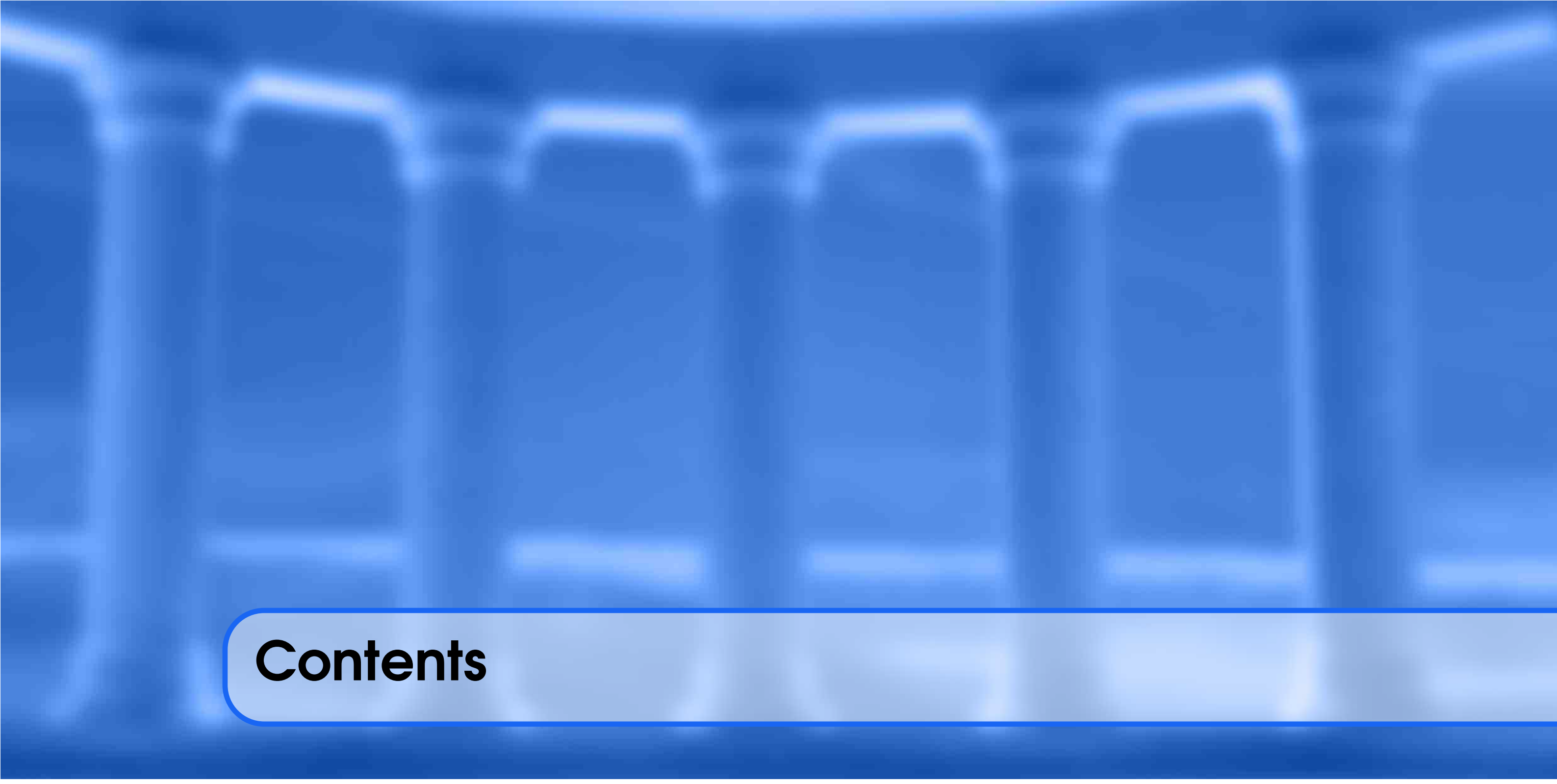
The Concurnas Programming Language

Reference Manual

v1.14.020 (08/03/2020)



Contents

Before the world had ever come to be,
This was set apart for you and I alone.
Although we go as shadows wandering,
Together we will make the journey home.

I've come so far
to see that every light will fade on the horizon
What do you seek?
Don't you know that there are eyes that see beyond you?

Countless days of solitude.
Frozen, my heart is frozen.
I feel the wound in my side
Open, I feel it open.

Who calls to me?
Will you fade into the wind like those before you?
In all your searching
What have you found that did not bind you or betray you?

Countless days of solitude.
Broken, my heart is Broken.
I feel the wound in my side
Open, I feel it open.

The woman came to me, clothed with the sun,
So wrapt in living flame, I begged to serve at her command.

This fire will not overcome you.
And fate is not bound in the past.
These flames will consume fear and weakness,
So don't forget where we began.

The same curse broke better men
I bow my head in shame
The same curse is set ablaze
as I walk into the flames.

ORACLE
from Nexus by Monolith, 2016
`https://monolithcanada.bandcamp.com/track/oracle`

The Basics

Terannia sent him books about mathematics and electrical circuits when she could, but they were all mimeographed copies of originals and didn't tell him what he really needed to know. But they gave him the fundamentals, so code could be written to solve the problems. Code could do anything. Code could save the world.

- Peter F. Hamilton. (2016) Night Without Stars. Pan Books..

1. Overview

This is a high level introductory overview of the Concurnas syntax written to give the reader a basic understanding of the core structure of code which can be written in Concurnas.

Broadly speaking, a Concurnas program consists of one or more `.conc` source files expressed in a directory structure (the organization of which denotes the 'package' structure of the code). The source code within these source files may be in Unicode format (i.e. permitting non-English/mathematical variable and method names) and consist of the following sorts of entities:

- **Functions and methods:**

```
def sum(a int, b int){
  return a+b
}

@Annotated(param=2)
def plusTwo(a int) => sum(a, 2)
```

- **Unicode:**

```
  π = Math.PI
  πStr = "π=" + π//π=3.141592653589793
```

- **Expressions:**

```
callAFunction()
[1 2 3 4] //an integer array
[1 2 ; 3 4] //an integer matrix definition
[1, 2, 3, 4]//an integer list
2+2
2 if eval() else 15
n+1 for n in [1 2 3]//list comprehension
```

- **Statements.** Such as assignments:

```
a = 8
anint int = 99
var reassignOk = 12//new variable
val nonReassign = 99//new variable cannot reassign
```

- **Control flow statements:**

```
for(a in [1 2 3]){
     processIt(a)
}
while(xyz()){
  doSomething()
}
if(a){
      doThis()
} elif(b){
      doAnother()
} else {
      another()
}
```

- **Long lines.** Code which would otherwise make for a very wide line of code may be split up for aesthetic purposes using a backslash, \:

```
a = 8 \
   + \
   10
```

- **Blocks.** A series of entities as par above surrounded by curly braces

```
{
//many lines of code go here...
}
```

- **Exceptions:**

```
try{
     something()
}catch(e AnException){
     //react as appropriate...
}catch(all){
     //catchall
}
```

- **Core object oriented components:**

```
class MyClass
trait MyTrait
annotation MyAnnotation
actor MyActor
actor AnotherActor of MyClass
enum MyEnum{ONE, TWO, THREE}
```

- **Isolates and refs:**

```
aref := 21 //a reference
res = { 'hello' + ' world' }!
//^ string concatenation performed in dedicated thread like isolate,
    returns ref
res int: = myfunc(12)!
//^ myfunc called in dedicated thread like isolate, returns ref
res = myfunc(12)!//same as above with return type inferred
```

- **Reactive concurrent statements:**

```
onchange(x){ doSomething()}//on change of x, perform an action
every(x){ doSomething(x)}//as above but including initial value
await(x; x > 2)//pause execution until the condition is met
trans{//a transaction acting upon two references
   a -= 10
   b += 10
}
z <= a+b//shorthand for every
z <- a+b//shorthand for onchange
```

- **Object providers:**

```
provider TestTweeterClient{
   provide TweeterClient
   Shortener => new TestShortner()
   Tweeter => new MockTweeter()
}
```

- **GPU computing:**

```
//a simple kernel for performing matrix multiplication...
gpukernel 2 matMult(wA int, wB int, global in matA float[2], global in matB
    float[2], global out result float[2]) {
   globalRow = get_global_id(0) // Row ID
   globalCol = get_global_id(1) // Col ID

   value = 0f;
   for (k = 0; k < wA; ++k) {
      value += matA[globalCol * wA + k] * matB[k * wB + globalRow];
   }

   // Write element to output matrix
   result[globalCol * wA + globalRow] = value;
}
```

- **Comments and documentation blocks:**

```
//This is a one line comment

/* This is a multi-
line Comment */

/**
A documentation block
*/
```

```
/*
Comment here
/*
A nested comment
*/
*/
```

Concurnas is an optionally semicolon terminated/separated language. The newline character operates to signify the end of a line of code. For example, the following function definitions are identical:

```
def helloWorld1(){
   ret = "hello World";
   return ret
}

def helloWorld2(){
   ret = "hello World"
   return ret
}

def helloWorld3(){
    ret = "hello World"; return ret
}
```

Concurnas usually offers a compact version of syntax in cases where it would otherwise be unnecessarily verbose to write (as often seen in older programming languages) for instance, for our hello world example above we could write:

```
def helloWorld4() => "hello World"
```

Concurnas runs on the JVM and thus gains access two both the high performance runtime of the JVM as well as the JDK standard library.

1.1 Coding with Concurnas

The ordinary workflow when building and running programs written in Concurnas is as follows:

1. Setup a project using an IDE - currently Concurnas has support for Atom, Sublime Text and VS Code *(coming soon!)*
2. Write code as appropriate saved into .conc files within a directory structure appropriately reflective of our desired packaged hierarchy.
3. Debug said code if a debugger is provided in our chosen IDE. (optional step)
4. Compile using the Concurnas command line compiler: concc, or via our IDE if it is integrated as such.
5. Execute the program using the Concurnas command line: conc.

Concurnas also offers the following 'lightweight' approaches.

- Concurnas REPL. The `conc` program can will spawn an interactive shell. This is discussed in the REPL chapter.
- Jupyter notebooks. Concurnas integrates well with Jupyter *(coming soon!)*

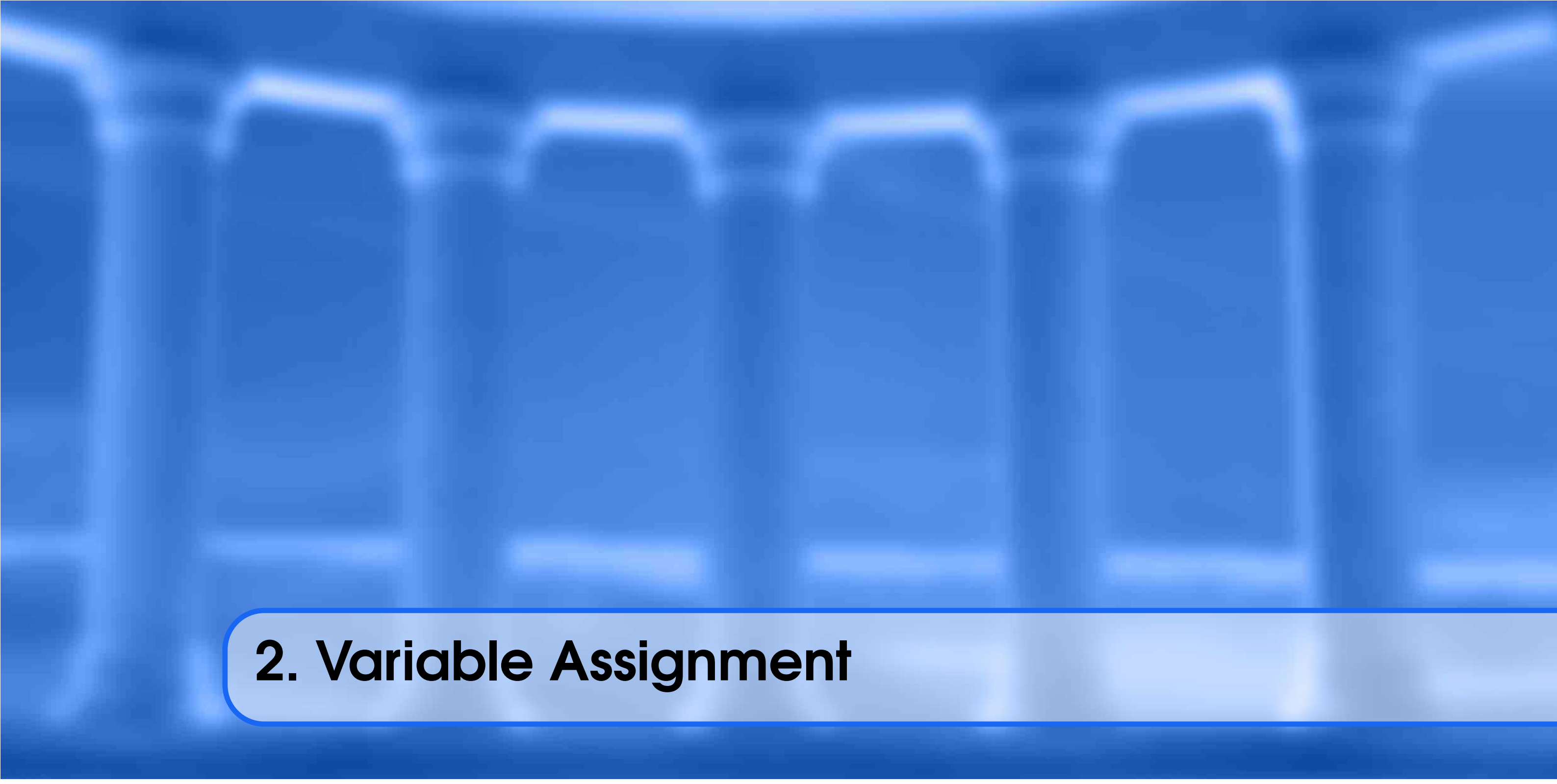
Both of the lightweight approaches above are great for trying out features of Concurnas in a scriptable manner, and also writing real software.

1.2 Requirements

Concurnas is a JVM language, as such it requires at least Java 1.8 to be accessible.

1.3 Compatibility

Concurnas has been verified as compatible with Java 1.8 to 14. Any system upon which these versions of Java runs on, Concurnas can also run on (i.e. Windows and Linux)



2. Variable Assignment

Let us first define what is means to declare a variable. A variable declaration consists of a variable name and a type. Variable names can be any combination of standard ASCII alphanumeric characters or non ASCII Unicode characters. Types may be either primitive, tuples, typedefs, Object types (which includes user defined classes) or refs to the aforementioned types. For example:

```
anint int //a primative type
atuple (String, int) //a tuple

typedef MyMap = java.lang.HashMap<String, String> //a typedef
aMyMap MyMap //use of a typedef

class MyClass //a class declaration
anObject MyClass //use of a class

aref int: //a ref of type int
```

2.1 Scopes

Usually if we declare a variable then we'd wish to assign a value to it at some point during the bounded scope of said variable. The bounded scope constitutes the inner most pair of curly braces and any nested braces within in which the variable is declared. Variables can only be assigned to and used within their bounded scope.

```
if(something()){
   anint = 99
   if(somethingelse()){
      anint = anint + 1
   }
}
//anint cannot be used after this point as that usage would be outside its
    bounded scope, the following is invalid:
```

```
res = anint//invalid useage
```

It can sometimes be useful to define a scope on its own (i.e. not associated with a control structure such as a if else statement or a function definition etc). This can be achieved as follows:

```
{
   anint = 99
   //do somethiing with anint
}
//anint is no longer in scope at this point...
```

2.2 Assigning values

We can assign a value to a predeclared variable using the assignment operator = as follows:

```
avar int//declaration

avar = 99 //assignment
```

More commonly we perform the initial assignment and variable declaration in the same step. This has the added advantage of allowing Concurnas to use type inference (if we wish) in order to determine the type of the variable for us, saving us the effort of having to write out a verbose type and increasing readability of our code.

So instead of this:

```
avar int = 99
```

We can write this:

```
avar = 99
```

Concurnas is an optionally type inferred language. What this means in practice is that 90% of the time it's not necessary to explicitly specify the type of a variable

The optionality in Concurnas' design regarding inference is useful for describing problems with complex typing structure which otherwise would not be obvious to determine from reading the code.

2.3 val and var

The `val` and `var` keywords are useful in two ways. First, either one make it explicitly clear that one is defining a new variable and secondly, the `val` keyword stipulates that no assignment post initial declaration/assignment is permitted. For example:

```
var anint = 99
val another = 99
```

Both the above variables are new (it would be a compilation error if they had already been declared in the scope prior to the above code), and in the case of the `another` variable, it cannot be reassigned.

2.4 Unassigned declared variables

It's considered a compilation error to define a variable but never assign a value to it (or to attempt to use a variable at a point after declaration when it's possible that execution of the code has progressed through a path in which a value is not assigned). The only case in which it's valid to declare a variable but not assign it a value is where the variable is a of a ref type (used to communicate between isolates - the principle unit of concurrency in Concurnas). For example:

```
def assignIntRef(an int:){//function to assign a value to a ref
   an = 15
}

aint int: //declaration of a local ref

onchange(aint) => System out println "aint was assigned: {aint}"

assignIntRef(aint)
```

2.5 Multi Assignment

Sometimes it can be useful to assign a value to more than one variables. To this end Concurnas supports multi assignment:

```
a = b = 20

//a == 20
//b == 20
```

We can have as many expressions as we like on the left hand side of the multi assignment before the final expression which we are assigning from. The expressions on the left hand side chain can be of any format normally valid on the left hand side of an expression:

```
ar = new int[10](0)
val aval = ar[1] = ar[5] = 30

//aval == 30
//ar == [0 30 0 0 0 30 0 0 0 0]
```

Note that the expression on the right hand side is only evaluated once, the assignees on the left hand side chain receive the same reference to the right hand side value, i.e. No copy is made.

```
myArray = [1 2 3]
b = c = myArray

b[0] = 99

//b == [99 2 3]
//c == [99 2 3]
```

One is not limited to the use of just the = assignment, one may use any valid assignor in the chain:

```
a = 100
a += b = 2
```

```
//a == 102
//b == 2
```

2.6 Compound assignment

Compound assignment refers to assignments performed on variables which have been defined already and which apply an operator in addition to assigning a new value to said variable. These are: `+=`, `-=`, `*=`, `/=`, `**=`, `mod=`, `or=`, `and=`, `<<=`, `>>=`, `>>>=`, `band=`, `bor=`, `bxor=`. Taking the addition assignment operator, `+=` as an example, the following two statements are functionally identical:

```
a1 = a2 = 10

a1 += 1
a2 = a2 + 1

assert a1 == a2
```

That is to say, `a1 += 1` is essentially shorthand for: `a1 = a1 + 1`.

