# Functions, methods and operators

# Functions, methods and operators

- These are, *more or less*, the same thing:
  - "method" is an aspect of object-oriented programming: essentially, it's a "field" of the class whose value happens to be a function rather than a value;
  - methods have a convenient syntax which allows you to name the parameters (in addition to specifying their types) and refer to those names in the body;
  - Classes and objects have methods of which their bodies define functions so…

```
scala> def increment(n: Int) = n+1
increment: (n: Int)Int
scala> increment(2)
res13: Int = 3
```

- which is equivalent to…

```
scala> val f1: Int=>Int = x=>x+1
f1: Int => Int = <function1>
scala> def increment(n: Int) = f1(n)
increment: (n: Int)Int
scala> increment(2)
res12: Int = 3
```

Equivalent to: `f1.apply(n)`

- but you can't do it this way…

```
scala> def increment(n: Int) = f1
increment: (n: Int)Int => Int
scala> increment(2)
res11: Int => Int = <function1>
```

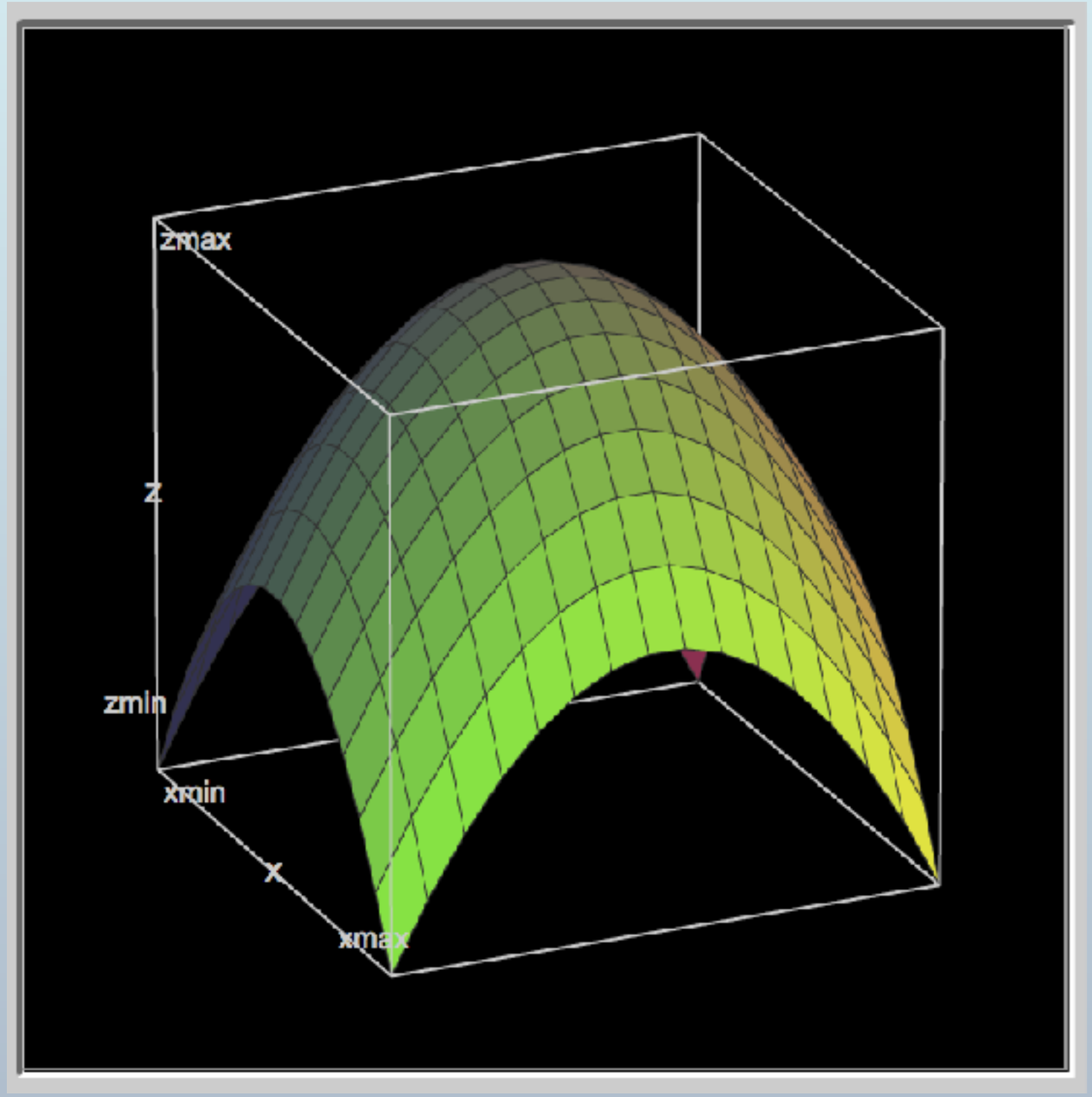
What's going on here?

