameter can also be passed explicitly:

```
f(a)(new B())
```

- Example:

```
def printToConsole(s: String)(implicit prompt: String) = println(prompt + " " + s)
implicit val p = "=>"
printToConsole("hello")("->") // -> hello
printToConsole("world")       // => world
```

- Often implicit values are defined in preference objects.

Given Instances and Using Clauses (Scala 3) (1)

- In Scala 3 a parameter can be declared as *contextual* by using the `using` keyword:

```
def f(a: A)(using b: B)
```

  - One can assign a default value to a type using the keyword `given`:

```
given B = B() // → bGiven
```

  - When the contextual parameter is omitted, the given value is used implicitly:

```
f(a) // → f(a)(bGiven)
```

  - Contextual parameters can also be passed explicitly:

```
f(a)(using B())
```

- Example:

```
def printToConsole(s: String)(using prompt: String) = println(prompt + " " + s)
given String = "=>"
printToConsole("hello")(using "->") // -> hello
printToConsole("world")             // => world
```

Given Instances and Using Clauses (Scala 3) (2)

- Given types can also be parametric:

```
trait Ord[T]:
  def compare(x: T, y: T): Int
  extension (x: T) def <(y: T) = compare(x, y) < 0
```

```
def min[T](a: T, b: T)(using ord: Ord[T]): Boolean = if a < b then a else b
```

- Implement trait and generate a given instance using `with`:

```
object Ord:
  given intOrd: Ord[Int] with
    def compare(x: Int, y: Int): Int = if x < y then -1 else if x > y then 1 else 0
```

- There are different ways to import given instances:

```
import Ord.given              // all givens
import Ord.intOrd             // Ord[Int]
import Ord.{given Ord[Int]} // import Ord[Int] by type
val m = min(4,3)              // → min(4,3)(using intOrd);
```

Implicit Conversion (Scala 3)

- *Type conversion functions* of the form:

```
given convertAtoB: Conversion[A, B] = a => new B(a)
```

  - are automatically applied by the compiler, when a method is called with an object of type A, but there is only a method that accepts a B object.

```
obj.m(a) → obj.m(convertAtoB(a))
```

  - The conversion is also applied to the receiver of a method call:

```
a.m() → convertAtoB(a).m()
```

- Example:

```
class Rational(val num: Int, val denom: Int):
  def +(that: Rational) = new Rational(…, …)
given intToRational: Conversion[Int, Rational] = i => new Rational(i,1)
val half = new Rational(1,2)
val sum1 = half + 1 // half.+(intToRational(1))
val sum2 = 1 + half // intToRational(1).+(half)
```

Extension Methods (Scala 3)

- One can add methods to types after they are defined.

```
extension (s: String)
  def wordCount() = s.split("\\s+").length
  def <(t: String): Boolean = s.compareTo(t) < 0
```

```
println("abc ef ghi".wordCount())
println("abc" < "efg")
```

- Extensions can also be defined for generic types.

```
extension [T](xs: List[T])
  def second = xs.tail.head
  def sumBy[U: Numeric](f: T => U): U = xs.map(f).sum
```

```
var list = List("abc", "efg", "hi")
println(list.second)
println(list.sumBy(_.length))
```

Classes and Objects

- Java and Scala share the same fundamental OO concepts.

```
class Rational:
  private val num:   Int = 0
  private val denom: Int = 1
  def max(that: Rational) = …
  def +(that: Rational): Rational = …
  def incrBy(incr: Int = 1) = …
  private def reduce() = …
end Rational
val r = new Rational
```

- But there are significant differences:
  - Members are public by default.
  - Scala supports operator overloading.
  - Methods can have default values.
  - Scala allows no static members → Companion Object.

Constructors

- Primary constructor

```
class Rational(n: Int, d: Int):
  private val num:   Int = n
  private val denom: Int = d
  reduce()
  …
```

  - Parameters are declared in the class's heading.
  - The primary constructor executes the statements in the class's body.