The compound assignment operators may be applied to array reference operations as follows:

```
a1 = [1 2 3]

a1[0] += 1

assert a1 == [2 2 3]
```

Again, `a1[0] += 1` is essentially shorthand for: `a1[0] = a1[0] + 1`.

2.7 Lazy variables

The form of evaluation used in Concurnas is "eager evaluation" - i.e. as soon as an expression is bound to a variable, it is evaluated. Additionally, this can occur concurrently in the case of ref assignments occurring in differing iso's. Let's look at an ordinary example:

```
myvar = {1+3}
```

In the above example, after the `1+3` expression is bound to the variable myvar, it is executed. Thereafter myvar holds the value of 4.

There are some problems for which this form of eager evaluation is inappropriate and for which we'd prefer to defer execution of the expression until the associated bound variable is accessed. This is lazy evaluation and Concurnas offers a controlled form of this expressed in the form of lazy variables.

We can define a lazy variable as follows:

```
lazy myvar = {1+3}
```

Now, only when myvar is accessed is the bound expression `1+3` evaluated. Note also that this evaluation occurs only once. Subsequent assignments to the variable are possible and the associated bound expressions will only be evaluated once the variable is accessed, just like the initial assignment.

Let's look at a more interesting example. First, normal execution without a lazy variable:

```
a = 100
myvar = {a=400; 22}//assign, with side effect of re-assigning a

res = [a myvar a] //lets create an array holding the value of a, the evaluated
    myvar and a again

//res == [400 22 400]
```

In the above case we see that in assigning a value to `myvar`, we invoke the side effect of assigning a new value to a. Since this occurs as `myvar` is initially created, before we have a chance to first access `a`, we only see a as having the new value of `400`. And this is reflected in the first and last elements of the `res` array referring to the value of variable `a`. Let's change this with a lazy variable:

```
a = 100
lazy myvar = {a=400; 22}//assign, with side effect of re-assigning a

res = [a myvar a] //lets create an array holding the value of a, the evaluated
    myvar and a again

//res == [100 22 400]
```

Now we can see that we can see the initial value of a, since when we are creating our `res` array, we access a before we access `myvar`, since the expression bound to `myvar` is lazy evaluated and has not yet been evaluated, the side effect assigning the value of `400` to a has not yet occurred. However, after accessing `myvar` for our second array value, a is not assigned the value `400` and this is reflected when we extract the value for the final element of the array.

Where function or method parameters are defined as being lazy, the passed bindings are only evaluated upon use within the function or method. For example:

```
a = 100

def makeArray(lazy operate int) => [a operate a]

res = makeArray({a=400; 22})

//res == [100 22 400]
```

We see in the above example, that the block passed as an argument to the `makeArray` function is only evaluated during execution of said function - and not initial invocation of the function.

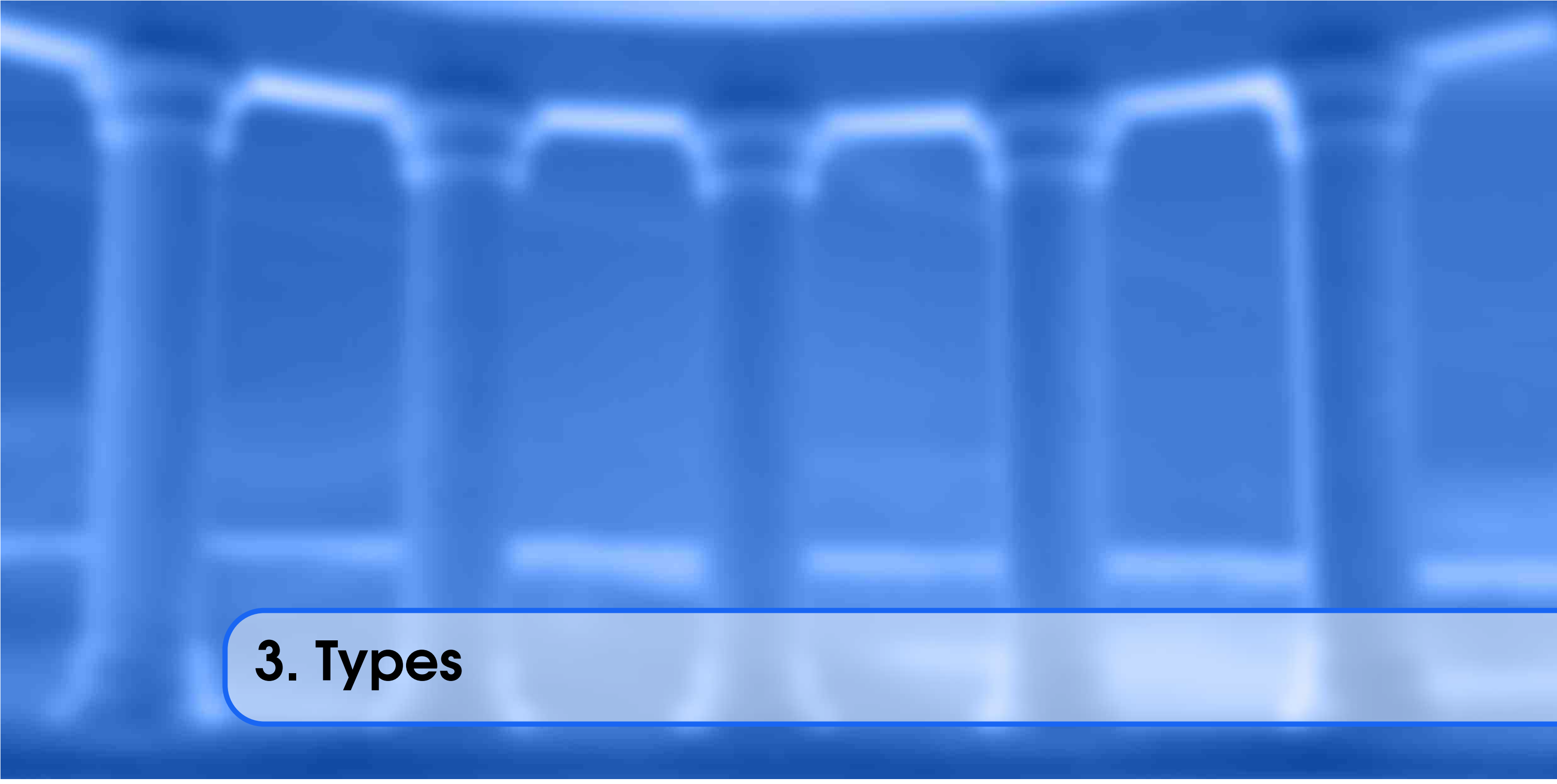
Behind the scenes, in order to implement lazy variables, we are overloading the unassign operator(see operator overloading). What if we don't want to evaluate the bound expression of a lazy variable when we refer to it, for instance when passing it to a function invocation? In this case we can use the `:` operator as follows:

```
a = 100
lazy myvar = {a=400; 22}//assign, with side effect of re-assigning a

def makeArray(lazy operate int) => [a operate a]

res = makeArray(myvar:)

//res == [100 22 400]
```

Here we are preventing unassignment of `myvar`, instead this (and therefore execution of the binding of `myvar`) is allowed to occur within the `makeArray` function.

We can make use of lazy variables within function/method references as follows:

```
a = 88

def afunc(lazy operate int) => [a operate a]

xx = afunc&(lazy int)
res = xx({a=400; 22})

//res == [100 22 400]
```



3. Types

In a sense, all programming is about data. As such the way in which we represent data, via types, is a fundamental concept in any programming language. In Concurnas, roughly speaking we have two foundation types: primitive and object types. From which we are able to derive six composite types of: arrays, method references, generics, actors, references and tuples.

Why does Concurnas support both primitive types and Object types (after all, primitive types can all be represented via object types). Two reasons: performance and memory utilization. Generally speaking if we represent data which can be expressed as a primitive type using an object, then we consume more memory than we need to and can make the job of memory management (garbage collection, heap vs stack allocation etc) more difficult than it needs to be. Additionally our CPU processors are designed to perform operations on primitive type data, so by using objects we are adding a level of indirection which of course has a (relatively minor) performance penalty. In any case, although behind the scenes via a compiler/runtime optimization this penalty is mostly eliminated, it is still useful to be able to give the programmer control over the use of primitive or object types.

That being said there are some circumstances in which using primitive types may seem to be a good idea, but we are forced instead to implicitly use object types - for instance, with generics. This is generally done to make code and libraries easier to understand, utilize and to eliminate code duplication.

3.1 Primitive Types

Let us first begin with the numerical primitive types of which there are eight:

Type Name	Bit width(bytes)	Min/max values
`bool` or `boolean`[a]	32 (1)	`true` or `false`
`char`	16 (2)	$-2^{16}, 2^{16}-1$
`byte`	8 (1)	$-2^7, 2^7-1$
`short`	16 (2)	$-2^{15}, 2^{15}-1$
`int`[b]	32 (4)	$-2^{31}, 2^{31}-1$
`long`	64 (8)	$-2^{63}, 2^{63}-1$
`float`	32 (4)	see IEEE_754 docs
`double`[c]	64 (8)	see IEEE_754 docs

[a]internally to Concurnas (or to be more precise, the JVM) a 32-bit int is used to represent a boolean.

[b]int is the default type used when an integer literal is expressed.

[c]double is the default type used when a decimal number literal is expressed.

The byte, short, int and long types allows us to store integers (whole number) values accurately. If one wishes to store real numbers (i.e. with a decimal point), then the float and double types are available (with double offering more precision) - though they can only offer a finite degree of precision post the decimal point. For an accurate decimal representation to a fixed degree, using a fixed point decimal representation such as the object type: java.math.BigDecimal is recommended.

- A **int** literal can be expressed as:
  - Decimal: `123`
  - Binary: `0b010110`
  - Hexadecimal: `0x0E`
- A **long** literal can be expressed as:
  - Decimal: `123l` or `123L`
- A **short** literal can be expressed as:
  - Decimal: `16s` or `16S`
- A **byte** literal must be cast from:
  - An int of value between $-2^7, 2^7-1$ using the cast operator `as`. e.g.: `12 as byte`
- A **float** literal can be expressed as:
  - Decimal: `12f` or `12F`
  - Real: `12.2f` or `12.2F`
  - Engineering: `12e6f` or `12e6F`
- A **double** literal can be expressed as:
  - Decimal: `12.` or `12d` or `12D`
  - Real: `12.2` or `12.2d` or `12.2D`
  - Engineering: `12e6` or `12e6d` or `12e6D`

3.1.1 Char's

Characters are represented by the primitive type `char`, are 32 bits wide and can be expressed as a single character surrounded by either a pair of `''` or `""`:

```
achar = 'a'
achar = "a"
```

Characters are of some use in modern programming, but Unicode Strings (as seen later) are usually more useful.

3.1.2 Booleans

Booleans are represented by the primitive type `boolean` or `bool` (use of which is a matter of personal preference), and are expressible as: `true` or `false`:

```
abool = true
another boolean = false
```

3.1.3 Bytes

Bytes are represented by the primitive type `byte`, and are creatable via a cast:

```
mybyte = 0x011010 as byte
```

3.1.4 Choosing primitives

When choosing which primitive type to represent our numerical data with, it is important to consider the range of values that our data can have and choose the smallest bit width type which best aligns to that data (so as to conserve memory). This is especially the case when defining large n dimensional arrays of data. However, these days our systems are generally non-memory bound, and as such over allocating and say using an `int` where a `short` would be more appropriate is not considered a problem. If in doubt over allocate, and don't worry too much.

3.2 Strings

It turns out that in modern programming Strings are so proliferant that they deserve their own first-class citizen support (believe it or not, but there was a time when this was not the case!). In Concurnas there are a few handy special considerations to the language just to aid in working with Strings. Note that strings themselves are objects (covered in the next section).

At the most basic level, Strings can be defined via use of either `''` or `""`. For example:

```
astring = 'Hello World!'
another = "I'm also a String"
```

Note above that by permitting both `""` and `''` for string defining, we're able to use `'` and `"` respectfully in our Strings. But, if one does not wish to switch between one style or another in order to use these characters inside a string, or has both a `'` and a `"` in the string one is defining, then an escape character \may be used. For example:

```
astring = 'I\'m also a String'
```

So as to disambiguate between the definition of a String and a character, for cases of single character length strings, using the `''` denotes a character, and `""` denotes a String. For example:

```
aString = "c"
aChar   = 'c'
```

3.2.1 The escape character

As we have already seen the escape character \can be used to input a `'` or `"` inside a `''` or `""` string. It may also be used to input a range of special characters including the newline character: \n and Unicode, UTF-16 characters. For example:

```
anewline = "Hello\nworld!"
/*=>Hello
world!*/
aString = "\u0048\u0065\u006c\u006c\u006f \u0077\u006f\u0072\u006c\u0064\u0021"
    //"=>Hello world!"
```

3.2.2 The String concatenation operator +

We can compose strings by using the concatenation + operator. This is useful for mixing raw text or pre-existing string variables and variables/expression values together. For instance:

```
avar = 8
str = " there m"
aString = "hi" + str + avar
//aString =="hi there m8"
```

In order for string concatenation via the + operator to work, at least one of the right-hand side or left-hand side arguments must be a String, however, even an empty String `""` is acceptable.

In the case where an object type is involved in String concatenation the object's `toString` method is invoked. This method is included for all objects by virtue of the fact that they all inherit from Class `java.lang.Object` where a default implementation returning the objects class name and hexadecimalized hashCode is returned as a string.

In the case where an n dimensional array is on the left or right hand side of the concatenation it will be converted into a formatted String. For example:

```
mat = [ 1 2 ; 3 4]

asStr = "" + mat
//asStr == "[[1 2] [3 4]]"
```

Note that since strings are immutable objects, a new String is created as a result of the + operator being applied.

3.2.3 Assignment plus +=

The assignment plus operator += operates in a similar manner. If the item on the left-hand side of the plus assignment operator is a String, then the item on the right-hand side of the plus assignment operator is appended to the left-hand side and a new string is assigned to the left-hand side item.

3.2.4 String is an Object

The type String is an object - therefore it can be created in the usual way for Objects:

```
str = new java.lang.String(23)
```

3.2.5 Code in Strings

Another handy trick when creating strings is to nest raw code within the string itself. This is great for including variables directly in strings without having to resort to the somewhat cumbersome usage of multiple + operators as par above.

```
avar = 8
str = " there m"
```

```
aString = "hi {str} {avar}"
//aString == "hi there m8"
```

We can include more complex expressions as follows:

```
aString = "addition result: {1+2}"
//aString == "addition result: 3"
```

Code in Strings can be applied everywhere except for when defining Strings for default annotation parameters - as these are required to be fully-defined constants at compilation time. Code within strings nest, that is to say that; code embedded within the string may nest other strings which may optionally have code within them too.

Escaping code in Strings

Code in Strings may be escaped by prefixing said code with a \:

```
aString = "addition result: \{1+2}"
//aString == "addition result: {1+2}"
```

3.2.6 Format Strings

One final mechanism by which we can create Strings is via the format, static method of `java.lang.String`

```
var1 = 99
var2 = 234.
String.format("var1: %s var2: %s", var1, var2)
```

3.2.7 Additional operations on Strings

Concurnas supports the additional following set of basic operations on Strings:

Contains

A substring may be checked for:

```
cont = "de" in "abcdefg"
```

Character at

The character at a specific point in a String may be determined:

```
achar = "abcdefg"[2] //c
```

Substring

A substring between two points:

```
substr = "abcdefg"[2 ... 4] //cd
```

Substring from

A substring from a point to the end of a string:

```
substr = "abcdefg"[4 ... ] //efg
```

Substring to

A substring from the start to a point:

```
substr = "abcdefg"[ ... 4] //abcd
```

Iteration

Iteration over all characters of a String:

```
elms = (x, x == 'b') for x in "abcdefg"
//elms ==> [(a, false), (b, true), (c, false), (d, false), (e, false), (f,
    false), (g, false)]
```

3.2.8 More information on Strings

Strings in Concurnas are supported by the JVM, for more information on Strings see: Oracle Java String documentation.

3.3 Regex

Regex, or regular expressions, are another area of modern programming so proliferant that Concurnas has special first-class citizen support. We can define a regex in much the same way as a String, by prepending the regex String within `''` or `""` with `r`, to make `r''` or `r""` for example:

```
aregex = r"ab*a"
...
amatch = aregex.matcher("abbbba").matches()
```

This syntax produces an object of type `java.util.regex.Pattern`. More details of this can be found here. For more details on the variant of regex supported by Concurnas, see here. Note that it's relatively expensive to construct regex objects since they are compiled at runtime upon definition so its recommended that they be created as top level variables or cached and/or otherwise reused.

3.4 Object Types

Objects and Classes are covered in more detail in the classes section. This is a brief introduction:

```
open class MyClass(public an int)
class ChildClass extends MyClass(88){
   def amethod() => "returns something"
}

myObject1 = MyClass(12)
myObject2 Object = MyClass(12)
myChildObj1 = ChildClass()
myChildObj2 MyClass = ChildClass()
```

`myObject1` and `myObject2` are instance objects of type class `MyClass`. However, in the case of `myObject2` since it has been declared as being of type `Object` - it can only be used as an instance of Object, the `an` field is not accessible unless the object is cast to an instance of `MyClass`.

`myObject1` and `myObject2` are instance objects of type `ChildClass`. However, in the case of `myChildObj2` since it has been declared as being of type `MyClass` - it can only be used as an instance of MyClass, the amethod method is not callable unless the variable is cast to an instance of ChildClass. For myChildObj1 we can access the field an from the superclass MyClass.

Note that java.lang.Object is an implicit superclass of every class if a super type is not defined. As such it is the superclass of every instance object.

3.5 null

Null is a special type of Object and is represented simply by the keyword null. Any object may be assigned a value of null as it is a subtype of all Object types (including arrays). Attempting to call a method or access a field on a nullable object which is null will result in a java.lang.NullPointerException object being thrown. This means that where one suspects an object may be null, it is necessary to check for the null state. As such null should only be used sparingly.

Generally speaking, the conventional wisdom of modern software engineering is that null should be used sparingly. As such if one wishes for a variable to be potentially nullable, its type must be declared as being nullable, this is denoted by appending a ? to the type. For example:

```
aString String = "hi"
nullable String? = null
nullable        = "no longer null"
```

The subject of null safety is covered in detail in the null safety chapter. For now let's look at how we can use null:

```
assignedNull String = null //an Object of type null assigned a String
aArray int[]? = null
aMatrix int[2]? = null

class User(name String, age int)

def extractName(user User?){//function taking an object argument which may
    resolve to null
   "unknown" if user == null else user.name //testing for nullability
}

def extractAge(user User? = null){//function with a default parameter resolving
    to null
   0 if user == null else user.age
}

name = extractName(null)//passing null as an argument to a function
age = extractAge()//the single argument to the function call will be populated
    with its default value of null
```

3.6 (Un)Boxed primitive types

For every primitive type, there exists an Object type. Concurnas allows either to be used on an interchangeable basis via a process known as boxing and unboxing.

