Introduction to Scala

Read-Eval-Print Loop

- Following the tradition of functional programming, the Scala programmer can interact with the language following the REPL
  - The interpreter repeatedly:
    - Read an expression
    - Evaluate the expression
    - Print the result
- In Scala you can also have (Java-like) compilation
  - Eg, for cloud deployment you will have to generate and upload a JAR file in the cloud

Expression Evaluation

- A non-primitive expression is evaluated as follows:
  - Take the leftmost operator
  - Evaluate its operands (from left to right)
  - Apply the operator to the operands
- A name is evaluated by replacing it with the right hand side of its definition
- The evaluation process stops once it results in a value (ie. a primitive expression like a number)
- Assuming `def radius = 10` and `def pi = 3.14159` :

```
(2 * pi) * radius  →
(2 * 3.14159) * radius  →
(6.28318) * radius  →
(6.28318) * 10  →
62.8318
```

Evaluation of Function Applications

- An application of a parameterized function is evaluated as follows:
  - Evaluate all function arguments, from left to right
  - Replace the function application by the function's right-hand side, and, at the same time..
  - ..replace the formal parameters of the function by the actual arguments

- Assuming `def square(x: Double) = x*x` and
  `def sumOfSquares(x: Double, y: Double)=square(x)+square(y)`:

```
sumOfSquares(3, 2+2) →  sumOfSquares(3, 4)   →
square(3)+square(4)  →  3*3+square(4)  →
9+square(4)          →  9+4*4  →
9+16                 →  25
```

Observations on evaluation

- Pay attention:
  - it is possible to have expressions with an evaluation that never terminates!

```
def loop: Int = loop+1
```

```
loop  →  loop+1   →
(loop+1)+1  →  ((loop+1)+1)+1    →
(((loop+1)+1)+1)+1   →  ((((loop+1)+1)+1)+1)+1  →
...
```

Evaluation strategies

- There exist alternative evaluation strategies for function application, in particular delay the evaluation of the arguments
  - Replace the formal parameters of the function by the corresponding expression in the function application
- For instance, in the sumOfSquares example we could alternatively proceed as follows:

```
sumOfSquares(3, 2+2) →  square(3)+square(2+2)   →
3*3+square(2+2)      →  9+square(2+2)  →
9+(2+2)*(2+2)        →  9+4*(2+2)  →
9+4*4                →  9+16     →    25
```

Call-by-value vs Call-by-name

- The first evaluation strategy is known as call-by-value (CBV), the second is known as call-by-name (CBN)
- Both strategies reduce to the same final values as long as
  - there are no side-effects (ie. the evaluation of an expression does not change the environment, ie. the name definitions)
  - both evaluations terminate
- CBV has the advantage that it evaluates every function argument only once
- CBN has the advantage that a function argument is not evaluated if the corresponding parameter is not used in the evaluation of the function body

Termination

- General result:
  - Given an expression *e* without side effects, if its evaluation following CBV terminates, then also the evaluation following CBN terminates
- The vice versa does not hold:
  there are expressions whose evaluation terminates under CBN but not under CBV
  - Exercise: define an expression with this property

Termination

- General result:
  - Given an expression *e* without side effects, if its evaluation following CBV terminates, then also the evaluation following CBN terminates
- The vice versa does not hold:
  there are expressions whose evaluation terminates under CBN but not under CBV
  - Exercise: define an expression with this property
- Assume **`def`** `loop:Int=loop+1` and **`def`** `first(x:Int, y:Int)=x`
  consider the expression: `first(1,loop)`
    - The evaluation terminates under CBN but not under CBV
  - In Scala, the default is CBV. A CBN parameter is denoted by adding => in front of the parameter type:

```
def first(x:Int, y: => Int)=x
```

Conditional expressions

- Alternatives can be expressed with conditional expressions (using an if-else syntax):

```
def abs (x:Int) = if (x >= 0) x else -x
```

  where `(x >= 0)` is an expression of type Boolean
- Boolean expressions are like in Java (including “short-circuit” evaluation for || and &&)

Value definitions

- The “def” definitions we have seen so far are like CBN (the r.h.s. is not evaluated at “def” time)
- Alternatively, there are “val” definitions:

```
val x=2
val y=square(x)
```

  in which the r.h.s. is evaluated and the resulting value is associated to the defined name

  - Exercise: present a definition for which using “def” or “val” makes an observable difference

Value definitions

- The “def” definitions we have seen so far are like CBN (the r.h.s. is not evaluated at “def” time)
- Alternatively, there are “val” definitions:

```
val x=2
val y=square(x)
```

  in which the r.h.s. is evaluated and the resulting value is associated to the defined name

  - Exercise: present a definition for which using “def” or “val” makes an observable difference