# A little mathematics

- Suppose: $z = f(x,y)$
- and also that: $z' = f'(y)$
- if $z = z'$, for all $x$, $y$, then what does that tell us about the relationship of $f(x,y)$ and $f'(y)$?
- Obviously, $f'$ must itself be a function of $x$
- Let's say $f' = g(x)$
- In that case, $z' = g(x)(y) = f(x,y)$
- And so $f(x,y) = g(x)(y)$
- $g(x)(y)$ is called the "curried" version of $f(x,y)$

# Let's take a look…

- $z = f(x,y)$
- $f(x,y) = 2-x^2-y^2$
- $g(x)(y) = 2-x^2-y^2$
- $g(x) = (2-y^2)-x^2$
  - where y is a constant as far as the function *g* is concerned (also referred to as a "free" variable).
- Of course:
  - $h(y) = (2-x^2)-y^2$



# Partially-*applied* functions

- What if you leave off the parameter(s) of a Scala method?

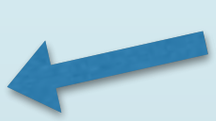
```
scala> def sqr(x: Int): Int = x*x
sqr: (x: Int)Int

scala> sqr
<console>:12: error: missing arguments for method sqr;
follow this method with `_' if you want to treat it as a partially applied function
       sqr
       ^

scala> sqr _
res0: Int => Int = <function1>

scala> res0(2)
res1: Int = 4

scala> def add(x: Int, y: Int) = x+y
add: (x: Int, y: Int)Int

scala> add _
res2: (Int, Int) => Int = <function2>

scala> res2(1,3)
res3: Int = 4

scala> val xs = Seq(1,2,3)
xs: Seq[Int] = List(1, 2, 3)

scala> xs map res0
res4: Seq[Int] = List(1, 4, 9)

scala> xs map sqr
res5: Seq[Int] = List(1, 4, 9)
```

This is a partially-applied function (indicated by "_"). Technically, this is the **η** expansion.

Huh? surely we can't pass a *method* as a parameter? Well, yes we can, where the context is clear. The compiler invokes the **η**-expansion for us.

# Partially-applied functions (2)

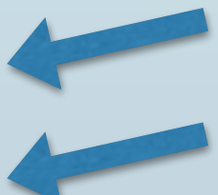
```
scala> val y = 1
y: Int = 1

scala> xs map { add(_,y) }
res6: Seq[Int] = List(2, 3, 4)

scala> val f: Int=>Int = y.+_
f: Int => Int = <function1>

scala> f(3)
res7: Int = 4

scala> xs map { f }
res8: Seq[Int] = List(2, 3, 4)

scala> xs map ( f )
res9: Seq[Int] = List(2, 3, 4)

scala> xs map f
res10: Seq[Int] = List(2, 3, 4)
```



This is called a “closure” because it “closes” on *y*

OK provided that we explicitly annotate type of *f.* This is also a closure.

apply *f* to 3.



map *xs* on *f.*



notice we can substitute parentheses for the braces

indeed we can dispense with the parentheses too

- What we’ve done here is to separate the two parameter “sets” of the *+* operator and close on one and bind the other to the values of *xs*. We haven’t been able to split parameter sets up (the compiler won’t allow it). In a moment, we’ll see how.

# Partially-applied functions (3)

- Let's try it now for a method where we explicitly use two parameter sets:

```
scala> def gMethod(x: Int)(y: Int) = x+y
gMethod: (x: Int)(y: Int)Int

scala> val g = gMethod _
g: Int => (Int => Int) = <function1>

scala> g(1)
res11: Int => Int = <function1>

scala> g(1)(2)
res12: Int = 3

scala> val h = g _
h: () => Int => (Int => Int) = <function0>

scala> h()(1)(2)
res13: Int = 3
```

This time when we partially apply the function we again get a function (*g*) but one which has an (unbound) parameter.

But now we can partially apply *that* function (*g*) and this time we get a function (*h*) which has no parameters but which yields a function.

