# Basics of Functional Programming



## Tail Recursion

A recursive call is said to be in **tail position** if the caller does nothing other than return the value of the recursive call.

**A recursion with a call in tail position** is called the **tail recursion**.

Is this tail recursion?

```
def factorial(n: Int): Int =
  if n <= 0 then 1
  else
    n * factorial(n - 1)
```

# Tail Call Elimination

If all recursive calls made by a function are in tail position, Scala automatically compiles the recursion to iterative loops.

```
def factorial(n: Int): Int =
  @annotation.tailrec
  def go(n: Int, acc: Int): Int =
    if n <= 0 then acc
    else go(n-1, n * acc)
  go(n, 1)
```

```
def factorial(n: Int): Int =
  var acc = 1
  var i = n
  while i > 0 do { acc *= i; i -= 1 }
  acc
```



# Exercise

Write a recursive function to get the *n*th **Fibonacci** number. The first two Fibonacci numbers are 0 and 1. The *n*th number is always the sum of the previous two—the sequence begins 0, 1, 1, 2, 3, 5.

Your definition should use a local tail-recursive function.

```
def fib(n: Int): Int = ???
```

# Any Duplication?

```
def abs(n: Int): Int = if n < 0 then -n else n

def formatAbs(x: Int) =
  val msg = "The absolute value of %d is %d"
  msg.format(x, abs(x))
```

```
def factorial(n: Int): Int = { … }

def formatFactorial(n: Int) =
  val msg = "The factorial of %d is %d."
  msg.format(n, factorial(n))
```



# DRY – Don’t Repeat Yourself

```
// We can generalize `formatAbs` and `formatFactorial` to
// accept a _function_ as a parameter
def formatResult(name: String, n: Int, f: Int => Int) =
  val msg = "The %s of %d is %d."
  msg.format(name, n, f(n))


// Now we can use our general `formatResult` function
// with both `abs` and `factorial`
def main(args: Array[String]): Unit =
  println(formatResult("absolute value", -42, abs))
  println(formatResult("factorial", 7, factorial))
```

# Higher-Order Functions (HOFs)

- Functions are **values**.
- Functions can be assigned to variables, stored in data structures, and passed as arguments to functions.
- **HOFs** are ones that **accept functions as parameters,** and/or **return functions as return values.**



# Monomorphic Functions

Functions that operate on **only one type of data**.

**abs** and **factorial** are specific to arguments of type **Int**, and the higher-order function **formatResult** is also fixed to operate on functions that take arguments of type **Int**.

```
def findFirst(ss: Array[String], key: String): Int =
  @annotation.tailrec
  def loop(n: Int): Int =
    if n >= ss.length then -1
    else if ss(n) == key then n
    else loop(n + 1)

  loop(0)
```

# Polymorphic Functions (aka, Generic Functions)

**Abstracting over the type**

Functions that works for **any** type it's given

**Parametric polymorphism**

Type parameters/Type variables



# Polymorphic Functions

```
def findFirst[A](as: Array[A], p: A => Boolean): Int =
  @annotation.tailrec
  def loop(n: Int): Int =
    if n >= as.length then -1
    else if p(as(n)) then n
    else loop(n + 1)

  loop(0)
```

Instead of hardcoding `String`, take a type `A` as a parameter. And instead of hardcoding an equality check for a given key, take a function with which to test each element of the array.

If the function `p` matches the current element, we've found a match and we return its index in the array.

# Exercise

Implement **isSorted**, which checks whether an **Array[A]** is sorted according to a given comparison function:

```
def isSorted[A](as: Array[A], ordered: (A,A) => Boolean): Boolean
```



# Anonymous Functions (aka Function Literals)

```
scala> findFirst(Array(7, 9, 13), (x: Int) => x == 9)
res2: Int = 1
```

**(x: Int) => x == 9** is a **function literal** or **anonymous function**.

In general, the arguments to the function are declared to the left of the => arrow, and we can then use them in the body of the function to the right of the arrow.

```
scala> (x: Int, y: Int) => x == y
res3: (Int, Int) => Boolean = <function2>
```

# Functions as Values in Scala

A function literal is actually being defined as an object with an **apply** method.

When we define **(a, b) => a < b**, this is really syntactic sugar:

```
val lessThan = new Function2[Int, Int, Boolean] {
  def apply(a: Int, b: Int) = a < b
}
```

**Function2[Int,Int,Boolean]** is usually written **(Int,Int) => Boolean**.

Objects that have an **apply** method can be called as if they were themselves methods.

```
lessThan.apply(10, 20) == lessThan(10, 20)
```



# Following Types to Implementations

The universe of possible implementations is significantly reduced when implementing a polymorphic function.

- Only one implementation might be possible!

**Example**: **partial application**

- The function is being applied to ***some but not all*** of the arguments it requires.

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C = ???
```

# Exercises

- **Currying** converts a function *f* of two arguments into a function of one argument that partially applies *f*. Write this implementation.

```
def curry[A,B,C](f: (A, B) => C): A => (B => C)
```

- Implement **uncurry**, which reverses the transformation of curry. Note that since => associates to the right, A => (B => C) can be written as A => B => C.

```
def uncurry[A,B,C](f: A => B => C): (A, B) => C
```

- Implement the higher-order function that composes two functions.

```
def compose[A,B,C](f: B => C, g: A => B): A => C
```