    ```
    def x = loop
    val x = loop
    ```

Exercise

- Define two functions “and” and “or” equivalent to && and ||
  (including the short-circuit evaluation)

Exercise

- Define two functions “and” and “or” equivalent to && and || (including the short-circuit evaluation)

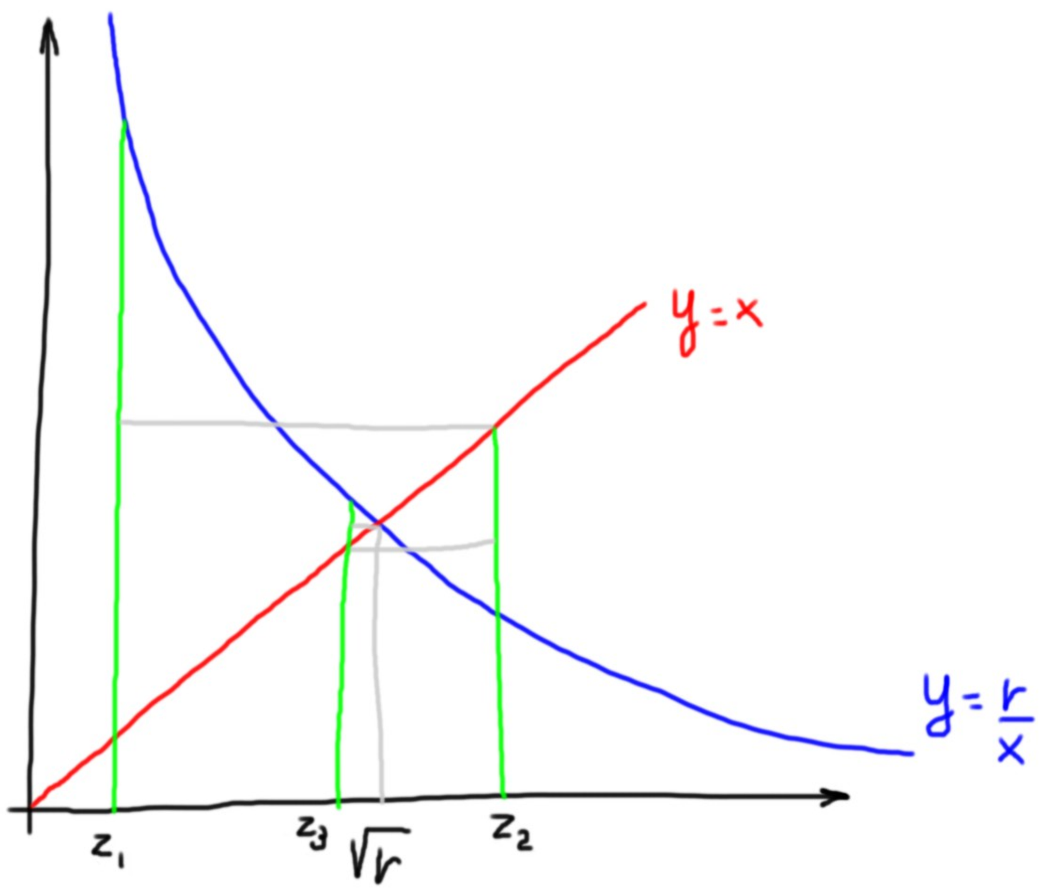
```
def and(x:Boolean, y: => Boolean) =
  if (x) y else false

def or(x:Boolean, y: => Boolean) =
  if (x) x else y
```

Functions

Square root with Newton's method

- To compute `sqrt(r)`
  - Start with an initial estimate `z` (any positive number e.g. 1)
  - Repeatedly improve the estimate taking the mean of `z` and `r/z`



Square root with Newton's method

- To compute `sqrt(r)`
  - Start with an initial estimate `z` (any positive number e.g. 1)
  - Repeatedly improve the estimate taking the mean of `z` and `r/z`

```
def sqrt(z: Double) = {

  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double, x: Double) =
    Math.abs(guess * guess - x)/x  < 0.001

  sqrtIter(1.0, z)
}
```

Blocks and scoping rules

- Curly braces are used to define blocks
  - It contains a sequence of definitions visible only inside the block
    - Such definitions shadow definitions of the same name outside the block
  - Last element is an expression that defines the value of the block
    - Blocks are themselves expressions
  - Use of standard static (lexical) scoping rules

```
val x = 10
def f(y: Int) = x
val result = {
  val x = 20
  x * f(x)
} + x
```

Efficiency

- One could argue that using recursion (instead of loops) is less efficient due to overhead in stack management
  - Scala optimizes execution in case of tail recursion (as in the square root example)
  - Exercise: write a tail recursive version of factorial