The pairs of primitive and object types are as follows:

Primitive Type	Object Type
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`
`char`	`java.lang.Character`
`boolean` or `bool`	`java.lang.Boolean`

These can be used interchangeability for example:

3.6.1 Boxing

```
def opOnInteger(an Integer) => an+1

opOnInteger(12)
```

Above, the value passed as an argument in the call to `opOnInteger` is 'boxed' to type Integer. Behind the scenes the code is converted into the form:

```
opOnInteger(new Integer(12))
```

3.6.2 Unboxing

```
def givesAnInteger(from int) => new Integer(from)

avar int = givesAnInteger(12)
```

Above, the value returned from `givesAnInteger`, of type `Integer`, is 'unboxed' to type `int`. Behind the scenes the code is converted into the form: `givesAnInteger(12).intValue()`

Autoboxing and unboxing also conveniently allows us to use primitive types as generic type qualifiers. For example:

```
anAr = new java.util.Arraylist<int>()
```

3.7 Arrays

Arrays are covered in more detail in the arrays section. This is a brief explanation.

When we refer to arrays we are typically referring to a one dimensional array. For example:

```
arra1 int[] = [1 2 3 4 5]
arra2 int[1] = [1 2 3 4 5]
```

The Array type is simply: normal type with `[]` postfixed or `[1]` - indicating one level of dimensionality.

When it comes to n dimensional arrays, for instance a matrix (n=2), the type syntax is very similar:

```
mat1 int[][] = [1 2 ; 3 4]
```

```
mat2 int[2] = [1 2 ; 3 4]
```

Hence, the syntax to define an n dimensional array is: normal type, with n `[]`'s or `[n]`.

An array can be composed of any type - in other words, you can have an array of elements of any type in Concurnas. However, all the elements of said array need to be of the same type. For example the type of this array:

```
["hi", 23]
```

Is `java.lang.Object`, since this is the most specific type which is a supertype of all elements of the array (the second `int` item will be boxed to an Integer, which is a subtype of `java.lang.Object`).

We can extract an array from a matrix and an individual item from a matrix as follows:

```
mat int[2] = [1 2 ; 3 4]
ar int[] = mat[1]
item int = mat[1, 2]
```

When we perform an m level extraction we are returning a type: composed[n-m], and where n-m == 0 the composed type itself is returned.

When we set values of n dimensional matrices, they do not have to match 1:1 with the declared type of the matrix, but they may be subtypes. For example:

```
objAr Object[] = [ "hi", "there", 123]
objAr[2] = "friend"//replace last item with a String
```

3.8 Method References

Method (or function) references can be created via a number of mechanisms; In addition to being the types of method references they are the types of lambda definitions. They take the form of a list of input arguments (which may be empty) and a return type. For example:

```
def plusTogether(a int, b int) => a+b

ref (int, int) int = plusTogether&//create a function reference to plusTogether

aslambda (int, int) int = def(a int, b int) => a + b//create a lambda
```

They are handy types and can be passed around our program and invoked as follows:

```
ans1 int = ref(1, 1)
ans2 int = aslambda(1, 1)
```

All method references are a subtype of `lambda` which is itself a shortcut for `com.concurnas.bootstrap.lang.Lambd`

```
def doer(a int) => a*2

afuncref lambda = doer&
afuncref com.concurnas.bootstrap.lang.Lambda = doer&
```

An alternative representation for a method reference is to use their associated object type. For a method reference returning a non void type:

```
def doer(a int) => a*2
```

```
afuncref (int) int = doer&
altrep Function1<int, int> = afuncref
```

And for a method reference returning void:

```
def doer(a int) void {}

afuncref (int) void = doer&
altrep Function1v<int> = afuncref
```

3.8.1 Composite types of Method References

A type which is composed of method references can be defined by surrounding the method reference in parentheses. This is handy for defining array or method references with arguments or return types themselves being method references:

```
def applyMethodRefs(what ((int) int)[]) //takes an array of (int) int method
    refernces as input
   => w(20) for w in what

duplicator ((int) int) ((int) int)[] = def ( input (int) int ) => [input input]
duplicatorDup (((int) int) ((int) int)[])[] = [duplicator& duplicator&]
```

3.9 Generic Types

Generics are covered in more detail in the Generics chapter. This is a brief explanation.

Let us create an instance object of a class that supports Generics:

```
class MyGenClass<X>

instObj1 MyGenClass<String> = new MyGenClass<String>()
```

Above we create an object `instObj1` which is of type `MyGenClass` and having generic qualification `String` since `MyGenClass` requires one generic type qualification which it refers to internally as `X`. When creating instance objects of classes requiring generic type qualification - these generic types must be provided.

Bear in mind that `MyGenClass<String>` is not a subtype of `MyGenClass<Object>` - the generic type qualifications must match, they cannot be subtypes.

In instances, except creation, where we really do not know the generic type, or don't care for it, then we can use the wildcard `?` in place of their declaration. For example:

```
instObj2 MyGenClass<?> = new MyGenClass<String>()
```

3.10 Tuples

Tuples are covered in more detail in the tuples section. This is a brief explanation:

```
atuple = 12, "hi"
another (int, String) = 12, "hi"
another2 Tuple2<int, String> = 12, "hi"
```

All three above are equivalent and resolve to a tuple with two elements of Type: `(Integer, String)`.

3.11 Typedefs

Typedefs are covered in more detail in the Typedefs section. They are most commonly used in order to avoid code duplication in writing long type definitions (particularly those with many complex generic type qualifications) so as to improve code readability. Let's briefly create and use a typedef:

```
typedef MyMap = java.util.HashMap<String, java.Util.HashSet<int>>

am MyMap = new MyMap()
am2 java.util.HashMap<String, java.Util.HashSet<int>> = new
    java.util.HashMap<String, java.Util.HashSet<int>>();
```

`am` and `am2` are essentially equivalent to one another in type. At compilation time the first definition and assignment to `am` is expanded into the second form of `am2`.

At this point the key thing to note with typedefs is that they are essentially shortcuts ('drop in replacements') for otherwise longer, more verbose type definitions.

3.12 Refs

Refs are covered in more detail in the Refs section. For now let's look at a brief introduction to refs from a typing perspective:

```
aref1 int: //defined but unassigned local ref of type int
aref2 int := 21 //defined and assigned local ref of type int
aref3 := 21 //assigned local ref of inferred type int
aref4 = 21: //assigned variable of inferred type int: with the right-hand side
    being a ref creation expression
aref5 = {21}! //assigned variable of inferred type int: - right-hand side being
    a ref returned from an isolate - this is the most computationally expensive
    variety of ref creation - if you find yourself doing this to create simple
    refs from expressions that don't require concurrent execution, consider
    using the preceding form
```

All the refs above are of type `int:`. As can be seen above, the key when creating refs is the use of : postfixed to a type or prefixed to the assignment operator = - this tells the compiler that we wish to create a ref type.

Refs enable us to do two very useful things core to the operation of most programs written in Concurnas: They provide a mechanism by which differing isolates (pieces of code which are executed at the same time) can communicate between each other. For now, consider that isolates are concurrently executed blocks of code indicated by post-pending the block with the isolate creation bang `!`. e.g `{1+1}!` They enable reactive computing, via use of onchange, every, async and await (this topic is explored later in the Concurrency section).

Refs have posses an very important feature which helps to enable the above. When we wish to perform an operation on a ref variable, or pass a ref to a function which is not expecting a ref - the ref will be unassigned. But, if no value has yet been assigned to the ref further execution will be blocked until a value is available.

It is for this reason that the following code will fail at compilation time with an error:

```
a int:
```

```
b = a //fails as a has not been assigned and and we will never be able to
    unassign from a to b, causing us to wait forever
```

Type wise, when unassigning a ref with type `X:`, the type of the value returned will be `X`. For `int:` this is `int`. For example:

```
a int:

{a = 99}! //execute our ref assignment in a separate isolate

b int = a //a is unassigned when our isolate spawned above completes execution
```

If we wish to prevent this unassignment we can use the `:` operator when referencing the variable. For example:

```
a int: = 12
b = a:
c := a

//the &== operator tests for equality by object reference and not by object
    value as the == operator does
assert b: &== a: //a and b are references to the same ref
assert c: &== a: //a and c are references to the same ref
```

3.13 Actors

Actors are covered in more detail in the Actors section. In summary, they are a hybrid between isolates and objects.

Method calls on actor objects occur within one dedicated isolate for the actor. Only one method is executed per actor at any point in time, as such actors can be shared between isolates. Method calls to actor methods return refs. An example actor:

```
class NormalClass(counter = 0){
  def inc() => ++counter
}

anActor actor NormalClass = new actor NormalClass()
anotherActor = new actor NormalClass()//the type of anotherActor is inferred as:
    actor NormalClass

aref = anActor.inc() //aref is infered as type int:
```

All actors are a subtype of `actor` which is itself a shortcut for `com.concurnas.lang.Actor`:

```
class MyClass()

act1 actor = new actor MyClass()
act2 com.concurnas.lang.Actor = new actor MyClass()
```

3.14 Pass by value/reference

Concurnas applies pass by value or pass by reference contingent on the type being referred to. Primitive types are passed by value, everything else (including primitive type arrays) are passed by reference. Therefore the following conditions hold true:

Concerning primitive types, a copy of the value is made:

```
anint1 = 99
anint2 = anint1 //copy value of anint1 to anint2

anint2++ //increment anint2

assert anint2 == 100
assert anint1 == 99
```

We see above that the original value held by variable `anint1` is unchanged by the operation applied to the value of `anint2`.

However, the behaviour is different with objects:

```
class MyClass(~avar int)

mcvar1 = MyClass(99)
mcvar2 = mcvar1

mcvar2.avar++

assert mcvar1.avar == mcvar2.avar
```

We see here that the internal state of the object referred to by `mcvar1` is the same as that of `mcvar2` - this is because a reference has been passed to `mcvar2` upon assignment - a copy of the value of `mcvar1` is not made.

Sometimes this behaviour is undesirable and we'd want `mcvar2` to be a copy of the `mcvar1`. This can easily be achieved by tweaking our code above to include the copy operator `@`:

```
class MyClass(~avar int)

mcvar1 = MyClass(99)
mcvar2 = mcvar1@

mcvar2.avar++

assert mcvar1.avar <> mcvar2.avar
```

Above we see that the state of `mcvar1` and `mcvar2` is no longer the same, as they refer to different objects.

Nothing's gonna harm you
Not while I'm around
Nothing's gonna harm you
No sir, not while I'm around

- Not While I'm Around from The Broadway Album by Barbra Streisand (1985).

4. Null Safety

Concurnas, like most modern programming languages, has support within its type system for `null`, this is useful but comes with one specific danger... In conventional programming languages supporting `null`, since **any** object reference can be of type `null`, then there is the potential for error if a field access or method invocation is attempted on that null instance. In Java this manifests itself via the dreaded and ubiquitous `NullPointerException`. Given that any object can be null, at any point of execution, this error can occur anywhere. Thus, in order to write strictly safe code one is forced to validate one's object references as not being null on a persistent basis to be really sure that this potential for error is eradicated. This time consuming and labour intensive process is often either skipped entirely or only partially completed in most projects. Leading for the potential for error. In fact this problem is considered so severe that it's often referred to as the Billion Dollar Mistake.

So why not remove `null` from the type system entirely? Well, it turns out that for some algorithms having null is incredibly useful for representing uninitialized state. So we need to keep `null`. But we can manage its negative aspects.

Concurnas, like other modern programming languages such as Kotlin and Swift, makes working with `null` easy and safe by incorporating nullability a part of the type system, and by providing a range of null safety operators to assist in those instances where null is a desired object state. This allows us to write safe code in Concurnas which is largely free of `NullPointerException` exceptions.

4.1 Type system nullability

All non-primitive types in Concurnas are non-nullable at the point of declaration by default. Thus, attempting to assign a null value to a variable having a non-nullable type (or anywhere else where a non-nullable type is expected, for instance in a method argument etc) will result in a compile time error:

```
aVariable String = "a String"
aVariable = null//this is a compile time error!
```

If one wishes for a variable to be able to hold a value of null, a nullable type must be declared, this can be achieved for a non-primitive type by appending a `?` to the end of the type declaration. As per our previous example:

```
nullableVar String? = "a String"//nullableVar can hold a null value
nullableVar = null//this is acceptable
```

With nullability as part of the type system we are able to write code such as the following safe in the knowledge that there is no possibility for a `NullPointerException` exception to be thrown:

```
len = aVariable.length()
```

Attempting the same method call on `nullableVar` results in a compile time error, as `nullableVar` might be null:

```
len = nullableVar.length()//compile time error!
```

But of course not all instances of `nullableVar` are null (otherwise there is no value to the code above), so how can we call the `length` method? Lucky for us Concurnas has some clever logic to support working with nullable types and a a number of useful operators...

4.2 Smart null checking

If a nullable variable is checked for nullability within a branching logic block, i.e. an if statement, then the fact that it has been established as being not null will be incorporated to any usage of said variable within the body of the if statement. So the following is valid code:

```
res = if(nullableVar <> null){
nullableVar.length()//we've already established that nullableVar is not null
}else{
-1
}
```

This logic is reasonably comprehensive and can cater for complex cases such as the below, where we establish that a lambda is null in one if statement branch, and so can conclude that is must be non-null within another:

```
alambda ((int) int)? = def (a int) => a+10

res = if(alambda == null){
        0
      }else{//we've established that alambda is not null
        alambda(1)
}
```

This logic applies on an incorporated running basis within the if test, thus the following code is valid:

```
res = if(nullableVar <> null and nullableVar.length() > 2){
        "ok"
      }else{
        "fail"
}
```

Furthermore, in cases where a nullable variable is assigned a non-nullable value, we know with certainty that said variable is now not nullable, Concurnas is able to make use of this information thus enabling the following code to be valid:

```
nullable String? = "not null"
//... potentially other code...
ok = nullable.length() //ok as we know at this point nullable isn't nullable
```

This logic is applied to branching control flow:

```
a=2; b=10
nullable String? = null
if(a > b){
   nullable = "ok"
}else{
   nullable = "also ok"
}
ok = nullable.length() //ok as we know at this point nullable isn't nullable
```

This inference logic extends to class fields with the caveat that calls to methods after the determination of non nullability will invalidate that inference (since it's possible such a method call may set our field in question to null). Thus the following holds:

```
class MyClass{
   aString String?

   def foo(){
      aString = null
   }

   def inferNonNull(){
      aString = "ok"
      len = aString.length() //ok aString is not nullable
      foo()//foo may set aString to null
      len = aString.length() //error aString might be nullable
   }
}
```

This logic does not apply to shared nullable variables since they can be set to null by any isolate having access to them.

4.3 Safe calls

The safe call dot operator, `?.` allows us to execute the method or field access on the right hand side of a dot for a nullable entity if it is not null. If it is null, then null is returned. This the type returned from any call of this nature will always be nullable.

```
nullableVar String? = "a String"
len = nullableVar?.length() //len is of type Integer? (nullable Integer)
```

The safe call dot operator may only be applied to nullable entities, applying it to a non-nullable type results in a compilation error:

```
normalString String = "a String"
len = normalString?.length() //compilaton error
```

4.3.1 Array reference safe calls

Safe calls can be applied to array reference calls by inserting a `?` between the expression and array reference brackets `[]` as follows:

```
maybeNull int[]? = [1 2 3 4] if condition() else null //maybe null
got = maybeNull?[0] //got is of type Integer?
"" + got
```

4.3.2 Chained safe calls

Safe calls can be chained together, this is quite a common usage pattern:

```
enum Color{BLUE, GREEN, RED}
class Child(-favColour Color)
class Parent(-children Child...)

parent Parent? = Parent(Child(Color.GREEN), Child(Color.RED))

firstChildColor = parent?.children?[0]?.favColour //chained safe calls
```

4.4 Elvis operator

The Elvis operator[1], `?:` in Concurnas serves a similar purpose to the safe dot operator above in that it allows us to react appropriately to the case where the expression on the left hand side of the operator resolves to `null`. The difference with the Elvis operator is that when `null` is found, the expression on the right hand side is evaluated and returned:

```
nullableVar String? = null
len int = (nullableVar?: "").length()
```

4.5 No null assertion

The final option for working with nullable types in Concurnas is the no null assertion operator, `??`. This simply will throw a `NullPointerException` if `null` is found on the left hand side, otherwise it will return the value (which is guaranteed to be not nullable) of the left hand side. For example:

```
nullableString String? = "value"
//...
notNull String = nullableString?? //throws a NullPointerException if
    nullableString is null
```

The operator may be used on its own in order to test for nullness without returning a value:

```
nullableString String? = "value"
//...
nullableString?? //throws a NullPointerException if nullableString is null
```

[1] So called as the token looks like the emoticon for Elvis Presley

The no null assertion, like many other operators, may be used preceding a dot operator:

```
nullableString String? = "value"
//...
length int = nullableString??.length() //throws a NullPointerException if
    nullableString is null
```

4.6 Nullable generics

When it comes to generic types, at the point of declaration by default the are non-nullable, that is to say, the default upper bound of a generic type declaration is `Object` - which is not nullable. i.e. Below the `X` generic types of our `TakesGeneric1` and `TakesGeneric2` classes are the same:

```
class TakesGeneric1<X>
class TakesGeneric2<X Object> //Object is non-nullable
```

Since we have declared `X` above as being non-nullable the following code is not compilable:

```
inst = TakesGeneric1<String?>() //compile time error
```

We could however redefine `TakesGeneric1` in order to achieve this:

```
class TakesGeneric1<X?>
inst = TakesGeneric1<String?>() //this is ok
```

Classes having non-nullable generic types may still define nullable instances of those generic types as follows:

```
class TakesGeneric1<X>{
~x X?
}
inst = TakesGeneric1<String>()
inst.x = null //this is ok
```

4.7 Class field initialization

Class fields if not initialized at point of declaration or via the constructor invocation chain will default to null. Class fields like this must be declared as being nullable otherwise they will be flagged up as an error at compilation time. For example:

```
class UninitNonNullables<X>{
   aString String //initially set to null, but type not nullable - hence error!
   anArray int[] //initially set to null, but type not nullable - hence error!
}
```

This can be solved by either declaring the variables as being of a nullable type or initializing them:

```
class UninitNonNullables<X>(anArray int[]){ //initialized in default constructor
   aString String? //nullable
}
```

4.8 Using non-Concurnas types

Code written outside of Concurnas, for instance in Java, may not have any null safety. As such, by default Concurnas is somewhat conservative when it comes to invoking methods originating from languages other than Concurnas. Though this does not come at a sacrifice to what code can be used, it does come with some caveats.

