2.9 Going a bit deeper

- The next few slides are important!
- But don’t worry, we’ll be covering all of this stuff in *much greater detail* in the upcoming weeks.

Values, variables and expressions

- As mentioned previously, a functional program is essentially an ***expression*** which yields a ***value***:
  - That expression is ***expressed*** in terms of ***functions*** of values. Semi-formally (using *productions*):
    - `expression ::= term | function(term,term,…)`
    - `term ::= value | expression`
    - `value ::= constant | variable`
    - `variable ::= identifier`
- A ***variable*** is something which we use to simplify an expression by extracting a sub-expression and giving it an identifier with which to refer to it.
- Note that what we are used to thinking of as “operators” are just functions under another name (and with a different style of invocation);

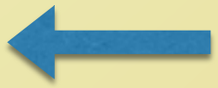
  `1 + 2` is equivalent to `1.+(2)` §
- Also note that in Scala, we also have classes and objects which complicate the structure noted above—but in pure functional programming, we just have expressions as described.
- What about keywords like ***if***, ***for, match, case,*** etc.? These are all “syntactic sugar” to define expressions.

§ is equivalent to `implicitly[Numeric[Int]].plus(1,2)`

Methods and functions

- A ***function*** is a value whose behavior is that it can transform a value of one type into a value of (maybe) another type:
  - so, for example, `Int=>String` is the type of a function which transforms an `Int` into a `String`
  - one way* to define a function is by declaring a variable with an appropriate type and provide an expression which says what the function does (where "_" is a placeholder for the value of the input to the function);
  - e.g.:

```
val f: Int=>Int = _*2
```



What do you think this function does?

  - A ***method*** is a property of a class or an object (it is not itself an object) which does the following:
    - it defines an identifier (so we can refer to the method);
    - it defines a list of input parameters (essentially, these are "variables" available to the function);
    - it defines a function (via the method body) in a convenient and readily understood format;
    - and—by convention, if the method belongs to a ***class*** (rather than an ***object***)—implicitly adds another parameter set: ***(this)***.
    - e.g.:

```
object MyMath { def double(x: Int): Int = x*2 }
```

* the other way is by defining a method

Types, values, etc.

- The strength of Scala rests to a large extent on its strict type safety. How do types, values and other properties relate to each other?
- Variables* all have six important aspects—which are shared knowledge between you and the compiler:
  - **name**: the *identifier* of the variable (i.e. how it gets referenced);
  - **type**: the *domain* the value belongs to, i.e. properties the variable supports—methods, range of legal values, etc.;
  - **scope**: *where* the variable can be referenced;
  - **mutability**: *whether* the value can be changed;
  - **value**: the *value* of the variable—(if mutable, then the *current* value);
  - **evaluation mechanism**: *how/when* the variable "evaluates" its value (if all functions are *pure* functions—idempotent—it won't matter when it gets evaluated).

**By "variable", a Scala programmer doesn't mean something that can change its value during a run. A variable is used in the sense of algebra—it's something that stands for some sort of quantity.*

A quick explanation: types

- Different styles of type:
  - **functions**: these can transform value(s) of one (or more) types into a value of some other type.

    e.g. `(x: Int) => x.toString`
  - **scalars**: ordinary value types such as *Int, Double, String, LocalDate*, etc.

    e.g. `3`
  - **containers**: wrappers around groups of values which may contain zero thru N members:
    - **longitudinal*** (collections): e.g. *Iterator, List, Array*, etc.

      e.g. `List("a","b","c")`
    - **transverse***: e.g. *Option, Tuple, Try, Future, Either*, etc.

      e.g. `Tuple("a",1,3.1415927)`
    - **hybrid***: e.g. *Map*, Seq[Tuple2[String,Int]] etc.

      e.g. `Map("a" -> 1, "b" -> 2)`

**These terms are not in common use: I use them to help differentiate different families of containers.*

A quick explanation: Scope

- Scope in Scala is similar (but not the same) as in Java.
- Example of legal code:

```
    val x = 3

def y = {
  val x = 5
  x + 8
}
```

This is called “shadowing”

The value of ***y*** is 13 (not 8).

Evaluation mechanism

- A variable or parameter can be ***evaluated*** in several ways, each by a different mechanism:
  - **direct reference (call-by-value)**: a variable (or method parameter) has a value and that value is effectively ***substituted*** for the variable/parameter wherever that variable/parameter is referenced.
    - `val x: X = expression`
    - `def y(x: X)`
  - **indirect reference (call-by-name)**: an indirect reference is like a reference via a pointer. We don't actually need to evaluate the pointer until we refer to it.
    - `def x = expression` which is equivalent to `val x: Unit=>Double = { _ => expression }`
    - `def y(x: => X)`
  - **lazy**: if a variable is lazily-evaluated, evaluation is ***deferred*** until it is needed. But unlike an indirect reference, which is evaluated each and every time it is referenced, a lazy variable is evaluated only when it is ***first*** referenced.

Exercise 1 - REPL

A. `def f(x: Int) = x*x`
B. `f(9)`
C. `def f(x: Int) = {println(x); x*x}`
D. `f(9)`
E. `val y = f(9)`
F. `lazy val z = f(9)`
G. `z + 19`
H. `f{println("hello"); 9}`
I. `def f(x: => Int) = x*x`
J. `f{println("hello"); 9}`
K. `def f(x: () => Int) = x()*x()`
L. `f{() => println("hello"); 9}`
M. `val g = {println(9); 9*9}`

Constructors & Extractors

- You're familiar with the idea of ***constructors*** such as:
  - `List(1,2,3)`
  - `Complex(1.0,-1.0)`
- But, surely, if you can <u>construct</u> objects, you ought to be able to "<u>destruct</u>" (or deconstruct/extract) them:

```
        case class Complex(real: Double, imag: Double)
        val z = Complex(1.0,-1.0)
z match {
  case Complex(r,i) => println(s"$r+i$i")
}


def show(l: List[Int]): String =
l match {
  case Nil => ""
  case h::t => s"$h,"+show(t)
}
```

- Yes you can! This is how pattern matching works.

Exercise 2 - REPL

A. `case class Complex(real: Double, imag: Double)`
B. `val z = Complex(1,0)`
C. `z match {case Complex(r,i) => println(s"$r i$i"); case _ => println(s"exception: $z") }`
D. `val l = List(1,2,3)`
E. `l match { case h :: t => println(s"head: $h; tail: $t"); case Nil => println(s"empty") }`