Efficiency

- One could argue that using recursion (instead of loops) is less efficient due to overhead in stack management
  - Scala optimizes execution in case of tail recursion (as in the square root example)
  - Exercise: write a tail recursive version of factorial

```
def factorial(x: Int) = {

  def loop(n:Int, acc:Int):Int =
    if (n==0) acc
    else loop(n-1, n * acc)

  loop(x,1)
}
```

Higher-Order Functions

Higher-order functions

- Functional languages treat functions as first-class values
- This means that, like any other value, a function can be passed as a parameter and returned as a result
  - This provides a flexible way to compose programs
- Functions with other functions as parameters or returning functions as results are called higher order functions
- Consider, for instance, the math notation $\sum_{n=a}^{b} f(n)$
  It is an example of a higher-order function:

```
def sum(f:Int => Int, a: Int, b:Int): Int =
  if (a>b) 0 else f(a)+sum(f, a+1, b)
```

Type and denotation of functions

- The type A => B is the type of a function that takes an argument of type A and returns a result of type B
  - So, Int => Int is the type of functions that map integers to integers
- It is possible to denote a function without defining it with an associated name:

```
x => x * x * x
```

These functions are called anonymous functions

Returning a function

- Consider:

```
def sumInts(a: Int, b:Int) = sum( x => x, a, b)
def sumCubes(a: Int, b:Int) = sum( x => x * x * x, a, b)
def sumFactorials(a: Int, b:Int) = sum( fact, a, b)
```

  there is a useless repetition of the parameters a and b

- Consider the following alternative definition of sum

```
def sum(f:Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int =
    if (a>b) 0 else f(a) + sumF(a+1, b)

  sumF
}
```

- We can write: `def sumInts = sum(x => x), def sumCubes = sum( x => x * x * x ), def sumFactorials = sum( fact )`

Returning a function

- Consider:

```
def sumInts(a: Int, b:Int) = sum( x => x, a, b)
def sumCubes(a: Int, b:Int) = sum( x => x * x * x, a, b)
def sumFactorials(a: Int, b:Int) = sum( fact, a, b)
```

  there is a useless repetition of the parameters a and b

- The same result of previous slide can be obtained as follows:

```
def currySum(f:Int => Int)(a: Int, b: Int): Int =
  if (a>b) 0 else f(a) + currySum(f)(a+1, b)

def sum(f:Int => Int) = currySum(f)(_,_)
```

- We can write: `def sumInts = sum(x => x), def sumCubes = sum( x => x * x * x ), def sumFactorials = sum( fact )`

Exercise

- Write a product function and a generalization of both product and sum

```
def product(f:Int => Int): (Int, Int) => Int ={
  def prodF(a: Int, b: Int): Int =
    if (a>b) 1 else f(a) * prodF(a+1, b)

  prodF
}
```

Exercise

- Write a product function and a generalization of both product and sum

```
def product(f:Int => Int): (Int, Int) => Int ={
  def prodF(a: Int, b: Int): Int =
    if (a>b) 1 else f(a) * prodF(a+1, b)

  prodF
}
```

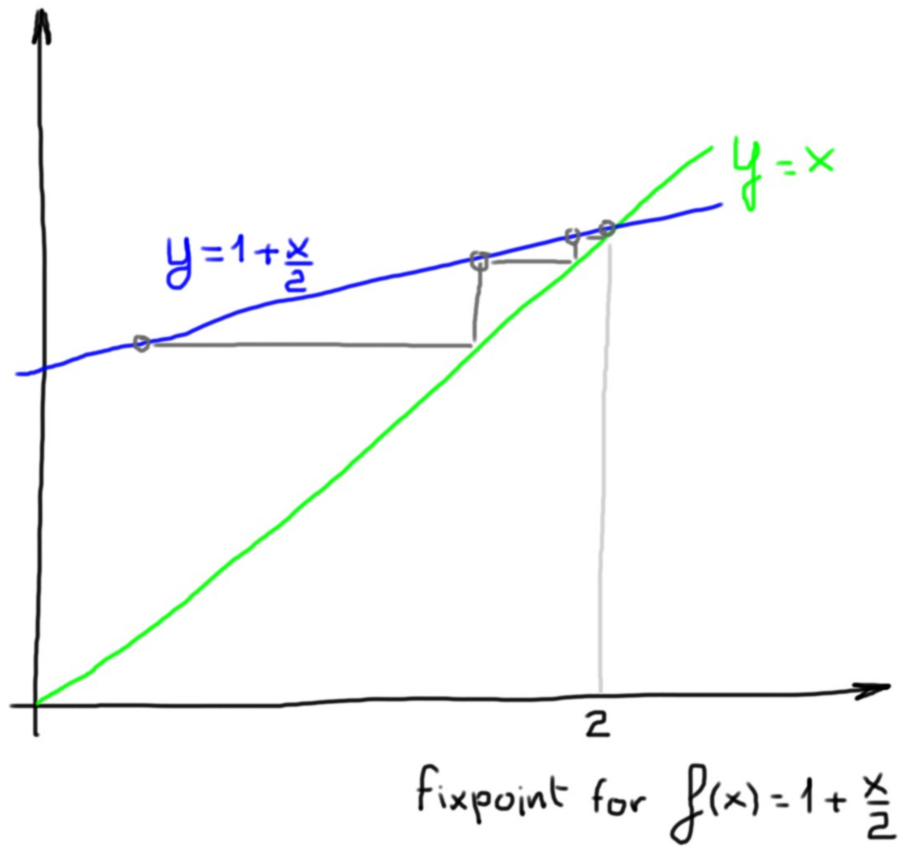
```
def mapReduce(f:Int => Int, combine:(Int, Int) => Int, zero: Int)
             (a: Int, b: Int): Int =
  if (a>b) zero else combine(f(a), mapReduce(f,combine,zero)(a+1, b))
```

