**Higher-order Functions**

**Example: Map**

// explicit argument list in function

List**(**1**,** 2**,** 3**,** 4**).**map**(**x **=>** x **+** 1**)** **=>** List**(**2**,** 3**,** 4**,** 5**)**

// equivalent to the above, but implicit arguments

List**(**1**,** 2**,** 3**,** 4**).**map**(**\_ **+** 1**)** **=>** List**(**2**,** 3**,** 4**,** 5**)**

// different output and input type

List**(**1**,** 2**,** 3**,** 4**).**map**(**\_**.**toString **+** "a"**)** **=>** List**(**1a**,** 2a**,** 3a**,** 4a**)**

// this unpacks a tuple, note use of curly braces

List**((**1**,** 5**),** **(**2**,** 6**),** **(**3**,** 7**),** **(**4**,** 8**)).**map **{** **case** **(**x**,** y**)** **=>** x**\***y **}**

**=>** List**(**5**,** 12**,** 21**,** 32**)**

// Generating indices:

val myList **=** List**(**"a"**,** "b"**,** "c"**,** "d"**)**

println**((**0 until 4**).**map**(**myList**(**\_**)))**

**=>** myList**:** List**[**String**]** **=** List**(**"a"**,** "b"**,** "c"**,** "d"**)**

**Example: zipWithIndex**

// note indices start at zero

List**(**1**,** 2**,** 3**,** 4**).**zipWithIndex

**=>** List**((**1**,**0**),** **(**2**,**1**),** **(**3**,**2**),** **(**4**,**3**))**

List**(**"a"**,** "b"**,** "c"**,** "d"**).**zipWithIndex

**=>** List**((**a**,**0**),** **(**b**,**1**),** **(**c**,**2**),** **(**d**,**3**))**

// tuples nest

List**((**"a"**,** "b"**),** **(**"c"**,** "d"**),** **(**"e"**,** "f"**),** **(**"g"**,** "h"**)).**zipWithIndex

**=>** List**(((**a**,**b**),**0**),** **((**c**,**d**),**1**),** **((**e**,**f**),**2**),** **((**g**,**h**),**3**))**

**Example: Reduce**

// returns the sum of all the elements

List**(**1**,** 2**,** 3**,** 4**).**reduce**((**a**,** b**)** **=>** a **+** b**))** **=>** 10

// returns the product of all the elements

List**(**1**,** 2**,** 3**,** 4**).**reduce**(**\_ **\*** \_**)** **=>** 24

// you can chain reduce onto the result of a map

List**(**1**,** 2**,** 3**,** 4**).**map**(**\_ **+** 1**).**reduce**(**\_ **+** \_**)** **=>** 14

// Important note: reduce will fail with an empty list

List**[**Int**]().**reduce**(**\_ **\*** \_**)**

**Example: Fold**

// equivalent to the sum using reduce

List**(**1**,** 2**,** 3**,** 4**).**fold**(**0**)(**\_ **+** \_**) => 10**

// like above, but accumulation starts at 1

List**(**1**,** 2**,** 3**,** 4**).**fold**(**1**)(**\_ **+** \_**) => 11**

// unlike reduce, does not fail on an empty input

List**().**fold**(**1**)(**\_ **+** \_**)** **=> 1**

**Example: Function Object**

// These are normal functions.

def plus1funct**(**x**:** Int**):** Int **=** x **+** 1

def times2funct**(**x**:** Int**):** Int **=** x **\*** 2

// These are functions as vals.

// The first one explicitly specifies the return type.

val plus1val**:** Int **=>** Int **=** x **=>** x **+** 1

val times2val **=** **(**x**:** Int**)** **=>** x **\*** 2

// Calling both looks the same.

plus1funct**(**4**)**

plus1val**(**4**)**

plus1funct**(**x**=**4**)**

//plus1val(x=4) // this doesn't work

**Functional Programming**

**Example: Functions vs. Objects**

// both x and y call the nextInt function, but x is evaluated immediately and y is a function

val x **=** Random**.**nextInt

def y **=** Random**.**nextInt

// x was previously evaluated, so it is a constant

println**(**s"x = $x"**)**

println**(**s"x = $x"**)**

// y is a function and gets reevaluated at each call, thus these produce different results

println**(**s"y = $y"**)**

println**(**s"y = $y"**)**

**Example: Higher-Order Functions**

// create our function

val plus1 **=** **(**x**:** Int**)** **=>** x **+** 1

val times2 **=** **(**x**:** Int**)** **=>** x **\*** 2

// pass it to map, a list function

val myList **=** List**(**1**,** 2**,** 5**,** 9**)**

val myListPlus **=** myList**.**map**(**plus1**)**

val myListTimes **=** myList**.**map**(**times2**)**

// create a custom function, which performs an operation on X N times using recursion

def opN**(**x**:** Int**,** n**:** Int**,** op**:** Int **=>** Int**):** Int **=** **{**