Essentially, unless otherwise annotated (see Annotating non-Concurnas code below), the methods of non-Concurnas types are assumed to consume and return values of unknown nullability. That is to say, they are assumed to be nullable but can be used as if they were both nullable and non-nullable. For example, a list:

```
alist = new list<String>() //generic param of java.util.ArrayList declared as a
    non-nullable type
alist.add('inst')
alist.add(null) //this is ok
res1 = alist.get(0) //res1 is of nullable type: String?
res2 String = alist.get(0)
```

The above would not be possible if the generic parameter of `java.util.ArrayList` were to be known as either nullable or non-nullable - but it's unknown in this case. Let's look at what this causes in detail:

- `alist.add(null)` - it is acceptable for null to be passed for a type of unknown nullability.
- `res1 = alist.get(0)` - `res1` will be inferred to be a nullable type.
- `res2 String = alist.get(0)` - The value resulting from execution of the `get` method call will be checked to ensure that it is not null before being assigned to `res2` which has been declared as being non-nullable. This helps avoid "null pollution" - i.e. a `null` value being inadvertently assigned to a non-nullable variable.

In the above code where Concurnas attempts to avoid "null pollution" - if an unknown nullability type resolves to a null value and is set to a non-nullable variable (or say passed as an argument to a method expecting a non-null parameter) then this will result in a `NullPointerException`. If the `alist` variable were to be created with a nullable generic type: `alist = new list<String?>()` then this logic would not be required, though of course we'd be unable to assign the return value of `alist.get(0)` to the non-nullable variable: `res2 String`. Throwing a `NullPointerException` in order to avoid "null pollution" is not desirable, but necessary - and is a beneficial approach where the alternative is permitting "null pollution" because at least the `NullPointerException` is thrown at the point of assignment/usage.

4.8.1 Annotating non-Concurnas code

non-Concurnas code may be decorated with the `@com.concurnas.lang.NoNull` annotation in order to indicate a type is not nullable. Types may also be declared as being explicitly nullable. Here is an example of this in action for some Java code:

```
import com.concurnas.lang.NoNull;
import com.concurnas.lang.NoNull.When;
import java.util.List;

public static @NoNull List<@NoNull String> addToList(@NoNull List<@NoNull
    String> addTo, @NoNull String item ){
   addTo.add(item);
   return addTo;
}
```

```
public static @NoNull(when = When.NEVER) List<@NoNull(when = When.NEVER) String>
    addToListNULL(@NoNull(when = When.NEVER) List<@NoNull(when = When.NEVER)
    String> addTo, @NoNull(when = When.NEVER) String item ){
   addTo.add(item);
   return addTo;
}
```

Above, for the `addToList` method the following elements are tagged as non-null:

- The return type: `java.util.List`
- The generic type qualification of the return type: `String`
- The first input argument of type: `java.util.List`
- The generic type qualification of the first input argument: `String`
- The second input argument of type: `java.util.List`

The same applies for the `addToListNULL` except all the elements above are tagged as being nullable.

4.9 Nullable wrapper

Sometimes when working with objects having generic type parameters, it is not possible to qualify those parameters in a nullable manner, for instance with refs. This presents a problem if we wish to store a nullable type within such a container object. Concurnas provides the `com.concurnas.lang.nullable.Nullable` class, auto imported as `Nullable` to wrap around such as nullable type:

```
nullableinst = new Nullable<String?>():
nullableinst.set("ok")
maybenull String? = nullableinst.get()
```

4.10 Nullable method references

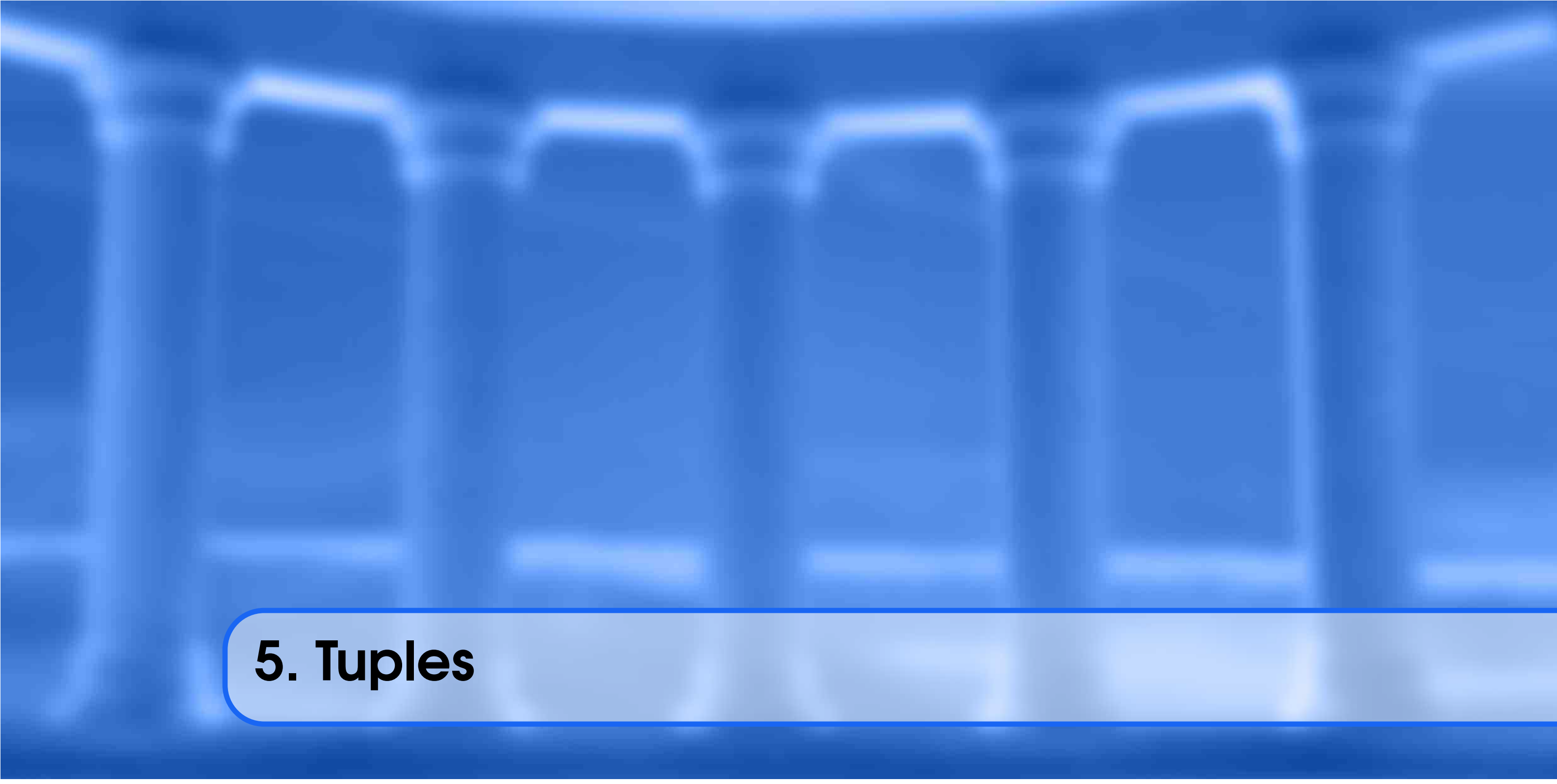
Method reference types can be declared nullable by nesting their declaration within parenthesis and affixing a `?` as per normal. For example:

```
nullLambda ((int) int)? //nullable method reference
```

4.11 Where can NullPointerException's still occur

There are some limited circumstances in which `NullPointerException` may still occur:

- **When using non-Concurnas types or calling non-Concurnas code.** Covered above.
- **When using the no null assertion.** Covered above.
- **Values from non-Concurnas code.** Concurnas attempts to avoid "null pollution" - as such it's possible for a `NullPointerException` exception to be thrown from checking that the return value of a unknown nullability type is non-null. This "null pollution" avoidance is not applied to the individual values of arrays as it would not be efficient to check every array in this manner. As such non-primitive arrays and n+1 dimensional arrays provided by non-Concurnas code should be used with care and attention to the possibility of null values being present within them.
- **When using non-Concurnas code.** Concurnas has no control over code authored in other languages, so they may throw a `NullPointerException`.



5. Tuples

The syntax for creating tuple types is defined as:

```
'(' type (','tuple)* ')'
```

In order to create an instance of a tuple we need only separate the individual tuple values with a comma, the syntax being:

```
expression (','expression)+
```

The maximum number of elements within a tuple is 24.

Tuples are an excellent tool for working with collections of typed data, whilst avoiding the need to define a container class specifically to hold it, or by having to use a list. For example:

```
ageAndName (int, String) = 24, "dave" //a tuple definition
ageAndNameAlt Tuple2<int, String> = 24, "dave" //a tuple definition with
    alternative mechanism for declaring tuple type
ageAndName2 = 67, "mary" //tuple type is inferred to be (int, String)
```

Without tuples, the above would have to be captured in the following ways (amongst others):

```
class AgeAndNameHolder(-age int, -name String)//using a holder object
ageAndName = new AgeAndNameHolder(24, "dave")

ageAndName2 = [67, "mary"] //holding the data within a list
```

The holder object can become a very heavy weight pattern to use in code, especially for cases where the data lifetime or manipulation context is very short (contained within a method for instance). Likewise, the list pattern is inconvenient as we must remember to explicitly cast got values and it's mutable, meaning the contents can be accidentally changed. Tuples provide a nice middle ground between the two, as much type safety as classes, but with as much convenience in defining as lists.

A nice advantage of tuples is that they are naturally immutable - once created they cannot be changed (note that the objects they contain, if not immutable can be changed).

Tuples are often used for returning multiple values to or from a function or method invocation:

```
def getAgeAndName(){
   return 12, "dave"
}

def getAgeAndName2() (int, String) {//with explicit return type definition
   return 12, "dave"
}
```

Tuples are often used to return multiple values from functions as illustrated in the `getAgeAndName2` function above.

5.1 Tuple decomposition

We can extract values from tuples in one of two ways. We can explicitly refer to the field being extracted:

```
x = 12, 14
e1 = x.f0
e2 = x.f1
```

Or we can use a decomposition assignment:

```
x = 12, 14
(e1, e2) = x
```

If only certain values need to be extracted this can be achieved by simply following the space to ignore with a comma:

```
x = 12, 14, 15
(e1, , e3) = x

//e1 == 12
//e3 == 15
```

We can make use of the decomposition assignment within for loops like so:

```
toSum = [(1, 2), (3, 4), (5, 6)]

for((a, b) in toSum ) { a + b }
//== [3, 7, 11]
```

This same effect can be achieved within list comprehensions of tuples:

```
toSum = [(1, 2), (3, 4), (5, 6)]

(a+b) for (a, b) in toSum
//== [3, 7, 11]
```

5.2 Iterating over Tuples

Tuples support iteration (i.e. can be used in for loops etc):

```
x = 12, 13
```

```
combined = "" + ("el: " + e for e in x)

//combined == "el: 12 el: 13"
```

5.3 Tuples are Reified Types

Tuples are reified types, which means that the following is perfectly valid code:

```
def extractAgeIfNameAndAge(an Object){
   if(an is (int, String)){
      (age,) = an
      age
   }
   -1
}
```

5.4 Tuples in multiple assignment

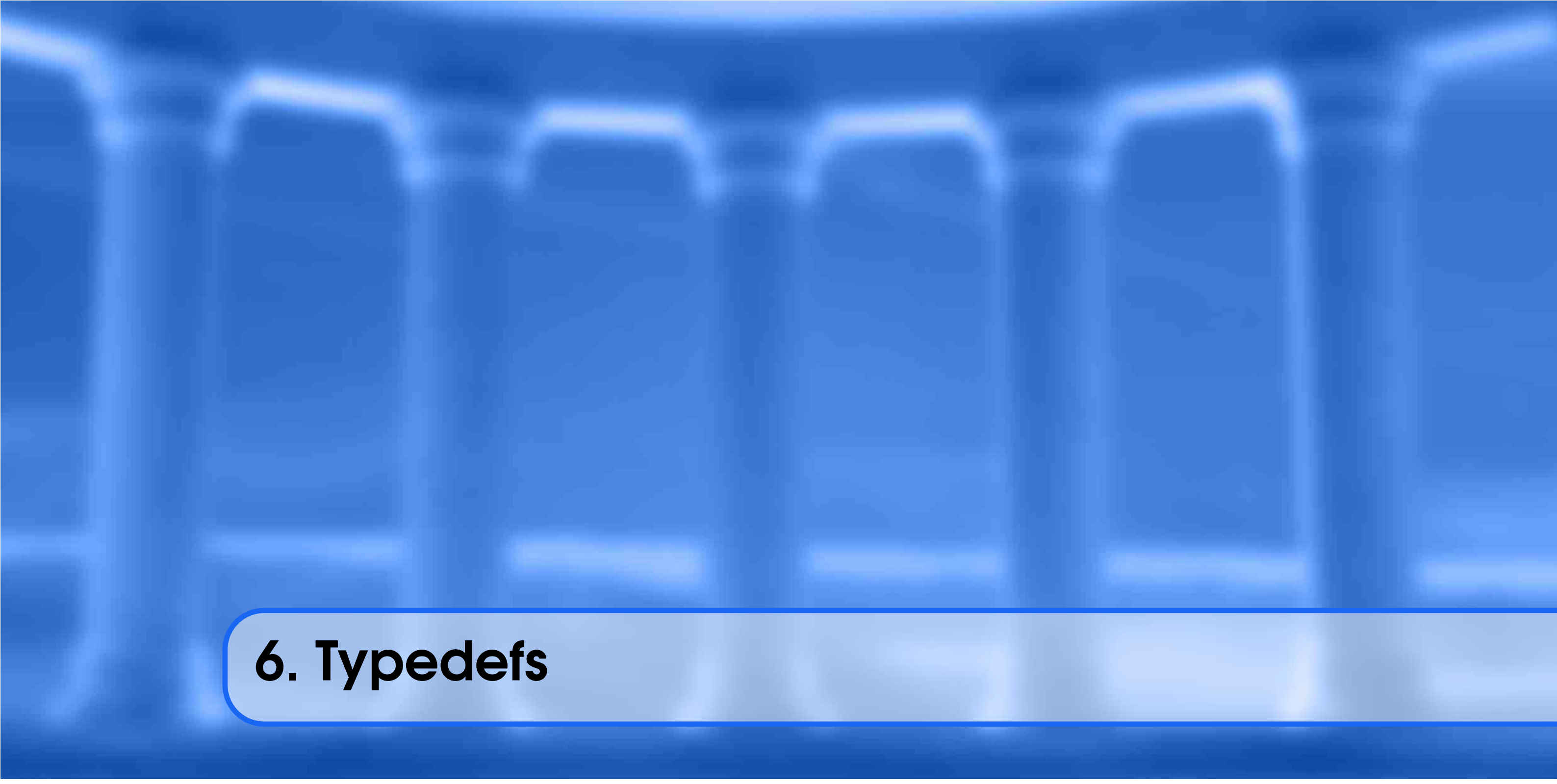
Tuples can be used in multiple assignment statements, e.g:

```
(a, b) = (c, d) = 1, 2

//a == 1
//b == 2
//c == 1
//d == 2
```

5.5 Tuples in typedefs

Tuples can be used in typedef statements. E.g.

```
typedef tuple3 = (int, String, double)
typedef myTuple2<x> = (int, x)//with one parameter: x
```



6. Typedefs

Concurnas provides a convenient mechanism for referring to type definitions. This is particularly useful for long qualified generic types. Typedefs can be used anywhere one would normally use a type. They enable us to translate the following, very verbose code:

```
ArrayList<Set<HashMap<ArrayList<String>, Set<int>>>> myDataStructureInstance =
    new ArrayList<Set<HashMap<ArrayList<String>, Set<int>>>>(10)
```

Into the following, far easier to read (but functionally identical) form:

```
typedef DataStructure = ArrayList<Set<HashMap<ArrayList<String>, Set<Integer>>>>

myDataStructureInstance DataStructure = new DataStructure(10)
```

This greatly improves the readability of code as well as increasing code reuse and reducing errors since less code needs to be written.