with mapReduce, one can define eg.:

```
def fact(n: Int) = mapReduce(x => x, (x,y) => x*y, 1)(1,n)
```

Fixed-Point computation

- A fixed point for a function f(), is a value x such that x=f(x)
- For (some class of) functions a fixed-point can be computed starting from (some) initial guess z, and compute f(f(..(f(z))..))



Fixed-Point computation

- A fixed point for a function f(), is a value x such that x=f(x)
- For (some class of) functions a fixed-point can be computed starting from (some) initial guess z, and compute f(f(..(f(z))..))

```
import Math.abs
def fixPoint(f: Double => Double) = {
  val tolerance = 0.00001
  def isCloseEnough(x: Double, y: Double) =
    abs((x - y) / x) < tolerance
  def iterate(guess: Double): Double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(1.0)
}
```

Square root as a fixed-point

- Is it possible to compute square root as a fixed point?
  - Yes: sqrt(x)=y if $y \times y = x$ which correponds to $y = x/y$
  - Hence, given x we can compute sqrt(x) as a fixed point of the function $f(y) = x/y$

Square root as a fixed-point

- Is it possible to compute square root as a fixed point?
  - Yes: sqrt(x)=y if $y \times y = x$ which correponds to $y = x/y$
  - Hence, given x we can compute sqrt(x) as a fixed point of the function $f(y) = x/y$

```
def sqrt(x: Double) = fixPoint(y => x / y)
```

Square root as a fixed-point

- Is it possible to compute square root as a fixed point?
  - Yes: sqrt(x)=y if y×y = x which correponds to y = x/y
  - Hence, given x we can compute sqrt(x) as a fixed point of the function f(y) = x/y

```
def sqrt(x: Double) = fixPoint(y => x / y)
```

- Problem: the computation diverges
  - Solution: average successive values

```
def average(f: Double => Double)(x: Double) =
  (x+f(x))/2

def sqrt(x: Double) = fixPoint(average(y => x/y)(_))
```

  - Notice: `average(y => x/y)(_)` returns a function

Classes

Modeling data

- In Scala, data structures are defined as classes
- We introduce class definitions considering the rational data type example
  - A rational is a pair of integers representing the numerator and the denominator, respectively

```
class Rational (val x: Int, val y: Int) {
  def add(r: Rational) = new Rational(
    x * r.y + r.x * y, y * r.y)
  def neg = new Rational(-x, y)
  def sub(r: Rational) = add(r.neg)
}
```

  - **val**: implicitly generates an accessible field with that name/type

Private methods and this

- We can improve the class `Rational` as follows:

```
class Rational (x: Int, y: Int) {
  private def gcd(a: Int, b : Int): Int =
    if (b == 0) a
    else gcd(b, a % b)
  val numer = x  / gcd(x,y)
  val denom = y  / gcd(x,y)
  def add(r: Rational) = new Rational(
    numer * r.denom + r.numer * denom,
    denom * r.denom)
  def neg = new Rational(-numer, denom)
  def sub(r: Rational) = add(r.neg)
  def less(r: Rational) =
    numer * r.denom < r.numer * denom
  def max(r: Rational) =
    if  (this.less(r)) r
    else this
}
```

Infix notation and operators

- In Scala it is possible to use infix notation for unary methods
  - `r add s` is equivalent to `r.add(s)`
  - `r less s` is equivalent to `r.less(s)`
- Methods identifier can be also operators, e.g. `add` can be named `+` and `less` can be named `<`

```
…
def + (r: Rational) = new Rational(
  numer * r.denom + r.numer * denom,
  denom * r.denom)
def - (r: Rational) = this + (r.neg)
def < (r: Rational) =
  numer * r.denom < r.numer * denom
…
r1 - r2; r1 < r2
```