- Auxiliary constructors

```
class Rational(n: Int, d: Int):
  def this(n: Int) = this(n, 1)
  def this()       = this(0)
  …
```

Parametric Fields

- Often constructor parameters are just copied into fields:

```
class Rational(n: Int, d: Int):
  private val num:   Int = n
  private val denom: Int = d
  …
```

- In Scala there is a shorthand syntax for this recurring pattern: Parameters of a primary constructor can be declared as fields:

```
class Rational(private val num: Int, private val denom: Int)
  …
```

Singleton and Companion Objects

- Instead of static members, Scala has *singleton objects*.

```
object Rational:
  def gcd(a: Int, b: Int): Int = if (b==0) a else gcd(b, a%b)
  def apply(n: Int, d: Int = 0) = new Rational(n, d)
```

- Methods of singleton objects can be called like static methods:

```
val gcd = Rational.gcd(24, 16)
```

- There is a shorthand syntax for calling the `apply` method:

```
val r = Rational(24, 16) // same as Rational.apply(24,16)
```

- When the singleton object shares the name with a class it is called *companion object*.

```
class Rational(…):
  import Rational._
  private def reduce() = { val g = gcd(num, denom); … }
```

Case Classes

- Case classes are classes that are equipped with additional functionality.

```
abstract class Expr
case class Number(num: Double) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr)
                 extends Expr
```

- Case classes have a factory (apply) method:

```
val sum = BinOp("+", Number(3.14), Number(1.41))
```

- All arguments are parametric fields (`val`):

```
val arg1 = sum.left
```

- There is a `copy` method for making modified copies:

```
val prod = sum.copy(operator="*")
```

- There are natural implementations of `equals`, `hashCode`, and `toString`:

```
println(prod) // BinOp(*,Number(3.14),Number(1.41))
```

Pattern Matching

- Pattern matching allows you to analyze the structure of objects.
- Application
  - Partial functions:
    ```
    val f: PartialFunction[T,R] = { (pattern => expr)* }
    ```
  - `match` expressions:
    ```
    x match { (pattern => expr)* }
    ```
- Supported Features:

```
val res: String = x match
  case "abc"      => … // constant pattern
  case Pi         => … // constant pattern: first character is upper case
  case v          => … // variable pattern: matched value is assigned to v
  case i: Int     => … // typed pattern
  case i: Int if i%2 == 0 => … // typed pattern with pattern guard
  case (a,b)      => … // tuple pattern
  case t @ (a,b) => … // expression (a,b) is assigned to t
  case _          => … // wildcard pattern
```

  - Patterns are evaluated in the order they are written

Pattern Matching and Case Classes

- Pattern matching unfolds its full power when it is used with case classes.
- An expression like

```
BinOp("+", e, Number(b))
```

  is matched against the constructor of the respective case class:

```
case class BinOp(operator: String, left: Expr, right: Expr)
```

- Scala supports *deep matches*: `Number(b)` is matched against the constructor of the `Number` case class.
- Example:

```
def simplify(expr : Expr) : Expr = expr match
  case BinOp("+", Number(a), Number(b)) => Number(a+b)
  case BinOp("+", left, Number(0)) => simplify(left)
  case BinOp("+", left, right) if (left == right) =>
         simplify(BinOp("*", Number(2), simplify(left)))
  case BinOp("*", left, Number(1)) => simplify(left)
  case BinOp("*", left, Number(0)) => Number(0)
  case _ => expr
```

Pattern Matching on Lists

- `List` supports the right associative `::` (cons) operator:

```
1 :: List (2,3,4) → List(1,2,3,4)
```

- A list can be built by repeatedly prepending elements to an empty list.

```
1 :: 2 :: 3 :: 4 :: Nil → 1 :: (2 :: (3 :: (4 :: Nil))) → List(1,2,3,4)
```