Typedefs go beyond traditional macro or type macro offerings seen in earlier languages in providing quantifiable type defs, this is particularly useful when defining typedefs on generic types where we wish to defer the generic type qualification. The syntax should feel familiar to that for using generic types:

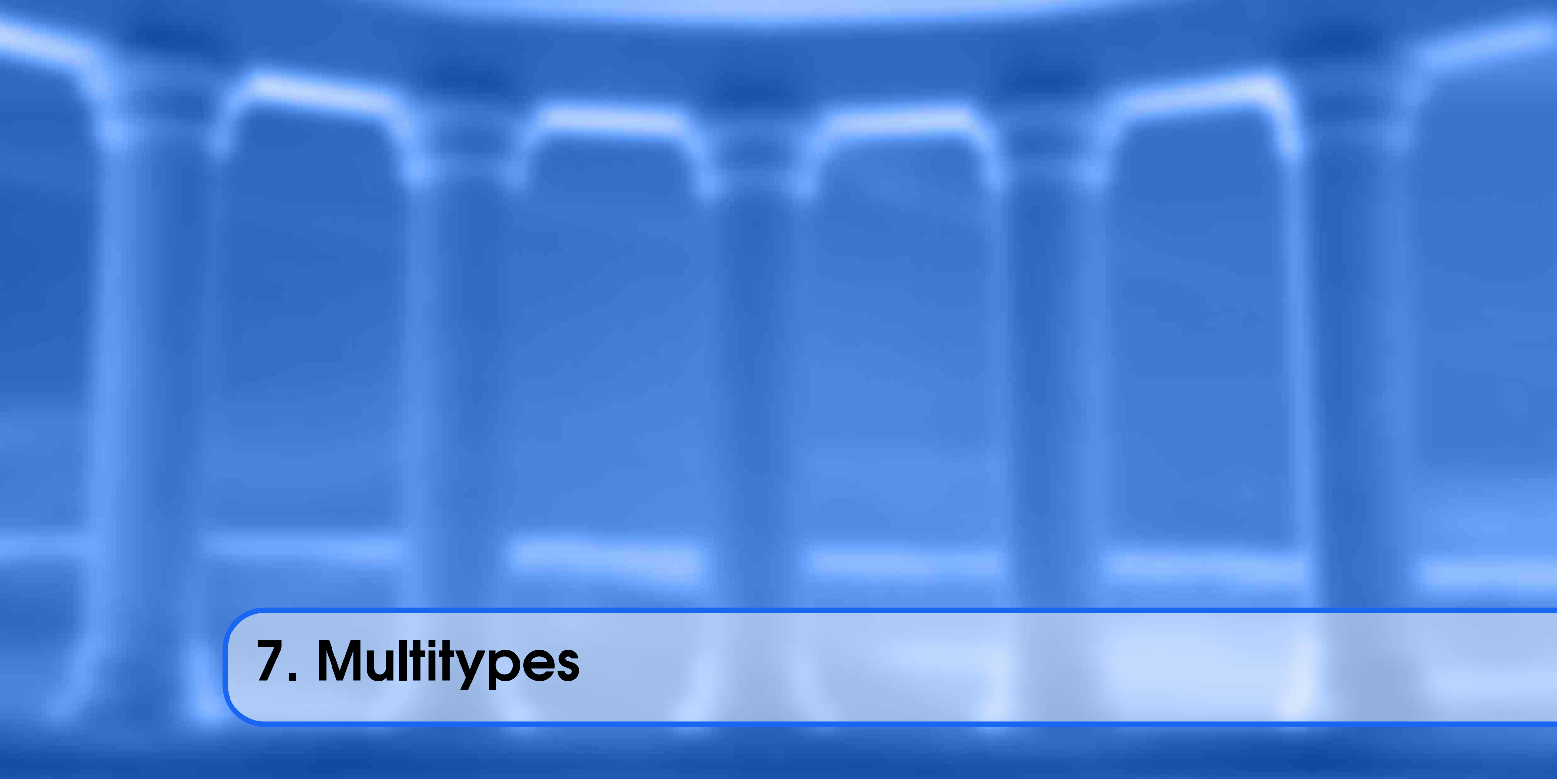
```
typedef MyHashMap<X, Y> = HashMap<X, Y>

ml = new MyHashMap<int, String>()
```

Typedefs may refer to other typedefs like so:

```
typedef MyList<X> = ArrayList<X>
typedef StringList = MyList<String>
```

Typedefs may also refer to other typedefs during qualification of generic types:

```
typedef MyHashMap<X, Y> = HashMap<X, Y>
typedef MyList<X> = ArrayList<X>

item = MyList<MyHashMap<int, String>>()
```



7. Multitypes

When writing code, it is generally best to avoid code duplication (especially copy-paste) as much as possible. However, when working with primitive types as function inputs in particular, some code duplication is normally necessary in order to support the same operation on multiple different types. In Concurnas, multitypes solve this problem meaning that you need only define said operation once.

We can specify a multitype in a function, method or constructor definition by delimiting our input types or return type parameters by a `|`. A multitype of `int`, `long` and `String` would look like: `int|long|String`.

For example, let's say that we wish to write a function which adds `10` to each element of a matrix, in place, and that we wish to support this operation for integer and double matrices. With multitypes this code looks like:

```
def add10(inp int|double) => inp+10
```

If multitypes were not an option this code would need to use copy-paste duplication and would be written as the less readable and harder to maintain:

```
def add10(inp int) => inp+10
def add10(inp double) => inp+10
```

Multitypes are not restricted to be composed of just primitive types, but rather, any type element may be used, including typedef references and multitypes. Typedef themselves may also refer to multitypes:

```
typedef numerical = short|int|long|float|double|char|byte

def addTen(inp numerical|String)
   => inp + 10 //multittype input parameter defined over short, int, ..., byte
       as well as String!
```

There are a number of commonly used multitype typedefs defined in `com.concurnas.lang.multitype`

Multitypes can only be used in the definition of function, method and constructor input parameters or return parameters as well as the type qualifier for extension functions and in the bodies of these. There may be more than one multitype parameter to a function, method or constructor or extension function qualifier, but each multitype parameter must be of the same size, likewise, any referenced inside the body must also be of the same size. Multitype return parameters can only be used in conjunction with at least one multitype input parameter:

```
def adderOk(a int|long|short, b int|long|short)
   => a+b

def adderFail(a int|long|short, b int|long)
   => a+b //mismatch in multitype input parameter size
```

Multitypes can be used to qualify extension functions:

```
def int|double|float addTen() => this += 10
```

Any they may be used inside bodies of methods, functions and constructors like so:

```
def repeater(inp int|String) {
   ret = new int|String()
   ret[0] = inp
   ret[1] = inp
   return ret
}
```

8. Casting and Checking Types

Casting and checking of types enable us to take one type and either check it can be treated as another (via `is` and `isnot`) or actually be treated as another type (via `as`). Intuitively this sounds like not something of much use, however consider the case where an object of type `java.lang.Object` is passed to a function which expects an `java.lang.Object` argument, but, contingent upon the type of said object uses that object differently. For example:

```
def doSomething(an Object){
   if(an is String){
      aString = an as String
      //...
   }else if(an is Integer){
      anInt = an as Integer
      //...
   }
}
```

To implement the above we needed to use both `is` and `as`[1].

8.1 Primitive type Implicit and Explicit conversions

Generally speaking we can use a lower bit width value where a larger bit width value is expected (e.g. in assignment). For example we can implicitly 'upcast 'like this:

```
anint int =123
along long = anint //implicity 'upcast' to a long
```

We see above that the value of `anint` is converted ('upcast') to a long value. Of course, this makes the following code possible:

```
expectLong long = 123 //123 is converted to a long value
```

[1] Actually, as we shall see in the Is with automatic cast section later on, the as cast was unnecessary.

```
def operateOnLong(along long) => along+1

operateOnLong(123) //123 is converted to a long value
```

Note however that if we were to attempt a ’downcast’ it would be considered a compilation error, as there is the potential for data loss (imagine a long value larger than $2^{31}-1$ being stored in an int variable, it’s not possible as we don’t have enough bits of information):

```
along = 100l
anint int = along //compilation error!
```

The way to achieve the above is to use an explicit cast in order to convert the value:

```
along = 100l
anint int = along as int
```

But generally speaking this is not considered a good idea.

8.1.1 Casting boxed types

Boxed types may be explicitly cast to other boxed types follows:

```
anInt = Integer(20)
asDouble = anInt as Double
```

This conversion can take place on an implicit basis as long as the cast operation is an ’upcast’ (e.g. `Integer` to `Long`, `Integer` to `Double` etc). So the following is perfectly acceptable:

```
def expectsDouble(an Double) => an

anInt = Integer(20)
expectsDouble(anInt)//this is ok the Integer is 'upcast' to Double
```

But of the following, representing a ’downcast’ is only possible via an explicit cast operation:

```
def takesInteger(an Integer) => an

aDouble = Double(0.32)
takesInteger(aDouble) //this is not ok as an implicit 'downcast' is required,
    potentially losing data
takesInteger(aDouble as Integer) //this will work, though is inadvisable
```

8.2 Primitive type operators

Something that often catches out new programmers is the following scenario, say we wish to perform division, we’d use code like the following and expect to get the result `0.25`:

```
1/4
```

But actually we’d get the result 0. This is because, by virtue of the fact that both our input primitive types to the division operator are integer, whole number division is applied. If we wish to perform fractional division then we’d need to convert at least one of the inputs to the division operator a floating point number, this is easy to achieve:

```
1/4.
```

Now we get the desired result: `0.25`

8.3 checking and casting for non-primitive Types

Any object subtype may be implicitly upcast to a supertype. Also, as has been touched upon earlier all non-primitive types are subclasses of type `java.lang.Object`. As such this is perfectly valid code:

```
open class SuperClass
class ChildClass < SuperClass

inst1 SuperClass = new ChildClass()
inst2 Object = new ChildClass()
```

In cases where we are converting to a non supertype, then the `is` and `isnot` operators come into play as follows:

```
something Object = "hi"

assert something is String
assert something isnot Integer
```

Once we are sure of the type (either by using `is` or by another means) we may wish to cast the object to the type of interest such that we can use it as such (i.e. access fields call methods etc):

```
something Object = "hi"
asString = something as String
```

When it comes to objects with generic types, we cannot perform a cast on generic type qualifiers. As such types requiring generic type qualification cannot be expressed with qualifiers:

```
class MyGenClass<X>

anObj Object = MyGenClass<String>()

check = anObj is MyGenClass<String> //not valid
convert = anObj as MyGenClass<String> //also not valid

okcheck = anObj is MyGenClass<?> //ok
okconvert = anObj as MyGenClass<?> //ok
```

Function refs, tuples, refs and actors are reified types which makes code like the following possible:

```
afuncRef Object = def (a int, b int) => a+b

checkfuncRef = afuncRef is (int, int) int
asafuncRef = afuncRef as (int, int) int


aTuple Object = 12, "hi", false
```

```
checkTuple = aTuple is (int, String, bool)
asaTuple (int, String, bool) = aTuple as (int, String, bool)


aRef Object = 12:

checkRef = aRef is int:
asaRef = aRef as int:


anActor = actor String()

checkActor = anActor is actor String
asaActor = anActor as actor String
```

We can cast an array to be a supertype (unlike lists - due to the restrictions of generic types previously outlined), but care must be taken when assigning values to said array since upon assignment type of the value will be checked so as to ensure that it matches the real component type of the array. This is because when we are casting Objects (which arrays are - they are a Subtype of `java.lang.Object`) we are treating them as another type, we are not actually transforming the type itself, thus an `int[]` array stays as an `Integer[]` array when we cast it to an `Object[]`.

```
anArray Object[] = [ 1 2 3 4 5 6 ] as Object[] //this is ok
anArray[0] = 99 //this is ok as the type of the array is really Integer[]
anArray[1] = "hi" //this is not ok as we cannot set a String value in an
    Integer[] array
```

A variable holding a value null can be cast to anything, it will remain null:

```
nullObj Object = null
stillnull String = nullObj as String
```

Null is considered an instance of any object type:

```
nullObj Object = null

assert nullObj is Integer
assert nullObj is String
```

8.4 Casting Arrays

Although it is not possible to directly cast an n dimensional array of primitive type to a differing n dimensional array of primitive type. The effect can nevertheless be achieved via use of vectorization. This will of course create a new n dimensional array:

```
vec = [ 1.0 2.0 ; 3.0 4.0]
res = vec^ as int
/
res = > [ 1 2; 3 4 ]
```

8.5 Is with automatic cast

`is` with automatic cast is a massive time saving feature of Concurnas in cases where one has already tested to ensure a variable is of a certain type and now needs to perform operations on said variable with it as cast to the type tested for. If the `is` operator is used on a variable in an `if` or `elif` test, then throughout the remainder of the test and the block associated with the test that variable will be automatically cast for us as the type checked against. Example:

```
class Dog(~age int)

def getAge(an Object){
   if(an is Dog){an.age} //an is automatically as cast to type Dog. Thus we do
       not need an explicit cast
   else{"unknown"}
}

getAge(Dog(3)) //returns 3
```

This also applies to if expressions:

```
class Dog(~age int)

def getAge(an Object){
   an.age if(an is Dog) else "unknown" //an is automatically as cast to Dog
}

getAge(Dog(3))//returns 3
```

The automatic cast resulting from the is test applies to the remaining body of an if test:

```
class Treatment(~level int)

def matcher(an Object)
   => 'match' if an is Treatment and an.level > 5 else 'none'

[matcher(Treatment(7)), matcher(Treatment(2))] //resolsves to [match, none]
```

Note that this only applies if the is operator is used on a variable. e.g. This does not apply to values resulting from function calls, array indexes etc.

This also does not apply to is operators used within tests which can be short circuited if the is is referenced past a short-circuit point. For example, the following is invalid:

```
class Dog(~age int)

maybe = true

def getAge(an Object){
   if(maybe or an is Dog){an.age} //compilation error as required as cast to be
       used in this way
   else{"unknown"}
}

getAge(Dog(3))
```

9. The class keyword

The `.class` keyword can be used in order to obtain a `java.lang.Class<?>` object of a certain class. The `.class` call is placed after the class name:

```
classvar = String.class
```

On instances of classes, calling `getClass()` will return a `java.lang.Class<?>` object:

```
aString = ""
classvar1 = String.class
classvar2 = aString.getClass()

assert classvar1 == classvar2//these both point to the same Class object
```

This also works for primitive types and arrays:

```
iclass = int.class
aclass = int[].class
```

There are a number of useful methods, mostly concerning reflection, which exist on instance objects of `java.lang.Class`. For more information see the JDK documentation.

10. Imports

When writing software it is generally considered best practice to separate our code into differing physical or logical files, generally with code relating to the same sub problem in the same file or directory structure. This improves code reuse and maintenance. Then, when we wish to make use of code defined in a separate module (likely within a package) we can make use of the variants of import statement which Concurnas supports, described herein.

We can import the following assets from other files: functions (including extension functions), classes, module variables (including those referring to function references), typedefs, enums and annotations.

Imports are essentially referential sugar. They help us to avoid the need to type the fully-qualified path of the asset we are making use of, instead being able to refer to the asset by its short name (the text after the full package name final dot) or by a name we assign at import point (via the `as` keyword). At compile time any references to the short name we are making use of are mapped to the fully-qualified path before being passed to the classloader (default or otherwise) for loading. For example, a fully-qualified package name for class `MyClass` might be: `com.mycode.MyClass`, the short name is `MyClass`.

Since Concurnas can run on the JVM, it is compatible with all other code compiled in JVM languages such as Scala, Kotlin, Clojure etc and of course Java. All that is necessary to import other JVM compiled code is is the compiled class itself (and appropriately supporting classloader).

There are three supported import statements. All but the star import may use the `as` clause to override the short name of the imported asset. In the following examples we import a class asset, but this may just as well be a function, module variable etc:

10.1 Import

```
import com.mycode.MyClass //MyClass will be imported with a short name of:
    MyClass
import com.mycode.MyClass as ImportedClass //using 'as' enables us to override
    the short name of the asset imported
```

Recall that importing an asset allows us to use it within having to refer to the fully-qualified name of the asset. So the following are equivalent:

```
import com.mycode.MyClass as ImportedClass //using as enables us to override the
    short name of the asset imported

inst1 = new ImportedClass()//mapped to: com.mycode.MyClass behind the scences
inst2 = new com.mycode.MyClass()
```

10.2 From Import

From import is particularly useful where we wish to import more than one asset from the same package path:

```
from com.mycode import MyClass
from com.mycode import MyOtherClass, MyOtherClass2

//As with conventional import we can override the short names of the imported
    assets:
from com.mycode import MyClass as ImportedClass
from com.mycode import MyOtherClass as ImportedClass, MyOtherClass2 as
    ImportedClass2
```

10.3 Star Import

If we wish to import all assets under a package name path, then we can use the star notation:

```
from com.mycode import *
import com.mycode.*
```

Import star should be used with careful consideration as it can easily cause problems with overuse as short names may conflict with one another.

10.4 Import sites

The variants of the aforementioned import statements may be used at any point in Concurnas code. They follow the normal scoping rules:

```
def myfunc(){
   from com.mycode import MyClass
   mc = MyClass() //MyClass can now be used within the { } and any nested scopes
   if(acondition){
      mc2 = MyClass() //MyClass may be used here
   }
}
//MyClass may not be be used from this point onwards as this is outside the
    imported scope
```

Most of the time however, convention dictates that imports are best placed at the top of a module code file for global usage inside said module.

10.5 Using imports

The `import` statement is "side effect free" - that is to say that no code will be directly run at the point in which an asset is imported, instead this only takes place when the asset is used for the first time. This behaviour is in contrast to the likes of Python which, at the point where code is imported, any top level code present within the imported script will be executed.

It is not essential to use import statements in one's code, one could simply refer to the fully-qualified paths of the assets of interest. For example, instead of importing `java.util.ArrayList` for use in a new object instantiation, one could just use the fully-qualified name, i.e: `mylist = new java.utils.ArrayList<String>()`

10.6 Packages

An asset's fully-qualified importable package name is a function of its name within the module it's declared within, the module name and its path relative to the root of compilation at compilation time. Note that Concurnas does not have a package keyword, instead it relies upon the directory structure relative to the root of compilation in order to determine this. So for example, using a conventional directory structure (found in almost all operating systems) when we compile our code if our root was set to `/home/project/src` and our code within this root, in a directory structure `./com/mycode.conc` (i.e. file `mycode.conc` is within subdirectory `com` containing the class definition `MyClass`) - then the fully qualified package name of the class at compilation time would be `com.mycode.MyClass`.

10.7 Accessibility Modifiers

Accessibility Modifiers play an important role in determining what assets can and cannot be imported. The following chart shows what can be imported from where:

Accessibility Modifier	Importable from
`public`	public assets may be imported into anywhere
`package`	package assets may be imported into modules within the same package/module only
`protected`	protected assets may be imported into modules within the same package/module only
`private`	Assets marked as private may not be imported

10.8 Default imports

The following packages are imported by default for all Concurnas code. Thus the short names of the Classes within these paths are directly usable within Concurnas code without an explicit import being required:

- `java.lang.*`
- `com.concurnas.lang.*`
- `com.concurnas.lang.datautils.*`

10.8.1 Prohibited imports

There are some Classes which one may not directly use in ones Concurnas code for various practical reasons:

- `java.lang.Thread`

- `com.concurnas.runtime.cps.IsoRegistrationSet`
- `com.concurnas.runtime.ConcImmutable`
- `com.concurnas.runtime.ConcurnificationTracker`
- `com.concurnas.bootstrap.runtime.cps.Fiber`
- `com.concurnas.lang.ParamName`
- `java.util.concurrent.ForkJoinWorkerThread`
- `com.concurnas.bootstrap.runtime.InitUncreatable`

11. Operators

Operators are an integral part of modern programming, Concurnas dedicates much of its syntax to supporting operators. Most of these operators are 'universal' and widely used outside of conventional programming, such as those concerning arithmetic and basic mathematics, some are more specific to programming in general such as bit shift related operators and some are unique to modern programming languages such as the null safety related operators.

Concurnas also dedicates much support to operator overloading, this helps facilitate the construction of domain specific languages within Concurnas.