Class hierarchies

- Scala has abstract classes (with undefined methods) and support class extension

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}

class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if  (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmpty(elem, left incl x, right)
    else if (x > elem) new NonEmpty(elem, left, right incl x)
    else this
}
```

Overriding

- To override already defined methods, it is necessary to explicitly add the `override` keyword

```
abstract class Base {
  def foo = 1
  def bar: Int
}

class Sub extends Base {
  override def foo = 2
  def bar = 3
}
```

Object definition

- In the integer set example, it is useless to have many instances of the class `Empty`
  - It is possible to define it as an object instead of a class (thus defining a singleton object that evaluates to itself)

```
object Empty extends IntSet {
  def contains(x: Int): Boolean = false

  def incl(x: Int): IntSet =
    new NonEmpty(x, Empty, Empty)
}
```

Dynamic dispatch

- Scala follows the dynamic dispatch/binding approach:
  - When a method is invoked on an object, the implementation in the class of the object is considered (hence considering the dynamic type and not the static type)
  - Example of dynamic dyspatch:

```
abstract class IntSet {
... override def toString = "noImplementation"
}
object Empty extends IntSet {
... override def toString = "."
}
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
... override def toString =
      "{" + left.toString + elem.toString + right.toString + "}"
}

val set:IntSet = new NonEmpty(3,Empty,Empty)
set.toString
```

Run Scala programs

- Objects with the special `main(args: Array[String])` method can be compiled and executed (not only interpreted in the REPL or in worksheets)
  - Using sbt it is sufficient to
    - create a directory
    - save the object definition in a .scala file
    - execute sbt in the directory and then use the "run <fileName>" command

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

Traits

Traits

- In Scala a class can have only one superclass (single inheritance) but it can inherit code from more than one trait
  - A trait can be seen as a “rich interface”
  - It is defined as an abstract class but without initialization parameters
  - Classes that inherits from traits, list them putting the keyword with in front of the traits
    - The order matters: the last trait is considered first in case of traits that override methods
  - Like for interfaces, also traits define types

(An example of usage of traits is in the next slide)

Traits - example

```
class Rational(x: Int, y: Int) {
  val numer = x; val denom = y
  override def toString = numer + "/" + denom
}

class Segment(x: Int, y: Int) {
  val x_axis = x; val y_axis = y
  override def toString = "<" + x_axis + "," + y_axis + ">"
}

trait TotOrder[T] {
  def compare (r: T): Double
  def > (r: T) = (this compare r) > 0
  def < (r: T) = (this compare r) < 0
  def >= (r: T) = !(this < r)
  def <= (r: T) = !(this > r)
}

class OrdRat(x: Int, y: Int) extends Rational(x, y) with TotOrder[OrdRat]  {
  def compare (r: OrdRat) = (numer * r.denom - r.numer * denom)
}

class OrdSeg (x: Int, y: Int) extends Segment(x, y) with TotOrder[OrdSeg]  {
  def compare (r: OrdSeg) =
    Math.sqrt(Math.pow(x_axis,2) + Math.pow(y_axis,2)) -
      Math.sqrt(Math.pow(r.x_axis,2) + Math.pow(r.y_axis,2))
}
```

Dynamic binding of super

- Another specific aspect of traits is the dynamic binding of `super`
  - The keyword `super` refers to the superclass
  - Inside a trait cannot be statically interpreted because the trait can be mixed-in different classes, with different superclasses
    - it is dynamically bound to the superclass of the mixed-in class

```
trait TotOrder[T] {
  ...
  def print() = println("super.toString inside trait: "
                        +super.toString)
}
...
new OrdRat(2,3).print()
  //super.toString inside trait: 2/3
new OrdSeg(2,3).print()
  //super.toString inside trait: <2,3>
```

Scala is “scalable”

- It is easy to extend the language with new primitives and mechanisms that resemble native
- As an example, we will redefine booleans (as it was a new data type)

```
trait Bool {
  def ifThenElse(t: => Bool, e: => Bool): Bool
  def && (x: => Bool): Bool = ifThenElse(x, ff)
  def || (x: => Bool): Bool = ifThenElse(tt, x)
  def not: Bool              = ifThenElse(ff, tt)
}
object tt extends Bool {
  def ifThenElse (t: => Bool, e: => Bool) = t
  override def toString = "tt"
}
object ff extends Bool {
  def ifThenElse (t: => Bool, e: => Bool) = e
  override def toString = "ff"
}
```

Scala is “scalable”

- It is easy to extend the language with new primitives and mechanisms that resemble native
- As an example, we will redefine booleans (as it was a new data type)