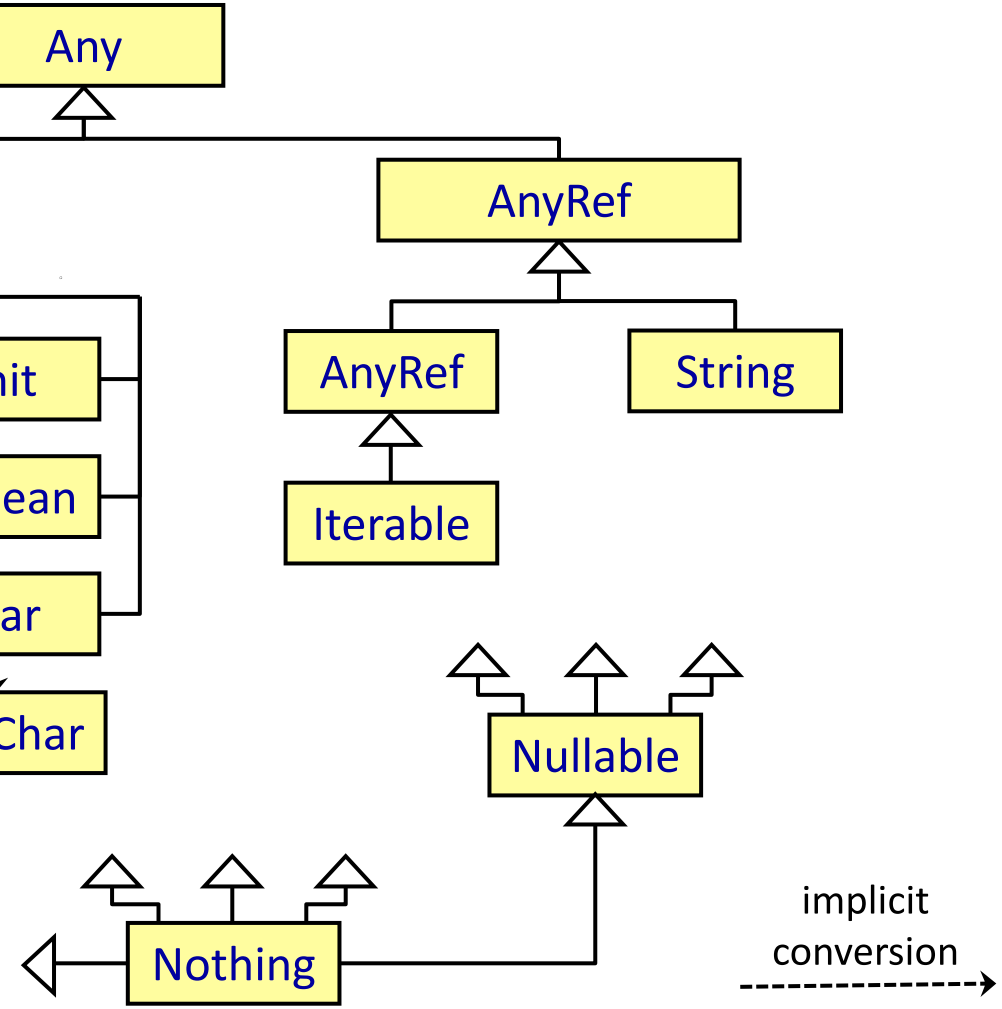
- Pattern matching can also be applied to lists:

```
List(1,2,3,4) match { case a :: b :: r =>
                        println(s"a=$a, b=$b, r=$r")}
// → a=1, b=2, r=List(3, 4)
```

- This is the basis of most recursive algorithms on lists:

```
def sumOf(l : List[Int]): Int = l match
  case Nil => 0
  case first :: rest => first + sumOf(rest)
```

Scala's Type Hierarchy (1)



Scala's Type Hierarchy (2)

- Value classes are implicitly converted into wider types.
- The Rich classes extend the functionality of the corresponding simple class.
- `scala.AnyRef` corresponds to `java.lang.Object`.
- `Nullable` is the subtype of every reference type and the literal `null`.
- `Nothing` is the subtype of every other type (bottom class).
  - Application 1:

```
def ??? : Nothing = throw new NotImplementedError
def someFunction(): Int = ???
```

  - Application 2:

```
class Base[+T] // Base is covariant
object None extends Base[Nothing]
val none : Base[String] = None // None subtype of Base[Nothing]
                      // Base[Nothing] subtype of Base[String]
```

Traits – Rich Interfaces

- Traits are the counterpart of Java interfaces.

```
trait Ordered[T]:
  def compare(that: T): Int
  def <(that: T) = (this compare that) < 0
  def >(that: T) = (this compare that) > 0
}
```

  - Traits may also contain method implementations → *rich interface*.
- Traits are *mixed in* to a class either using the `extends` or the `with` keyword:

```
class Integer(val value: Int) extends Number with Ordered[Integer] {
  override def compare(that: Integer) = value.compare(that.value)
}
```

```
val b = new Integer(1) < new Integer(2)
```

- Multiple traits can be mixed in.
  - Multiple inheritance: Method called by `super.m()` depends on where the call appears.
  - With traits, the method called is determined by a *linearization* of the classes and traits.