11.1 Supported Operators

Concurnas supports the following extensive set of operators:

Class	Operator	Token	Description	Example	Input types	Return type	Associativity
Sizeof	Sizeof	sizeof	Calculate the off heap size of the operand	sizeof "myString" => 128	Object	Long	Left to right
Contains	In	in, not in	Check if the left hand operand is in the right hand operand	1 in [1 2 3 4 5] => true	Any Array or Collection of Component type	Boolean	Left to right
Not	Not	not	Flips boolean value of right hand operand	not true => false	Boolean	Boolean	Right to left
Postfix	Increment	++	Increments operand by 1	expr++	Integral	Integral	Left to right
	Decremement	–	Decrements operand by 1	expr–	Integral	Integral	Left to right
Prefix	Increment	++	Increments operand by 1	++epxr	Integral	Integral	Right to left
	Decremement	–	Decrements operand by 1	–expr	Integral	Integral	Right to left
	Positive	+	Positive value of operand	+10 => 10	Integral	Integral	Right to left
	Negative	-	Negative value of operand	-10 => -10	Integral	Integral	Right to left
Multiplicative	Multiplication	*	Multiplies left and right hand side operands	2*2 => 4	Integral	Integral	Left to right
	Power	**	Raises left operand to the power of right operand	3**2 => 9	Integral	Integral	Right to left
	Division	/	Divides left hand operand by right hand operand	10/3 => 3	Integral	Integral	Left to right
	Modulus	mod	The remainder of the division operator applied to the operands as above	10 mod 3 => 1	Integral	Integral	Left to right
Additive	Addition	+	Adds the left and right hand side operands	2 + 2 => 4, "hi" + "there" => "hi there"	Integral or String	Integral or String	Left to right
	Subtraction	-	Subtracts the right hand operand from the left hand operand	2 - 4 => -2	Integral	Integral	Left to right
Bitshift	Left shift	«	The left operands value is moved left by the number of bits specified by the right operand.	0b001 « 2 => 4 (0b100)	Integral	Integral	Left to right
	Right Shift	»	The left operands value is moved right by the number of bits specified by the right operand.	0b100 » 2 => 7 (0b111)	Integral	Integral	Left to right
	Unsigned right shift	»>	As above but shifted values are filled with zeros	0b100 » 2 => 1 (0b001)	Integral	Integral	Left to right
	Complement	comp	Complement of a single expression	comp 0b0001 => 14 (0b1110)	Integral	Integral	Right to left
Relational	Less than	<	Check if the left hand operand is less than the right hand operand	2 < 4 => true	Integral	Integral	Left to right
	Greater than	>	Check if the left hand operand is greater than the right hand operand	2 > 4 => false	Integral	Integral	Left to right
	Less than or equal	<==	Check if the left hand operand is less than or equal to the right hand operand	2 <== 2 => true	Integral	Integral	Left to right
	Greater than or equal	>==	Check if the left hand operand is greater than or equal to the right hand operand	2 >== 2 => true	Integral	Integral	Left to right
Instance of	Instance of	is, isnot, is not	Check if the left handoperand is (or is not) a subtype of the right hand type operand	expr is not Object => false	Object, Type	Boolean	Left to right
Cast	Cast	as	Cast left operand to be type of right hand type operand	expr as MyClass => expr is now of type MyClass	Object, Type	Object	Left to right
Equality	Structurally equal	==	Check if the value of the left and right hand operands are equal	10 == 10 => true	Any	Boolean	Left to right
	Structurally not equal	<>	Check if the value of the left and right hand operands are not equal	10 <> 9 => true	Any	Boolean	Left to right
	Referential equality	&==	Check if the object on the left and right hand operands are the same	Integer(1) &== Integer(1) => false	Any	Boolean	Left to right
	Referentially not equal	&<>	Check if the object on the left and right hand operands are not the same	Integer(1) &<> Integer(1) => true	Any	Boolean	Left to right
Bitwise	Bitwise and	band	Compares each bit of the operands and returns 1 for each if they are both 1	0b110 band 0b101 => 0b100	Integral	Integral	Left to right
	Bitwise or	bor	Compares each bit of the operands and returns 1 for each if either equals 1	0b110 bor 0b101 => 0b111	Integral	Integral	Left to right
	Bitwise exclusive or	bxor	Compares each bit of the operands and returns 1 for each if either but not both equals 1	0b110 bxor 0b101 => 0b011	Integral	Integral	Left to right
Logical and	and	and	Check if the left and right hand operands both resolve to true	true and false => false	Boolean	Boolean	Left to right
Logical or	or	or	Check if either the left and right hand operands both resolve to true	true or false => true	Boolean	Boolean	Left to right
Ternary	If expression	x if y else z	Return x operand if y operand resolves to true otherwise return z operand	8 if true else 9 => 8	Any, Boolean, Any	Any	Left to right
Invoke	Invoke	obj(args?)	Call the invoke method on object with provided methods	afunc(); afunc2(1, 'arg2')	Object	Any	Left to right
Null Safety	Not null assertion	??	Throws a Null Pointer Exception if left hand side is null, otherwise returns no null type	maybeNull??	Object?	Object	Left to right
	Elvis operator[1]	?:	Returns left hand side if it is not null, otherwise the right hand side expression will be evaluated and returned	maybeNull?:otherwise	Object?	Object	Left to right
Assignment	Assignment	=	Vanilla assignment	a = 10	Integral	Integral	Right to left
Compound Assignment	Addition Assign	+=	Apply addition operator to left and right operands and set value to left operand	a += 10	Integral or String	Integral or String	Right to left
	Subtract Assign	-=	As above but subtraction	a -= 10	Integral	Integral	Right to left
	Mutliply Assign	*=	As above but multiplication	a *= 2	Integral	Integral	Right to left
	Divide Assign	/=	As above but division	a /= 13	Integral	Integral	Right to left
	Power Assign	**=	As above but power	a **= 2	Integral	Integral	Right to left
	Mod Assign	mod=	As above but mod	a mod= 15	Integral	Integral	Right to left
	Bitwise And Assign	band=	As above but bitwise and	a bor= 16	Integral	Integral	Right to left
	Bitwise Or Assign	bor=	As above but bitwise or	a bor= 17	Integral	Integral	Right to left
	Bitwise Xor Assign	bxor=	As above but bitwise xor	a bxor= 17	Integral	Integral	Right to left
	Complement Assign	comp=	As above but complement	a comp= 19	Integral	Integral	Right to left
	Left shift Assign	«=	As above but left shift	a «= 20	Integral	Integral	Right to left
	Right Shift Assign	»=	As above but right shift	a »= 21	Integral	Integral	Right to left
	Unsigned right shift Assign	»>=	As above but unsigned right shift	a »>= 22	Integral	Integral	Right to left

The majority of these operators operate upon primitive or boxed primitive types. The integral

[1] So called as the token looks like the emoticon for Elvis Presley

(or 'numerical') types are defined as: `int`, `long`, `short`, `char` and `byte` and their equivalent boxed object variants: `Integer`, `Long`, `Short`, `Character` and `Byte` respectfully.

11.1.1 Sizeof

The `sizeof` operator is designed for working with off heap memory. It provides us an indication of the size in bytes that any object (including arrays) will consume when serialized into a byte format as par the scheme referenced within its optional qualifier. For example, when working on the gpu we can use a qualifier as follows:

```
leAr = [1 2 3 4 5 6 7 8 9 10]
gpusize = sizeof<gpus.gpusizeof> leAr//size of leAr on the gpu
```

If no qualifier is provided then the default off heap serialization built into Concurnas is used:

```
leAr = [1 2 3 4 5 6 7 8 9 10]
gpusize = sizeof leAr//size of leAr as par default scheme
```

Defining a sizeof qualifier

If we were to design a customized object serialization scheme, it can be beneficial to implement a sizeof qualifier such that clients of our custom scheme are able to know the size in bytes that objects will consume if serialized via our scheme. This can be easily achieved by defining a function taking a single Object as input and returning a value of type `int`. For example:

```
def sizeofCustomScheme(some Object) int{
   //some logic...
   return 1//in this example every object is 1 byte large
}
```

This can then be used via qualification of the sizeof operator as par above.

11.1.2 Contains

The contains class of operator consists of two variants, `in` and `not in`. `in` can be used with common data collection structures including: lists, sets and arrays to test to see if a value on its left hand side is within the expression of one of the aforementioned data structures on the right hand side. If it's present a `boolean` value of `true` is returned, otherwise `false`. `not in` returns `true` if the value on the left is not present within the data structure.

```
alist = [1,2,3,4,5]
within = 2 in alist
```

`in` may be used in a similar way with maps in order to examine whether a key is present within the map. For example:

```
amap = {2 -> 23, 3 -> 67. 1 -> 55}
within = 2 in amap
```

The `in` keyword has special meaning when used within the context of an iterator style for loop, this is documented in the Iterator style for section.

Supporting contains

Any class may support the operator `in` by providing a contains method, this is discussed in more detail in the operator overloading section.

11.1.3 Not

The not keyword simply flips the booleanarity of a value of boolean type. true becomes false and vice versa.

11.1.4 Postfix

Concurnas supports the usual set of postfix operators one would expect from a modern programming language. Namely the increment ++ and decrement -- operators appearing on the right hand side of an integral expression consisting of a variable, list or array reference. These operators are applied in place. The operators may be used in a place where a value is expected to be returned from it, in which case the previous value of the variable prior to the increment or decrement operator being applied will be returned:

```
avar = 10
anarray = [1 2 3 4 5]

avar++
anarray[0]--

prevvar = avar++
prevarray = anarray[0]--
```

11.1.5 Prefix

The prefix operators behave in a similar manner as the postfix operators though with the addition of the negation: - operator. Unlike the others, the negation: - operator is not applied in place and will always return a value. The operators may be used in a place where a value is expected to be returned from it, in which case the value of the variable post operator application will be returned:

```
avar = 10
anarray = [1 2 3 4 5]

++avar
--anarray[0]

newvar = ++avar
newarray = --anarray[0]

anothervar = 10
negative = -anothervar//negative == -10
```

11.1.6 Multiplicative

Concurnas supports the standard set of multiplicative operators which can act upon two, potentially differing, integral types, multiplication: *, division: /, power: ** and modulus: mod. For example:

```
mul = 3*2 //==6
div = 12/2 //==6
pow = 3**2 //==9
pow = 9**-2 //square root
modu = 10 mod 4 //==2
```

Attempting to divide by zero with two non floating point (either float or double) values will result in an exception of type java.lang.ArithmeticException being thrown. Attempting to divide by zero with at least one floating point value will result in: `Infinity` being returned. `0./0.` will resolve to `NaN`.

In situations where the left and right types of the multiplicative operator differ, the more general type shall be used for the return type. For example:

```
div1 = 13/2f //==6.5 (float)
div2 = 13/2. //==6.5 (double)
div3 = 13f/2. //==6.5 (double)
mulong = 100*10L //1000L (long)
mulong = 100L*10. //1000. (double)
```

Division and power gotacha

Often new and even experienced programmers will come across the following sort of problem:

```
assert 13/2 == 6.5 //fails!
```

The above fails because `13/2` for integers resolves to `6` and not `6.5`. Perhaps this is the answer desired, but to obtain the 'correct answer' of `6.5`, what we are looking for is a floating point calculation and this can be achieved by casting either of the arguments to the division operator to a floating point type, either float or double:

```
assert 13./2 == 6.5 //expected asnwer
```

11.1.7 Additive

Concurnas supports addition: + and subtraction -. These are straightforward and the rules concerning generalization of numerical types as par the multiplicative operators above apply. Examples:

```
addition = 1+1
subtraction = 1-1
```

Additionally, the addition operator + can be used for String concatenation. This is described in more detail in the The String concatenation operator + section.

11.1.8 Bitshift

Concurnas supports bit shift operators for integral types. They operate on a bit pattern, given by the left hand operand and a number of positions to shift by the right hand operand.

- << - Left shift. Shifts a bit pattern to the left.
- >>> - Unsigned right shift. Shifts a bit pattern to the right. Shifts a zero to the leftmost position.
- >> - Right shift. Shifts a bit pattern to the right. Leftmost bit is shifted contingent on sign extension.

Example:

```
def shifty(){
   a = 17
   x1 = a << 2
   x2 = a >> 2
   x3 = a >>> 2
   [x1, x2, x3]
}
```

```
shifty() //resolves to => [68, 4, 4]
```

11.1.9 Relational

The main relational operators in Concurnas are less than: <, greater than: >, less than or equal:<== and greater than or equal: >==. These operate upon two integral types. And return a `boolean` type. Examples:

```
lt = 1 < 2
gt = 3 > 2
lteq = 1 <== 2
gteq = 3 ==> 2
```

11.1.10 Instance of and Cast

The instance of or cast operators may not be overloaded. They are covered in more detail in theCasting and Checking Types chapter.

11.1.11 Equality

Concurnas supports two variants of equality, structural equality (as: equal: == and not equal: <>) and referential equality (denoted by prefixing the equality operator with an ampersand: `&`). The structural equality operators may operate upon any type, whereas the referential quality operators may only operate upon Objects. For all operator variants, they return a boolean value indicating equality. Let's look at some examples of structural equality:

```
eq = 10 == 10 //true
eq2 = 10 <> 12 //true
eq3 (1>3) == false //resolves to true
```

When it comes to structural equality for objects this is achieved by calling the equals method on the left hand side object and passing it the object on the right hand side as an argument. Note that in Concurnas this equality method is automatically created for all Objects and it resolves to provide structural equality, this behaviour can be user overridden to provide different behaviour. More details of this can be found in the Automatically generated equals and hashcode methods section. Examples:

```
class Person(name String)

d1 = Person('dave')
d2 = Person('dave')
f3 = Person('freddy')

assert d1 == d2 //struturally equal
assert d1 <> f3 //struturally not equal
assert d1 &== d1 //object is referentially equal to itself!
assert d1 &<> d2 //d1 and d2 though structurally equal (above) are not
    referentially equal
```

11.1.12 Bitwise operators

The major bitwise operators take two integral types as input. The following operators are supported:

- `band` - bitwise and

- bor - bitwse or
- bxor - bitwise exclusive or

These operators are commonly used in order to apply masks. Example:

```
def pprint(xx int) => '0x' + String.format("%8s",
    Integer.toBinaryString((xx+256) % 256)).replace(" ", "0")

value = 0b00010101
bitmask = 0b00000001

pprint(value band bitmask) // -> 0x00000001
pprint(value bor bitmask)// -> 0x00010101
pprint(value bxor bitmask)// -> 0x00010100
```

Complement

The comp operator can be used on an integral type in order to derive the complement of a single expression. Every 0 is flipped to a 1 and vice versa. Example:

```
orig byte = 0b00000011
complement= comp orig

Integer.toBinaryString((complement+256) % 256) // resolves to: '11111100' -
    which is the complement of the input byte.
```

11.1.13 Logical and/or

Concurnas supports the logical and and or operators which can be used in order to chain together boolean expressions. Naturally, both arguments to these operators must be of boolean type. More then one and or or may be chained together. The operators both return boolean values. Examples:

```
orand1 = 2 == 3 or 3>1
orand2 = 2 == 3 or 3>1 or 22>1 or 3 < 9
orand3 = 2>1 and 4>1
```

Short circuiting

Short circuiting is a logical optimization in which the second argument to an operator is executed or evaluated only if the first argument does not suffice to determine the value of the expression. This sometimes catches people out when they expect all arguments of an operator to be evaluated.

For a chain of or operators, evaluated left to right, as soon as one argument evaluates to true, the value true is returned (otherwise false), similarly for the and operator, evaluated left to right, as soon as an argument evaluates to false, false is returned (otherwise true).

```
def eval1(toRet boolean) => toRet
def eval2(toRet boolean) => toRet
def toCall(toRet boolean) => toRet

shortcir = eval1(false) or eval1(true) or toCall(true)//toCall will never be
    called!
shortcir = eval1(true) and eval1(false) and toCall(true)//toCall will never be
    called!
```

11.1.14 Ternary

The Ternary operator `x if test else z` can be thought of as a handy shorthand for the slightly more verbose: `if(test){ x }else{ z }` (in fact, under the hoot the Ternary operator is translated into this more verbose form). `x` and `z` must both return a value (of any type) and `test` must resolve to a value of `boolean` type. Example:

```
def test() => true

result = 12 if test() else 99
```

11.1.15 Invoke

Concurnas allows an `invoke` method to be defined for all objects, this permits the following syntax to be used in order to call this `invoke` method as an operator, with optional arguments:

```
class FormatPlus{
   def invoke(a int, b int) => "{a} + {b}"
}

fmt = FormatPlus()
fmt(12, 13)//returns: "12 + 13"
```

This can occasionally be useful tool to use when defining domain specific languages.

11.1.16 Null safety

The null safety operators are documented in detail in the Null Safety section.

11.2 Assignment

Variable assignment is covered in detail in the Variable assignment chapter. The assignment operator may be overloaded, this is described in the Overloading the assignment operator section, in the case of an overloaded assignment operator being defined for a type, the escaped assign of: \\= will suppress this behaviour when a value is assigned.

11.3 Compound assignment

Here we concern ourselves with compound assignment applied to a variable which has already been defined. These are: `+=`, `-=`, `*=`, `/=`, `**=`, `mod=`, `or=`, `and=`, `<<=`, `>>=`, `>>>=`, `band=`, `bor=`, `bxor=`. Taking the addition assignment operator, `+=` as an example, the following two statements are functionally identical:

```
a1 = a2 = 10

a1 += 1
a2 = a2 + 1

assert a1 == a2
```

That is to say, `a1 += 1` is essentially shorthand for: `a1 = a1 + 1`.

The compound assignment operators may be applied to array reference operations as follows:

```
a1 = [1 2 3]
```

```
a1[0] += 1

assert a1 == [2 2 3]
```

Again, `a1[0] += 1` is essentially shorthand for: `a1[0] = a1[0] + 1`.