```scala
val boolVar: Bool = tt

val otherBoolVar: Bool = tt.not

val testOr = boolVar || otherBoolVar

val testAnd = boolVar && otherBoolVar

val testIf1 = boolVar.ifThenElse(tt,ff)

val testIf2 = otherBoolVar.ifThenElse(tt,ff)
```

Exercise

- Add the binary operators == and != to the Bool type

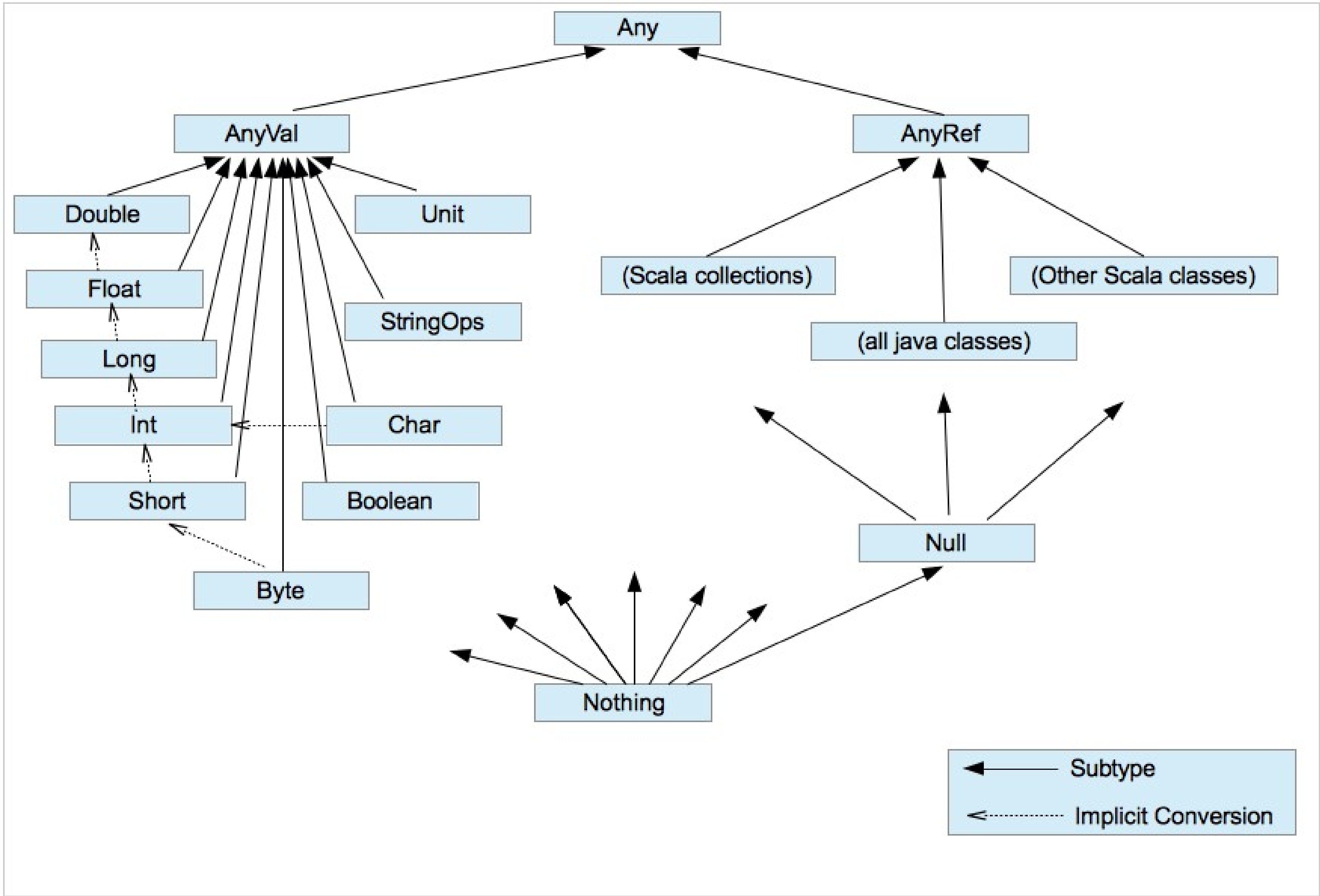
Exercise

- Add the binary operators == and != to the Bool type