Traits: Stackable Modifications (1)

- Given a trait `Printer` and an implementing class `ConsolePrinter`:

```
trait Printer:
 def print(msg: String)

class ConsolePrinter extends Printer {
  override def print(msg: String) = scala.Predef.println(msg)
}
```

One can define *modifications* that are performed on `Printer`:

```
trait Emphasizer extends Printer:
  abstract override def print(msg: String) =
    super.print("<em>" + msg + "</em>")

trait Boldifier extends Printer:
  abstract override def print(msg: String) =
    super.print("<b>" + msg + "</b>")
```

  - `Emphasizer` and `Boldifier` modify a concrete `Printer` class rather than implementing a full `Printer` class.

Traits: Stackable Modifications (2)

- Any of these two modifications can be mixed into a class:

```
class PrettyPrinter extends ConsolePrinter
  with Emphasizer with Boldifier
```

```
val printer1 = new PrettyPrinter
printer1.print("test") // → <em><b>test</b></em>
```

- Traits can also be mixed in when objects are created:

```
val printer2 = new ConsolePrinter with Emphasizer with Boldifier
printer2.print("test") // → <em><b>test</b></em>
```

- The order in which the traits are specified is significant:

```
val printer3 = new ConsolePrinter with Boldifier with Emphasizer
printer3.print("test") // → <b><em>test</em></b>
```

  - The order depends on the linearization: from right to left (simplified).
- Stackable Modifications can be used to implement the decorator pattern in a simple way.

Self-Type Annotation

- If a class contains a *self-type annotation*:

```
trait Emphasizer extends Printer:
  def emphasize(msg: String) = "<em>" + msg + "</em>"
```

```
class PrettyPrinter extends ConsolePrinter
  self: Emphasizer =>
  override def print(msg: String) = super.print(self.emphasize(msg))
```

  it is enforced that instances of that class mix in the self type:

```
 val printer = new PrettyPrinter // → snytax error
 val printer = new PrettyPrinter with Emphasizer
```

- Frequently, in Scala this technique is used to implement *dependency injection* → *cake pattern*.

Type Parameterization

- Type parameterization allows you to write generic classes and traits.
- This concept corresponds to Java generics.
  - Scala allows no raw types → it is always required to specify type parameters.
- Example: Immutable Stack

```scala
class Stack[T](private val elems: List[T]):
  def this() = this(Nil)
  def isEmpty: Boolean = elems.isEmpty
  def push(elem: T): Stack[T] = new Stack(elem :: elems)
  def pop: Stack[T] = if (!isEmpty) new Stack(elems.tail) else …
  def top: T = if (!isEmpty) elems.head else …
```

```scala
val stack = new Stack[Int]
val item = stack.push(42).push(1).pop.top // item == 42
```

Type Variance

- Declaration of covariant and contravariant types:

```
class CovariantType[+T] { … }
```

```
class ContravariantType[-T] { … }
```

- It's also possible to specify lower (LB) and upper bounds (UB) for types:

```
T <: UB // T must be a subtype of UB
```

```
T :> LB // T must be a supertype of LB
```

- Example: Immutable stack

```
class Stack[+T](private val elems: List[T]):
  def this() = this(Nil)
  def isEmpty: Boolean = elems.isEmpty
  def push[U >: T](elem: U): Stack[U] = new Stack(elem :: elems)
  def pop: Stack[T] = ???
  def top: T = ???
```

```
val persStack: Stack[Person] = new Stack[Student]

val studStack = new Stack[Student]
val persStack2: Stack[Person] = studStack.push(new Person)
```