11.3.1 Parentheses

In Concurnas, parentheses can be used or order disambiguate expressions composed of multiple operators. They may also be used in a "no operation" capacity in order to make code easier to read. For example:

```
orand4 = not (2>1 and 4>1)
```

11.4 Operator Overloading

Concurnas provides a mechanism by which the aforementioned operators can be supported for object types in addition to the primitives (and boxed versions thereof) above. This functionality can be extremely useful when implementing Domain Specific Languages. The following operators can be overloaded:

Class	Operator	Normal Token	Method name	Alt Name	Type overloaded	Notes
Assign	Assignment	=	assign	=	Left	Use v̄ariant of assign to avoid calling overloaded operator
	Unassign		unassign		Left	Where variable of type x implements unassign, use x: to avoid calling unassign operator
Compound Assign	Addition Assign	+=	plusAssign	+=	Left	
	Negation Assign	-=	minusAssign	-=	Left	
	Multiply Assign	*=	mulAssign	*=	Left	
	Divide Assign	/=	divAssign	/=	Left	
	Power Assign	**=	powAssign	**=	Left	
	mod Assign	mod=	modAssign	mod=	Left	
	or Assign	or=	orAssign	or=	Left	
	and Assign	and=	andAssign	and=	Left	
	left shift Assign	«=	leftShiftAssign	«=	Left	
	right shift Assign	»=	rightShiftAssign	»=	Left	
	unsigned right shift Assign	»>=	rightShiftU	»>=	Left	
	bitwise and Assign	band=	bandAssign	band=	Left	
	bitwise or Assign	bor=	borAssign	bor=	Left	
	bitwise xor Assign	bxor=	bxorAssign	bxor=	Left	
Not	Not	not	not		Unary	
Postfix	Increment	++	inc	++	Unary	
	Decremement	–	dec	–	Unary	
Prefix	Increment	++	inc	++	Unary	
	Decremement	–	dec	–	Unary	
	Positive	+	plus	+	Unary	No arguments specified
	Negative	-	neg	-	Unary	No arguments specified
	Complement	comp	comp		Unary	
Multiplicative	Multiplication	*	mul	*	Left	
	Power	**	pow	**	Left	
	Division	/	div	/	Left	
	Modulus	mod	mod		Left	
Additive	Addition	+	plus	+	Left	
	Subtraction	-	minus	-	Left	
Bitshift	Left shift	«	leftShift	«	Left	
	Right Shift	»	rightShift	»	Left	
	Unsigned right shift	»>	rightShiftU	»>	Left	
Relational	Less than	<	compareTo		Left	a.compareTo(b) < 0
	Greater than	>	compareTo		Left	a.compareTo(b) > 0
	Less than or equal	<==	compareTo		Left	a.compareTo(b) <== 0
	Greater than or equal	>==	compareTo		Left	a.compareTo(b) >== 0
Equality	Structurally equal	==	equals		Left	a.equals(b)
	Structurally not equal	<>	equals		Left	not a.equals(b)
Bitwise	Bitwise and	band	band		Left	
	Bitwise or	bor	or		Left	
	Bitwise exclusive or	bxor	bxor		Left	
Logical and	and	and	and		Left	
Logical or	or	or	or		Left	
Invoke	Invoke	a()	invoke		Left	Any number of arguments may be specified
Lists, Arrays, Maps	Get	a[y]	get		Left	
	Put	a[y] = z	put		Left	
	Sublist	a[y1 ... y2]	sub		Left	a.sub(y1, y2)
	Sublist from	a[y1 ...]	subfrom		Left	a.subfrom(y1)
	Sublist to	a[... y2]	subto		Left	a.subto(y2)
	Put sublist	a[y1 ... y2] = z	subAssign		Left	a.subAssign(y1, y2, z)
	Put sublist from	a[y1 ...] = z	subfromAssign		Left	a.subfromAssign(y1, z)
	Put sublist to	a[... y2] = z	subtoAssign		Left	a.subtoAssign(y2, z)

Class	Operator	Normal Token	Method name	Alt Name	Type overloaded	Notes
	In	in	contains		Right	b.contains(a)
	Not in	not in	contains		Right	not b.contains(a)

11.4.1 Implementing and using overloaded operators

Overloaded operators operate upon the left hand element of an expression unless it is unary where there is only one expression the operator is being applied to, or for the `int` and `not in` operators where the right hand expression must have the relevant method implemented. The left hand element must support the operator being invoked. For non alphanumerical named operators (e.g. `+`, `**` etc) the raw alphanumerical operator token may be defined as a method name, or it's alternative longhand name may be used (`plus` for `+`, `mul` for `*` etc).

```
class Complex(real double, imag double){
   def +(other Complex) => new Complex(this.real + other.real, this.imag +
       other.imag)
   def +(other double) => new Complex(this.real + other, this.imag)
   def +=(other Complex) => this.real += other.real; this.imag += other.imag
   def +=(other double) => this.real += other
   override toString() => "Complex({real}, {imag})"
}

c1 = Complex(2, 3)
c2 = c1@
c3 = c1@
c4 = Complex(3, 4)

result1 = c1 + c4
result2 = c1 + 10.
c2 += c4 //compound plus assignment
c3 += 10.//compound plus assignment

//result1 == Complex(5.0, 7.0)
//result2 == Complex(12.0, 3.0)
//c2 == Complex(5.0, 7.0)
//c3 == Complex(12.0, 3.0)
```

Above we see the plus and compound plus assignment operator have been overloaded for the object class: Complex. Note that operator overloading methods themselves may be overloaded with differing argument types(just like regular methods).

11.4.2 Operator overloading via extension functions

It's possible to implement operator overloading via use of extension functions. this is documented in the Extension functions as operator overloaders section.

11.4.3 Overloading the assignment operator

The assignment operator, = can be overloaded. Example:

```
class AssignOPOverload(value int){
   def =(a int){ value = a; }

   override toString() => "AssignOPOverload: {value}"
}

obj= new AssignOPOverload(100)
```

```
obj= 66
//thing == AssignOPOverload: 66
```

If the overloaded assign method returns a value then this will be ignored. If you wish to suppress invoking the assign operator then using the escaped assign of: \\= will suppress this behaviour, and will cause the assignment to behave a normal, assigning whatever is on the right hand side to the variable expressed on the left hand side:

```
class AssignOPOverload(value int){
   def =(a AssignOPOverload){ value = a.value + 10000; }

   override toString() => "AssignOPOverload: {value}"
}

inst1 = new AssignOPOverload(100)
inst2 = new AssignOPOverload(100)

inst1 = new AssignOPOverload(22)
inst2 \= new AssignOPOverload(22) //bypass overloaded = operator
/
inst1 => AssignOPOverload: 10022
inst2 => AssignOPOverload: 22
```

Overloading of the assignment operator is used to great effect with off heap memory and gpu memory interaction.

11.4.4 Unassign

Just as we can overload the assignment operator, so too can be implicitly overload the 'unassignment operator', consisting of a zero argument method named `unassign` returning any non void type. This is best illustrated with an example:

```
class MyUnassignable{
   myvar int
   def assign(anint int) => myvar = anint;;
   def unassign() => myvar
}

inst = MyUnassignable()
inst  = 56
res = inst
/
res => 56
```

If we wish to suppress the unassignment operation this can be achieved by using the : operator as follows:

```
class MyUnassignable{
   myvar int
   def assign(anint int) => myvar = anint;;
   def unassign() => myvar
   override toString() => 'MyUnassignable {myvar}'
}

inst = MyUnassignable()
inst  = 56
```

```
res1 = ''+ inst
res2 = '' + inst: //suppress calling of unassign()
res3 = inst:tostring() //suppress calling of unassign()
/
res1 => '56'
res2 => 'MyUnassignable 56'
res3 => 'MyUnassignable 56'
```

If we wish to create a ref to a variable of type implementing the `unassign()` method, and we wish to avoid 'unassigning' it then we must use an extra `:`.

```
class MyUnassignable{
   myvar int
   def assign(anint int) => myvar = anint;;
   def unassign() => myvar
   override toString() => 'MyUnassignable {myvar}'
}

inst = MyUnassignable()
inst  = 56

ref MyUnassignable: = inst:: //ordinarily we'd just use 'inst:' to create a ref
```

Overloading of unassignment is used to great effect with lazy variables.

11.4.5 Assignment operators

If there is an object returned from an overloaded assignment operator which happens to be a ref type, then Concurnas will await for the assignment on the ref type to take place before continuing with execution. similarly, if the operator overload method being invoked is tagged with the `@com.concurnas.lang.DeleteOnUnusedReturn` annotation, then delete will be called on the object returned before continuing with execution.

This applies to the following assignment operators:

- **The assignment operator:** `=`
- **The in place assignment operators:** `+=, -=, *=, /=, **=, mod=, band=, bor=, bxor=, comp=, <<=, >>=, >>>=`
- **The sublist assignment operators:** `[a ... b ] = z, [a ... ] = z, [ ... b ] = z`

The escaped assignment operator: \= is excluded from this behaviour.

12. Control Statements

Crucial to imperative programming is the concept of control flow within a function or method. To this end a number of branching statements are provided within Concurnas. These are: `if elif else`, the `if else` expression, `while`, `loop`, `for`, `parfor`, `match` and `with`. First, lets examine the blocks which these control statements are composed of.

12.1 Blocks

Blocks form the foundation of control flow statements. They serve two purposes:

1. Defining scope - Variables and functions declared within a block cannot be used outside of that block. But code within a scope can use the functions and variables defined in parent nestor scopes.
2. Returning of values.

Simply put a block is a pair of curly braces:

```
{
   //more code in here
}
```

Concurnas also offers a compact, single line block format => which can be used as follows:

```
def myMath(a int, b int, c int) => a*b+c

def mypy(a int, b int) => res = a**2 + b**2; return res**.5
```

Blocks are able (but not obliged) to return values if their final logical line of code returns something:

```
result = {
   a = 2+3;
   a**2
}//result is assigned value 25
```

Blocks which are used as part of control statements (soon to be elaborated) may return values as well, this becomes a very useful programming construct for creating concise units of code. For example:

```
result = if(cond1){
   "condition 1 met"
}elif(cond2){
   "condition 2 met"
}else{
   "no conditions met"
}
```

In the case where there are two or more different types returned by the individual blocks associated with a control statement, the most specific type which can satisfy all the available returned types is chosen as the overall return type of the statement:

```
something Number = if(xyz()){
   82//this is boxed to an Integer, which is a subtype of the more general type
       Number
}else{
   new Float(0.2)//Float is a subtype of Number
}
```

All branches must return something for the above sort of code to be valid, if at least one branch does not return something, then this approach cannot be used (unless another flow of control statement causing breakout is encountered such as `return`, `throw` etc as the final line of the offending branch).

Since blocks return values, and lambdas/functions/methods are composed of a name (except for lambdas), signature and block, then we can skip having to use the `return` statement in our function/method definitions.

So instead of:

```
def plusTwo(an int){
   return an + 2
}
```

We can write:

```
def plusTwo(an int){
   an + 2
}
```

In fact, with Concurnas, we can take this a step further by using the compact block form of =>:

```
def plusTwo(an int) => an + 2
```

Note that the return type in the `plusTwo` function definitions is inferred as `int`. Sometimes however this auto-return behaviour is undesirable. Say we are trying to implement a function returning void and our last expression can return something, here we can use nop `;;` to suppress this being returned:

```
count = 0
def incrementor() => count++;;
```

Alternatively, we can specify the type for our function:

```
count = 0
def incrementor() void => count++
```

12.2 Anonymous Blocks

It's often useful to specify a block on its own, without an attached control statement or association to a lambda/function/method etc. This provides one a bounded scope which is handy if one is working on a large/complex section of code and wishes to make clear a certain part of code is 'separate' functionality wise from the rest, and enable variable names to be potentially reused. It also allows one to return a value. For example:

```
//complex code here

oddcalc = {
   a = 23
   a*2 + a
}

anotherone = {
   a = 57
   b = 99
   a*b-1
}
```

In fact, this turns out to be especially useful functionality in so far as defining isolates is concerned. For it enables us to write this kind of code:

```
res = {
   //complex code here
   a= 99
   a + 1
}!//Use the ! operator to create an isolate
```

12.3 If elif else

If, else, if else is a branching control statement. It is composed of a if test and block, optionally any number of elif (or else if) test and blocks and may optionally include an else block for when all the if test and any defined elif tests resolve to false. For example:

```
def fullif(an int) String {
   if(an == 1){
      "one"
   } elif (an==2){
      "two"
   } else if(an ==3){//'elif' and 'else if' are considered to be syntactically
       identical
      "three"
   } else{
      "other"
   }
}
```

```
def ifelse(an int) String {
   if(an == 1){
      "one"
   } else{
      "other"
   }
}

def ifelif(an int) String {
   res = "unknown"
   if(an == 1){
      res = "one"
   } elif (an==2){
      res = "two"
   }
}

def justif(an int) String {
   res = "unknown"
   if(an == 1){
      res = "one"
   }
}
```

In the final two examples above `ifelif` and `justif`, since no else block is provided, there is no certainty that all paths will resolve to return a value (it's possible that all the if and elif tests will fail) - so we cannot return a value from the if statements.

12.4 If else expression

The if else expression, whilst not a statement, is translated into a if elif else statement (without any elif units) and behaves otherwise identically to an if-elif-else statement. An if test and else case must be included. For example:

```
avar = 12 if xyz() else 13
```

Functionally, this is identical to writing:

```
avar = if(xyz()){ 12 } else{ 13 }
```

12.5 Single expression test

In Concurnas, a expression may be referenced in isolation within a test requiring a boolean value, e.g. an `if`, `elif` or `while` test expression:

```
def gcd(x int, y int){
   while(y){//translated to y <> 0
      (x, y) = (y, x mod y)
   }
   x
}
```

In cases such as these where the value resulting from evaluating the test expression is non `boolean` and in fact integral in nature, this is compared against the value `0`, if the result is non zero the expression resolves to true.

12.6 toBoolean

As a corollary and in aid of our Single expression test above, all objects in Concurnas have a `toBoolean` method automatically defined. This returns a single `boolean` value resolving to `true`. This value method may be overridden, for instance if one were defining a data structure, one may wish to define the `toBoolean` as returning `false` if the structure were empty.

The `toBoolean` method is called when a single object is referenced where a boolean value is expected. Additionally, if the object is nullable, then it is checked for nullability, only if these two conditions are met is a value of `true` returned. For example:

```
class MyCounter(cnt = 0){
   def inc(){
      cnt++
   }
   override toBoolean(){
      cnt > 0
   }
}

counter = MyCounter()

result = if(counter){//equivilent to: counter <> null and counter.toBoolean()
   'counter is greater than one'
}else{
   'counter is zero'
}
```

The behaviour for Strings, arrays, lists, sets and maps in Concurnas is slightly different from the above. In order to return true for:

- Strings. The value must be non-null and non zero (i.e. not an empty String)
- Arrays. The value must be non-null and the `length` field must be greater than 0.
- Lists, sets and maps. For `java.lang.List`, `java.lang.Set`, `java.lang.Map`'s the value must be non-null and a call to the `isEmpty` method must return `false`.

12.7 Loop

The `loop` statement is our first repeating control statement. It will execute the block of code attached to it repeatedly until either the code breaks out from the loop (by using the `return`, `break` or by throwing an exception) or the program is terminated - which is the less common use case. For example:

```
count=0
loop{
   System out println ++count
   if(count == 100){
      break
   }
}
```

12.8 While

The `while` statement is our second repeating control statement. It behaves in a similar way to loop but has a test at the start of the loop which if passed results in the attached block being called, at the end of the block the test is attempted again and if it passes then the block is called again, etc. If the test ever fails then repetition ceases. For example:

```
count = 0
while(++count <= 100){//test
   System out println count
}//block to execute
```

12.9 For

The `for` loop statement is our third and final repeating control statement. There are two variants of for loop: c style for and iterator style for.

12.9.1 C style for

C style for is the syntactically older of the two variants of for loop. In addition to its main block to repeat, it is composed of three key components:

1. An initialization expression, this is executed once at the start of the for loop. If a variable is created here it is bounded to the scope of the main block.
2. A termination expression. For as long as this resolves to `true` the loop is repeated, it is executed at the start of each potential repetition of the loop.
3. A post expression. This is executed at the end of the loop.

Here is a typical instance of a c style for loop:

```
for(an = 0; an < 10; an++){
   System out println an
}
```

Note that it's perfectly acceptable to omit any or all of the key components above, the following is equivalent to our loop example explored previously:

```
count=0
for(;;){
   System out println ++count
   if(count == 100){
      break
   }
}
```

12.9.2 Iterator style for

The iterator style for is great for performing an operation per instance of an item in a list, array or otherwise iterable object - which is often the case about 80% of the time in modern programming with for loops. Instead of the three separate key components as par the c style for loop above, we just have one, an iterator expression using the `in` keyword, which creates a new variable with scope bound to the main block of the for loop:

```
for(x in [1 2 3 4 5 6 7 8 9 10]){
   System out println x
}

for(x int in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]){//here we choose to specify the
    type of our new variable x
   System out println x
}
```

In the above instances we are choosing to iterate over an n dimensional array and a list respectfully. We can also iterate over any object that implements the `java.util.Iterator` interface, for example a range object:

```
for(x in 1 to 10){
   System out println x
}
```

12.10 Parallel for

`parfor` looks a lot like a repeating control statement and indeed it does largely behave like a for loop in so far as iteration is concerned - indeed, both styles of for iteration are permitted, c style and iterator style. But execution of the main block is quite different. `parfor` will spawn the maximum number of isolates it sensibly can (i.e. no more than one per processor core), and will assign each of the iterations of the for loop to these isolates in a round robin fashion, equally distributing the workload. Let's look at an example:

```
result1 = parfor(x=1; x <= 100; x++){
   1 + x*10
}

result2 = parfor(x in 1 to 100){
   1 + x*10
}
```

Bear in mind that code expressed in the main block that attempts to interact with the iteration operation (i.e. the value of `x` in the first `parfor` statement above) will not behave in the same way as an ordinary for loop. Likewise, break and continue are not permitted within a `parfor` loop because the execution of the main block across a number of differing isolates occurs on a concurrent and non deterministic basis.

Instead of returning a `java.util.List<X>` object, as par a regular for loop, an array of ref's corresponding to each 'iteration' of the main block is returned of type: `X:[]` - where `X` is the type returned from the main block.

12.10.1 Parforsync

`parforsync` behaves in the same way as `parfor`, but all the ref's returned from the 'iterations' of the main block must have a value set on them before execution may continue past the statement - i.e. all iterations must have completed. Example:

```
result1 = parforsync(x=1; x <= 100; x++){
   1 + x*10
}

result2 = parforsync(x in 1 to 100){
   1 + x*10
}
```

12.11 Repeating Control Statements Extra Features

The Repeating control statements; `loop`, `while`, `for`, `parfor` and `parforsync` share some common extra features which make them especially useful.

12.11.1 else block

An `else` block can be attached to all the repeating control statements, except `loop`, `parfor` and `parforsync`, and allows one to cater for cases where the attached main block is never entered into (i.e. the relevant test resolves to false on the first attempt):

```
res = ""
while(xyz()){
   res += "x"
}else{
   res = "fail"
}
```

```
res = ""
for(x in xyz()){
   res += x
}else{
   res = "fail"
}
```

12.11.2 Returns from repeating control statements

The repeating control statements may return a list of type `java.util.List<X>` where `X` is the type returned by the attached loop block - `int` in the below example:

```
count=0
series = loop{
   if(count == 100){
      break
   }
   ++count//returns a value of type int
}
```

For `series` above the type is `java.util.List<int>`
when used in a for loop:

```
series = for(a in 1 to 100){ a }
```

Note that `parfor` and `parforsync` have their own, different, return logic covered previously.