```
def == (x: Bool): Bool = ifThenElse(x, x.not)

def != (x: Bool): Bool = ifThenElse(x.not, x)
```

Scala built-in types hierarchy



Top and Bottom types

- `Any`:
  - Base type of all types (eg methods ==, !=, equals, toString)
- `AnyRef`:
  - Base type of all reference types (alias of java.lang.Object)
- `AnyVal`:
  - Base type of all primitive types
- `Nothing`:
  - It is a subtype of every other type (no value of type Nothing)
  - Used to type abnormal termination (it is the type of `throw Exc`)
- `Null`:
  - It is the type of `null` (Null is a subtype of all reference types as `null` can be assigned to variables of these types)

Generic classes

Generic classes

- Assume to define lists of integers
  - following the traditional Cons-list implementation

```
trait IntList {
  def isEmpty: Boolean
  def head: Int
  def tail: IntList
}

class Cons(val head: Int, val tail: IntList) extends IntList {
  def isEmpty = false
}

object Nil extends IntList {
  def isEmpty = true
  def head = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")
}
```

Generic classes (continue)

- We can easily define generic lists
  - That can contain a parametric type T
  - Also generic functions can be defined

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: T, val tail: List[T]) extends List[T] {
  def isEmpty = false
}

class Nil[T] extends List[T] {
  def isEmpty = true
  def head = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")
}

def singleton[T](elem:T) = new Cons[T](elem, new Nil[T])
```

Generic classes (continue)

- Nil is no longer an object!
  - If we use object Nil instead of class Nil[T] we have an error because List[T] needs a previous declaration of the type parameter T
  - We will see how to rewrite generic List in such a way that Nil is a unique object

```
...
class Nil[T] extends List[T] {
  def isEmpty = true
  def head = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")
}
...
```

Exercise

- Write a function nth that given a list l and an integer n
  - Return the n-th element of l, or throws an IndexOutOfBoundsException in case l has length smaller than n

Exercise

- Write a function nth that given a list l and an integer n
  - Return the n-th element of l, or throws an IndexOutOfBoundsException in case l has length smaller than n

```
def nth[T] (l:List[T], n: Int): T = {
  if (l.isEmpty) throw new IndexOutOfBoundsException
  if (n==0) l.head
  else nth(l.tail, n-1)
}
```

Polymorphism

- We have seen so far two kinds of polymorphism:
  - Subtyping:
    - instances of subclasses are used where objects of superclasses are expected
  - Generics:
    - To express parameterized types (i.e. types that include type parameters)
- We now investigate the interplay among them:
  - Type bounds:
    - Type parameters that can range within limited type intervals
  - Variance:
    - Subtyping rules for parameterized types

Type bounds

- Consider the function id that, given an integer list, return the same list (considering the previous IntList class)
- How to declare this function?

```
def id(l: IntList): IntList
```

  – Ok, but if s is Empty, then an Empty is returned; if s is nonEmpty then a nonEmpty is returned
- We can then be more precise as follows:

```
def id[T <: IntList](s: T): T
```

- The notation `T <: IntList` introduces the type parameter T imposing IntList as an upper bound
  – In this case T can be Nothing, Null, Nil.type, Cons, IntList
- It is possible to express also lower bounds:

  `T >: Cons` or even `T >: Cons <: IntList`

  (an example of their use will follow)

Variance

Covariance

- We have that Cons <: IntList
  - Is it reasonable to assume List[Cons] <: List[IntList] ?
  - In principle this is fine:
    - We can pass to every program that expects a List[IntList] a List[Cons] keeping the program correct
- This interpretation of subtyping follows from the so-called Liskov substitution principle:
  - Let q(x) be a property provable about objects x of type B. Then q(y) should be provable for objects y of type A where A <: B.
- Parameterized types C[T] (like List[T] above) for which we have that A <: B implies C[A] <: C[B] are called covariant

Covariance pitfalls

- Attention: it is not always safe to consider covariant data structure (like eg covariant arrays)
  - Example: Java compiler accepts the following code

```
public class Arrays {
  public static void main(String arg[]) {
    Derived[ ] arr1 = new Derived[10];
    Base[ ] arr2 = arr1;
    arr2[0] = new Base();
    Derived o = arr1[0];
    o.g(); //method g is in Derived but not in Base
  }
}
```

  (but execution throws an exception)
  - Problem: covariance is not safe with mutable data structures (in Scala: Array[Cons] is not subtype of Array[IntList])

Declaring variant classes

- In Scala it is possible to declare variant classes

```
class C[+A]{...}  // C is covariant
class C[-A]{...}  // C is contravariant
class C[A]{...}   // C is nonvariant
```

  - In the first case we have C[Cons] <: C[IntList]
  - In the second case we have C[IntList] <: C[Cons]
  - In the third case we have neither C[Cons] <: C[IntList] nor C[IntList] <: C[Cons]

Example: the Function trait

- In Scala f(x) is a macro for f apply x where apply is declared in package scala as follows:

```
trait Function1[-T, +U]
{
  def apply(x: T): U
}
```