# Partially *defined* functions

- In mathematics a function which is valid for all possible inputs is called a “total” function.
  - Contrarily, if a function only works for certain values, it is called a “partial” function (or partially-defined function).
  - For example, $cos^{-1}(x)$ is only defined for $-1<=x<=1$
  - Scala also has partially-defined functions. You’ve met some of them already (from week 1 - ticket agency):

```
while (state.availability) {
      receive {
        case sale: Sale => state = BoxOffice.makeTransaction(state,sale)
        case Status => sender ! state
        case _ => throw new Exception(“unknown message")
      }
    }
```

- The function which is provided as the body of receive is a partially-defined function: it only yields a valid result when the message received is of type *Sale* or *Status*.
- And of course our old friend *match.*

# Currying (1)

- This same notion of substituting a partially-applied function such that we end up with multiple parentheses is called Currying.
- No, it's not a culinary reference to delicious Indian food. Nor is it a reference to the Student Center at Northeastern. It is named after Haskell Curry (1900-1982). Wouldn't you feel cheated and disappointed if Scala **didn't** work this way?
- Actually, we can take an *uncurried* function definition and make it curried simply by calling the curried method:

```
scala> def f(x: Double, y: Double) = math.sqrt(x*x + y*y)
f: (x: Double, y: Double)Double

scala> val fDash = (f _).curried
fDash: Double => (Double => Double) = <function1>

scala> def g(x: Double)(y: Double) = math.sqrt(x*x + y*y)
g: (x: Double)(y: Double)Double

scala> val gDash = g _
gDash: Double => (Double => Double) = <function1>
```

*fDash* is a function which takes a *Double* and returns a function which converts a *Double* into a *Double*

*gDash* has the same type as *fDash*

# Currying (2)

- Let's say we have the following matrix:

```
scala> val matrix = List(List(1,2,3),List(2,3,1),List(3,1,2))
matrix: List[List[Int]] = List(List(1, 2, 3), List(2, 3, 1), List(3, 1, 2))
```

  - We can access an element of the matrix using this method:

```
scala> def element(r: Int)(c: Int) = matrix(r)(c)
element: (r: Int)(c: Int)Int
```

  - In general, if we have a function with multiple parameter sets:
    - `def f(args1)…(argsN-1)(argsN) = E` ← *E* is some expression
- then we can also write:
  - `def f(args1)…(argsN-1) = { def g(argsN) = E; g }` ← *g* is an arbitrary identifier
  - `def f(args1)…(argsN-1) = ( argsN => E )` ← which simplifies to

```
scala> def row(r: Int) = element(r) _
row: (r: Int)Int => Int

scala> val row1 = row(1)
row1: Int => Int = <function1>

scala> row1(2)
res17: Int = 1
```

# Currying (3)

- You may already have noticed that Scala methods allow for several parameter lists, as in the example from the RNG assignment (method meanU).
  - Or, take a look at this from the Scala Documentation:

```
object Currying extends App {

  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

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

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

The *App* trait basically wraps your code in a *main* program, providing you with *args*.

*modN* has two parameter lists, in this case each of them has just one parameter.

Here *modN* is given only one parameter list! What's a poor method to do?

- Well, we recognize this as a "partially-applied-function", except that here, context is unambiguous and we don't need to provide "_" to confirm that we want a PAF.
- Look at *modN* (which you may describe as a "curried" function) in the REPL (you will want to use "paste mode" to enter the whole program):

```
modN: (n: Int)(x: Int)Boolean
```

- But *modN(2)* is a *Int=>Boolean* so in order for *modN(2)* to yield a result of *Boolean*, it must be given another *Int*. That's exactly what the context of *filter* does: it successively applies the *Int* elements of *nums* to *modN(2)* yielding *true* if the element is even.

# Lambdas, also known as anonymous functions; function literals

- Like in Java, we can use anonymous functions

```
scala> val list = List(1,2,3)
list: List[Int] = List(1, 2, 3)
scala> list map (_.toString)
res0: List[String] = List(1, 2, 3)
scala> list map {_.toString}
res1: List[String] = List(1, 2, 3)
scala> list map {x => x.toString}
res2: List[String] = List(1, 2, 3)
scala> list map ((x: Int) => x.toString)
res3: List[String] = List(1, 2, 3)
```

This is an anonymous function. It is also a closure, although in this case, it doesn't capture any values.

Alternative syntax: with anonymous variable (can use _ once only for each parameter)

Here we name our identifier as *x*

We can even specify the type of *x* if we like—but careful—you need parentheses.

- We could also do this of course (use a named function):

```
scala> def stringify(x: Int) = x.toString
stringify: (x: Int)String
scala> list map stringify
res4: List[String] = List(1, 2, 3)
scala> list.map(stringify)
res5: List[String] = List(1, 2, 3)
```

- And we could also do this (defines a "partial function"):

```
scala> list map {case x: Int => x.toString}
res0: List[String] = List(1, 2, 3)
```