**if** **(**n **<=** 0**)** **{** x **}**

**else** **{** opN**(**op**(**x**),** n**-**1**,** op**)** **}**

**}**

opN**(**7**,** 3**,** plus1**)**

opN**(**7**,** 3**,** times2**)**

**Example: Anonymous Functions**

val myList **=** List**(**5**,** 6**,** 7**,** 8**)**

// add one to every item in the list using an anonymous function

// arguments get passed to the underscore variable

// these all do the same thing

myList**.**map**(** **(**x**:**Int**)** **=>** x **+** 1 **)**

myList**.**map**(**\_ **+** 1**)**

// a common situation is to use case statements within an anonymous function

val myAnyList **=** List**(**1**,** 2**,** "3"**,** 4L**,** myList**)**

myAnyList**.**map **{**

**case** **(**\_**:**Int**|**\_**:**Long**)** **=>** "Number"

**case** \_**:**String **=>** "String"

**case** \_ **=>** "error"

**}**

**Object Oriented Programming**

**Example: Abstract Class**

abstract class MyAbstractClass **{**

def myFunction**(**i**:** Int**):** Int

val myValue**:** String

**}**

class ConcreteClass **extends** MyAbstractClass **{**

def myFunction**(**i**:** Int**):** Int **=** i **+** 1

val myValue **=** "Hello World!"

**}**

// Uncomment below to test!

// val abstractClass = new MyAbstractClass() // Illegal!

val concreteClass **=** **new** ConcreteClass**()** // Legal!

**Example: Trait**

Similar to abstract classes, but differ in two ways:

* A class can inherit from multiple traits
* A trait cannot have constructor parameters

trait HasFunction **{**

def myFunction**(**i**:** Int**):** Int

**}**

trait HasValue **{**

val myValue**:** String

val myOtherValue **=** 100

**}**

class MyClass **extends** HasFunction with HasValue **{**

override def myFunction**(**i**:** Int**):** Int **=** i **+** 1

val myValue **=** "Hello World!"

**}**

// Uncomment below to test!

// val myTraitFunction = new HasFunction() // Illegal! Cannot instantiate a trait

// val myTraitValue = new HasValue() // Illegal! Cannot instantiate a trait

val myClass **=** **new** MyClass**()** // Legal!

// to inherit multiple traits, chain them like

class MyClass **extends** HasTrait1 with HasTrait2 with HasTrait3 **...**

**Example: Objects**

Singleton classes (no need to call **new**)

object MyObject **{**

def hi**:** String **=** "Hello World!"

def apply**(**msg**:** String**)** **=** msg

**}**

println**(**MyObject**.**hi**)**

println**(**MyObject**(**"This message is important!"**))**

// equivalent to MyObject.apply(msg)

**Example: Companion Objects**

Common in Chisel, used for following reasons:

* to contain constants related to the class
* to execute code before/after the class constructor
* to create multiple constructors for a class

**case** class SomeGeneratorParameters**(**

someWidth**:** Int**,**

someOtherWidth**:** Int **=** 10**,**

pipelineMe**:** Boolean **=** **false**

**)** **{**

require**(**someWidth **>=** 0**)**

require**(**someOtherWidth **>=** 0**)**

val totalWidth **=** someWidth **+** someOtherWidth

**}**

object Animal **{**

val defaultName **=** "Bigfoot"

private var numberOfAnimals **=** 0

def apply**(**name**:** String**):** Animal **=** **{** // factory method

numberOfAnimals **+=** 1

**new** Animal**(**name**,** numberOfAnimals**)**

**}**

def apply**():** Animal **=** apply**(**defaultName**)** // factory method

**}**

class Animal**(**name**:** String**,** order**:** Int**)** **{**

def info**:** String **=** s"Hi my name is $name, and I'm $order in line!"

**}**

// Calls the Animal factory method

val bunny **=** Animal**.**apply**(**"Hopper"**)**

val cat **=** Animal**(**"Whiskers"**)**

val yeti **=** Animal**()**

**Example: Case Classes**

Very common in Scala, useful features:

* Allow **external access** to the **class parameters**
* **Eliminates** the need to use **new** when instantiating the class
* Automatically creates an **unapply** **method** that supplies access to all of the class Parameters
* Cannot be subclassed from

class Nail**(**length**:** Int**)** // Regular class

val nail **=** **new** Nail**(**10**)** // Requires the `new` keyword

// println(nail.length) // Illegal! Class constructor parameters are not by default externally visible

class Screw**(**val threadSpace**:** Int**)** // By using the `val` keyword, threadSpace is now externally visible

val screw **=** **new** Screw**(**2**)** // Requires the `new` keyword

println**(**screw**.**threadSpace**)**

**case** class Staple**(**isClosed**:** Boolean**)** // Case class constructor parameters are, by default, externally visible

val staple **=** Staple**(false)** // No `new` keyword required

println**(**staple**.**isClosed**)**