- This means what follows:
  - Let `f` be of type `Function1[Df,Cf]` and `g` of type `Function1[Dg,Cg]`. Then `f` is subtype of `g` iff:
    - `Df >: Dg` (contravariance on the domain)
    - `Cf <: Cg` (covariance on the codomain)

Rationale behind function variance

- Consider:

```
def useFunction(f: A=>B) = {
  val x: B = f(new A())
  x
}
```

  - Consider `useFunction(g)`. Which constraints should be satisfied by `g`?
    - `g` should accept in input objects of type `A` (ie. its domain should be a supertype of `A`)
      - This means contravariance on the domain
    - `g` should return an object that can be stored in `x` (ie. its codomain should be a subtype of `B`)
      - This means covariance on the codomain

Reconsider the Function trait

```
trait Function1[-T, +U]
{
  def apply(x: T): U
}
```

- Scala checks whether types T and U are in wrong positions:
  - Type T (declared contravariant) cannot appear as a return type (and cannot appear as type lower bound)
  - Type U (declared covariant) cannot appear as a parameter type (and cannot appear as type upper bound)

Covariant lists

- We have already observed that it is safe to declare lists as covariant (because lists are immutable):

```
trait List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
```

- Exercise:
  - Can we add the following prepend method?

```
  def prepend(e:T): List[T]
```

Covariant lists

- We have already observed that it is safe to declare lists as covariant (because lists are immutable):

```
trait List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}
```

- Exercise:
  - Can we add the following prepend method?

```
  def prepend(e:T): List[T]
```

  - No! It must be declared as follows:

```
  def prepend[U>:T](e:U): List[U]
```

Exercise

- Remember, we did not declare Nil as an object, but as a class Nil[T] to have a binder for the type parameter T

```
class Nil[T] extends List[T] {
  ...
}
```

- Assuming list is covariant (as in previous slide), we can declare Nil as an object, how?

Exercise

- Remember, we did not declare Nil as an object, but as a class Nil[T] to have a binder for the type parameter T

```
class Nil[T] extends List[T] {
  ...
}
```

- Assuming list is covariant (as in previous slide), we can declare Nil as an object, how?

```
object Nil extends List[Nothing] {
  ...
}
```

Pattern Matching

Pattern matching

- Pattern matching can be used to decompose objects, i.e. inspect their structure
- Example: expression evaluation

```
trait Expr {
  def eval: Int = this match {
    case Number(n) => n
    case Sum(l,r) => l.eval + r.eval
    case Prod(l,r) => l.eval * r.eval
  }
}

case class Number(n: Int) extends Expr

case class Sum(e1: Expr, e2: Expr) extends Expr

case class Prod(e1: Expr, e2: Expr) extends Expr
```

Pattern matching: syntax

e match { case p_1 => e_1 … case p_n => e_n }

- e expression to be decomposed
- $p_1 ... p_n$ patterns
- $e_1 ... e_n$ expressions to be evaluated in case of matching of the corresponding pattern (considered according to the order in which they appear)

- The patterns contain
  - constructors used to create the objects to be decomposed
  - variables that are bound to the matching subexpressions
  - wildcards for subexpressions that are irrelevant
  - constants like 1 or true

Pattern matching: evaluation

- The whole match expression is rewritten to the r.h.s. of the first case where pattern matches the selector e
  - Pattern variables are replaced by the corresponding subexpressions in the selector

```
eval(Sum(Number(1), Number(2)))
→
Sum(Number(1), Number(2)) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
→
eval(Number(1)) + eval(Number(2))
→
```

Pattern matching: evaluation

- The whole match expression is rewritten to the r.h.s. of the first case where pattern matches the selector e
  - Pattern variables are replaced by the corresponding subexpressions in the selector

```
Number(1) match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
} + eval(Number(2))
→
1 + eval(Number(2))
→...→
1 + 2
→
3
```

Exercise

- Add a method show to the Expr trait to print the expression
  - add the minimum amount of disambiguating parentheses

Exercise

- Add a method show to the Expr trait to print the expression
  - add the minimum amount of disambiguating parentheses

```
trait Expr {
  ...
  def show: String = this match {

    ...

  }
  ...
}
```

Exercise

- Add a method show to the Expr trait to print the expression
  - add the minimum amount of disambiguating parentheses

```
trait Expr {
  ...
  def show: String = this match {
    case Number(n) => n.toString
    case Sum(l,r) => l.show + " + " + r.show
    case Prod(l,r) => addPar(l) + " * " + addPar(r)
  }
  ...
}

def addPar(e:Expr) = e match {
  case Sum(_,_) => "("+e.show+")"
  case _ => e.show
}
```