12.11.3 The index variable

All the repeating control statements (except the c style for loop) may have an attached index variable. This is handy in cases where one say wishes to use the Iterator style for loop, but also have a counter for the number of iterations so far. The index may have any name and by default the type of the index is `int`. `long` is a popular choice if you need to explicitly define the type of the index where working with very large iterations. An initial value assignment expression may be defined, if one is not then the value of the index is set to `0`. For example, with a for loop:

```
items = [2 3 4 5 2 1 3 4 2 2 1]

res1 = for(x in items; idx) { "{x, idx}" }//idx implicitly set to 0
res2 = for(x in items; idx long) { "{x, idx}" }//idx implicitly set to 0L
res3 = for(x in items; idx = 100) { "{x, idx}" }
res4 = for(x in items; idx long = 100) { "{x, idx}" }

alreadyIdx = 10//index tobe used defined outside of loop
res5 = for(x in items; alreadyIdx) { "{x, alreadyIdx}" }//alreadyIdx already
    defined
```

Index variables may be applied to loops and while loops in a similar fashion to iterator style for loops:

```
items = [2 3 4 5 2 1 3 4 2 2 1]
n=0; res1 = while(n++ < 10; idx) { "{n, idx}" }//idx implicitly set to 0
n=0; res2 = loop(idx) {if(n++ > 10){break} "{n, idx}" }//idx implicitly set to 0
```

12.12 With statement

The with statement which, though not being an actual control statement, does look a lot like one, and for that reason is included here. This is particularly useful in cases where one must call many methods, or interact with many fields of an object in a block of code. Using the with statement allows us to skip having to explicitly reference the object variable of interest:

```
class Details(~age int, ~name String){
   def getSummary() => "{name}: {age}"
   def rollage() => age++;;
}

det = new Details(23, "dave")

with(det){
   name = "david"
   rollage()
}

det.getSummary()
/
=> "david: 24"
```

With statements, as with all block based code in Concurnas can return values:

```
class Details(~age int, ~name String){
   def getSummary() => "{name}: {age}"
   def rollage() => age++;;
}

det = new Details(23, "dave")

summary = with(det){
   name = "david"
   rollage()
   getSummary()
}
//summary == "david: 24"
```

With statement may be nested. In instances where there is an identically callable method in the nesting layers, then the innermost layer is prioritized.

12.13 Break and Continue

The `break` and `continue` keywords may be used within all the repeating control statements: loop, while, for (but not `parfor` or `parforsync`) in order to affect the flow of control within the main block of the statements.

The `break` keyword allows us to break out (escape from) from the inner most repeating control statement main block, and continue on with execution of the code post the control statement.

The `continue` keyword allows us to terminate the current repetition of the inner most repeating statement main block and carry on execution as if we had reached the end of the main block attached to the statement normally - so for a `loop` this would be the start of the main block, for a `while` it would be the beginning of the test etc.

Let's look at an example of `continue` and `break`:

```
items = x for x in 1 to 50

for(x in items){
   if(x <== 5){
      continue //go back to the start of the loop, i.e. skip the code which
          outputs to console
   }elif(x == 9){
      break//go to the code after the for loop
   }
   System out println x
}
```

The `continue` and `break` keywords may be used when the repeating control structures which return a value for each iteration in the following manner:

```
series = for(x in items){
   if(x <== 5){
      continue x//continue but add a value to the result list
   }elif(x == 6){
      continue//continue and don't add a value to the result list
   }elif(x == 9){
      break x//break but add a final value to the result list
   }elif(x == 10){
      break//break and don't bother to return a value to the result list
   }
   x//if we've made it throught the above, then this value is added to the
       result list
}
```

13. List Comprehensions

List comprehensions provide a convenient and concise mechanism by which lists can be created and operated upon. Typical use cases include iteration over elements of an existing iterable structure (just like a for list) with an operation applied to each element, and potentially with a filter condition resulting in a subsection of the input list being evaluated. The parallel for and synchronised parallel for variants are available in addition to conventional for. The syntax is as follows:

```
[expression or block returning a value] (for|parfor|parforsync [variable name]
              ([type])? in expression)+ (if [boolean expression])?
```

The expression on the right hand side of the in token above must return either an n-dimensional array or an object implementing the java.util.Iterator interface. The list comprehension expression will return a type contingent upon the variant of for used:

	for	parfor	parforsync
List or iterable (of type X)	`List (List<X>)`	`RefArray (X:RefArray)`	`Array (X[])`

Let's say we wish to produce a list for which the values of all elements have 100 added to them:

```
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mynewlist = a+100 for a in mylist
//mynewlist == [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
```

A block of code may be included in place of a single expression as above:

```
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mynewlist = {x=a+100; x} for a in mylist
//mynewlist == [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
```

Now let's say we wish to only include those items divisible by two. This is a filter condition. We can add this filter condition as follows:

```
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mynewlist = a+100 for a in mylist if a mod 2 == 0
//mynewlist == [102, 104, 106, 108, 110]
```

List comprehensions can operate on any expressions which a for loop can - i.e. any object implementing the java.lang.Iterable interface (including maps) and arrays:

```
myarray = [1 2 3 4 5 6 7 8 9 10]
mynewlist = a+100 for a in myarray if a mod 2 == 0 //note, a list is always
    returned
//mynewlist == [102, 104, 106, 108, 110]
```

13.1 Nested List Comprehensions

We can easily nest list comprehensions consisting of more than one for clause (though this comes at the cost of code readability so is generally not recommended):

```
res = [x,y] for x in [1,2,3] for y in [3,1,4] if x <> y
//res == [[[1, 3], [1, 4]], [[2, 3], [2, 1], [2, 4]], [[3, 1], [3, 4]]]
```

Behind the scenes: Concurnas implements for comprehensions by translating the defined expressions into conventional `for` loop compound statements. In the above example this is translated into:

```
res = for( x in [1,2,3] ) {
   for( y in [3,1,4] ) {
      if( x <> y ) {
         [x, y]
      }
   }
}
//res == [[[1, 3], [1, 4]], [[2, 3], [2, 1], [2, 4]], [[3, 1], [3, 4]]]
```

And now we're stuck on rewind
Let's follow the cops back home
Follow the cops back home
Lets follow the cops back home
And rob their houses

- Follow the Cops Back Home from Meds by Placebo (2006).

14. Exceptions

Exceptions allow us to interrupt the ordinary flow of control of our application, and fail up the call stack until we arrive at code designed to catch said exception and perform some action in response to it.

All exceptions in Concurnas are considered Runtime exceptions. i.e. it's not required that one have to explicitly decorate one's method or function, which has the capability to throw an exception, with details of that exception. The rational being that exceptions are meant to be 'exceptional' - i.e. not an ordinary part of structured programming. If one is relying upon them in order to pass the flow of control around a program, then one may wish to reconsider the design of one's software. In fact, many successful programming languages don't have the concept of Exceptions at all, but rather, error state is managed via return values. With Concurnas we feel that we've achieved the best of both worlds; exceptions are there if you need them, but they don't get in the way.

14.1 Throwing Exceptions

Exceptions are objects which are subtypes of `java.lang.Throwable`, only these objects may be thrown. Objects which are not subtypes of `Throwable` cannot be thrown.

Exceptions are objects, are created as objects, and can be thrown via use of the /throw/ keyword:

```
class MyException(msg String) < Exception(msg)

def carefulOperation(a double){
   if(a <== 0.){
      throw new MyException("Input: {a} to 'carefulOperation' must be greater
          than zero")
   }
   100*a
}
```

It's recommended that one specialize one's range of exceptions thrown, by defining an appropriate exception hierarchy, as par above. This is a better approach than using a catch all exception and throwing this every time one encounters an exceptional situation, or just throwing `java.lang.Exception` (with or without an appropriate message) - because it makes the job of catching exceptions, and differentiating the reaction to those exceptions easier as we shall see next...

14.2 Catching Exceptions

We can catch exceptions thrown within the bounds of a try block by using series of one or more catch blocks. A catch block specifies the type of exception it will catch and a variable to assign the exception to which is bound to the scope of the attached catch block. An exception is handled by a catch block if it's type is equal to or a subtype of the one specified. If no type is specified then the type `Throwable` (i.e. to catch any exception) will be used. From our previous example:

```
try{
   carefulOperation(-1.)
   //some more code here...
}catch(e MyException){
   System.out.println("operation failed: " + e.getMessage())
}catch(e Exception){
   //special logic here to handle this
}catch(e){//catch everything else
   System.out.println("operation failed for unexpected reason: " +
       e.getMessage())
}
```

Note that an exception can only be caught and handled by at most one catch block (or non at all if there isn't one for the type of the exception thrown).

If there is no catch block defined for the exception type then the exception will be thrown by the method/function containing the try block - and passed up the call stack for catching by an appropriate catch block (if defined).

We must order our exceptions so as to avoid a superclass of an exception specified in a later catch block being defined before it - as this would mask the latter defined exception. But fear not, the compiler is on our side, because if we do this then it will result in a compilation error, thus bringing the problem to our attention.

14.3 Returning values

Try catch blocks, like control statements, may return values. For example:

```
def runAndtry(an double) double {
   result double = try{
      carefulOperation(an)
   }catch(e MyException){
      System.out.println("operation failed: " + e.getMessage())
      -1.
   }
   return result
}
```

The above form can only be used if the try block and all the specified catch blocks return something. In the case where the try block and catch blocks return values of differing type, the most specific common superclass amongst the specified types will be returned.

14.4 Multi-catch

Often one needs to catch multiple exceptions, but does not want to catch the superclass of those exceptions. Normally one would need to define a catch block per exception type and duplicate the code to handle the exception multiple times for each type within each of the corresponding catch blocks. But we can instead use a multi-catch block as par below:

```
open class MasterException(msg String) < Exception(msg)

class Excep1(a String) < MasterException(a)
class Excep2(a String) < MasterException(a)

def thrower(an int) void{
   match(an){
      1 => throw new Excep1("value was 1")
      2 => throw new Excep2("value was 2")
      3 => throw new Exception("another exception")
   }
}

def multicatch(an int) String{
   try{
      thrower(an)
      "execution ok"
   }
   catch(a Excep1 or Excep2){//here is our multicatch
      "Exception thrown: " + a.getMessage()
   }
   catch(e){//catch everything else
      "mystery exception: " + e.getMessage()
   }
}
```

14.5 Finally blocks

In the situation where there is no appropriate handler for an exception thrown within a try block, the exception will be passed up the callstack. Often we need to ensure that a piece of code is run after a try block has regardless of whether an exception has been thrown/catched (for instance, to clean up/close resources used prior to and within the try block). To this end we can use a finally block to guarantee the code is run regardless of whether our try block has run normally or thrown any exceptions (which have been caught or not caught and rethrown):

```
class Resource(){
   def open(){
      //code to open resource
   }

   def close(){
      //clean up code
   }

   def use(){
      //use resource in some way
   }
}

class AnException < Exception("An exception thrown")

def myFinallyMethod(){
   r = Resource()
   try{
      r.open()
      //code here using the resource which may throw an exception
   }catch(e AnException){
      //handle exception
   }finally{//code is always run regardless of whether exception is thrown, not
       caught and rethrown, caught or not thrown at all
      r.close()
   }
}
```

In fact, we may omit the use of any catch blocks and just have a `try finally` pair:

```
def myFinallyMethod(){
   r = Resource()
   try{
      r.open()
      //code here using the resource which may throw an exception
   }finally{//code is always run regardless of whether exception is thrown, not
       caught and rethrown, caught or not thrown at all
      r.close()
   }
}
```

Note that specifically for the case above of resource management it can often be beneficial to use a try with resources block form, discussed later.

Finally blocks may not return values, either explicitly or implicitly. If Concurnas where to permit this then it would make reasoning about what value is returned from a try catch block with finally difficult for the author of the code and anyone who reviews it.

14.6 Dealing with finally blocks that throw Exceptions...

If the code within a finally block can throw an exception, then the exception thrown by the code within the main try block (if any) will be suppressed by this new exception originating from the finally block. In instances such as this, the suppressed exception can be access by calling the `getSuppressed()` method on the exception thrown.

Through generally speaking it's not a good idea to allow code within a finally block to throw an exception, for the above reason of suppression and also for the sake of reasoning about said exception further up the callstack.

14.7 The Default Isolate Exception Handler

All code in Concurnas runs within isolates. If an exception boils up the callstack, uncaught and reaches the top level of an isolate, then the default isolate exception handler will be invoked. The default handler simply outputs a description of the exception to the console.

Generally speaking, if exceptions are reaching the top level, uncaught then there is probably something very wrong about the design of the software being executed, or the exceptions are not interesting and ignorable - which is strange in and of itself since they really ought not to be thrown in the first place if this is the case.

14.8 Try With Resources

Often when we are working with managed resources (files, database connections , hardware interfaces etc), we must remember to explicitly close the resource after we have finished using it, so as to avoid a resource leak. To aid this process Concurnas implements try-with-resources. For example:

```
def readFirst(path String) {
   fr = new java.io.FileReader(path)
   br = new java.io.BufferedReader(fr)
   try (br) {
      br.readLine()
   }
}
```

In the above instance we do not need to remember to explicitly call the close method on the br object as the close method on the `BufferedReader` object will be called at the end of the try block of code. In fact, even if our try block throw an exception, the close method will still be called.

We can assign the object to be closed to a variable for closing within our try with resources expression, and make use of that variable within the try block:

```
def readFirst(path String) {
   fr = new java.io.FileReader(path)
   try(br = new java.io.BufferedReader(fr)) {
      br.readLine()
   }
}
```

This is a nice design pattern since we avoid the risk of accidentally using the br variable outside the try block once it has been closed.

We can assign multiple variables as part of the try with resources expression by delimiting them with ;:

```
def readFirstTwo(path1 String, path2 String) {
   try(br1 = new java.io.BufferedReader(new java.io.FileReader(path1)); br2 =
       new java.io.BufferedReader(new java.io.FileReader(path2))) {
      br1.readLine() + br2.readLine()
   }
}
```

In order to make a class compatible with try with resources one must implement a zero argument close method returning void like the following:

```
def close() void { /* closing code */ }
```

15. Arrays Matrices and Lists

There are said to be two schools of thought with regards to linear or serial collections of data. 1). Arrays, in which data is stored in memory in a linear (and often continuous) manner where post creation the size of the structure cannot be changed and 2). lists, where though often not as performant as arrays, permit more flexibility in changing their size and are generally easier to work with. For historical as well as practical reasons, arrays tend to be more popular in the scientific and numerical computing communities and lists more popular with enterprise engineering.

With Concurnas, we provide native support for both structures as well as a simple syntax which makes them equally quick and easy to work with.

15.1 Array and Matrix Types

In Concurnas, one can define a one dimensional of any type. For example an array of integers is defined as `int[]`. A `n > 1` dimensional array can be defined by specifying the dimensionality between the brackets, e.g. a two dimensional String array: `String[2]`. Alternatively a multidimensional array may be represented by using empty brackets, e.g. a two dimensional String array: `String[][]` - but of course, for declarations involving many dimensions, it can be more convenient to use the numerical representation instead of the brackets. Example:

```
res1 int[][][]
res2 int[3]
```

`res1` and `res2` above are of the same equivalent type a 3 dimensional int array.

A two dimensional array is special in that it's also known as a matrix (structurally, in memory it is an array of arrays). Higher orders of n than 2 are possible and as with single dimensional arrays, these are multidimensional arrays and they are definable for any type.

15.2 Array and Matrix Creation with default Initialization

In Concurnas n-dimensional arrays are objects. As such they can be created using the new keyword as follows:

```
myArray int[] = new int[10] //single dimensional array with 10 elements
```

And, like all other objects, the new keyword may be omitted:

```
myArray int[] = int[10] //single dimensional array with 10 elements
```

If we wish to define a multidimensional array we must specify the length of the dimensions. These are delimited using a comma:

```
myMatrix1 int[2] = new int[2, 10]//2d matrix 2 rows of 10 columns each
myMatrix2 int[2] = new int[2][10]//2d matrix 2 rows of 10 columns each
```

The default value for each of the elements in the array, when instantiated in this manner, are is as follows:

Type	Default Array Value
Primitive numerical: `short`, `int`, `long`, `float`, `double`	0
Primitive `char`	0
Primitive `boolean`	false
Primitive `byte`	0
Boxed Primitive Type	null
Non primitive types (e.g. Objects)	null

Note that the final dimension does not need to be qualified, in this case the final dimension(s) of array omitted will resolve to null:

```
myMatrix1 int[2] = new int[3][]
myMatrix2 int[2] = new int[3,]

//both myMatrix1 and myMatrix2 == null null null
```

15.3 List creation with default Initialization

In order to maintain compatibility with other JVM languages, lists in Concurnas are provided via use of the `java.lang.List implementations`. For example, if one wishes to create an empty array list one can do so as par a normal object:

```
mylist = new ArrayList<int>();//create and empty arraylist holding Integers
```

15.4 Array and List creation with initialization

Arrays and lists can be initialized with non default values using very similar convenient syntax. The elements of a list must be delimited using a comma, whereas we use a space for arrays.

```
myArray = [1 2 3]
myList = [1, 2, 3]
```

Arrays can be created using comma delimiters, this is enabled by prefixing the initial bracket with an `a`. This is useful when the contents of the array contain many expression lists or invocations to extension functions and otherwise using brackets to bind these would appear inelegant:

```
mySimpleArray = a[1, 2, 3]

def int powAndPlus(pow int, plus int) => (this ** pow) + plus

anotherArray = a[10 powAndPlus 2 4, 2 powAndPlus 3 6]
```

If possible, Concurnas will automatically cast all of the subelements of an array or list to be the same component type when stored as appropriate. In order to do so it will find the most generic type which is equal to or is the supertype of all the elements. In the degenerate case this of course would be Object. For example, here all of the elements are converted to double:

```
mydoubleArray = [1. 2 3]//mydoubleArray resolves to [1.0 2.0 3.0]
```

In the case where one or more elements is a boxed variant of a primitive type, all elements will be boxed to the relevant Object type:

```
res = [1 2 Integer(3)] //this array is of type Integer[]
```

15.5 Array creation with element wise initialization

Often it can be useful to create array or lists with each element initialized to the same logical value. In Concurnas this is supported for both primitive and object type n dimensional arrays as follows:

```
Class MyClass(a int, b int){ override toString() => 'mc {a:b}' }

myArrayPrim = new int[2,3](99)
myArrayObj = new MyClass[2,3](new MyClass(99, 1))


//myArrayPrim => [[99, 99, 99], [99, 99, 99]]
//myArrayObj => [[mc 99:1, mc 99:1, mc 99:1], [mc 99:1, mc 99:1, mc 99:1]]
```

In the above case, for the object type array, `myArrayObj`, the value resulting from the call of the `MyClass` constructor call is copied for each element. Each element of the matrix is its own object, copies are not shared between array elements.

15.6 Matrix creation with initialization

Given the importance of matrix operations in numerical computing, Concurnas has special support for matrices. One can create a matrix with initial value as follows:

```
A = [ 1 2 ; 3 4]

//A resolves to:
//1  2
//3  4
```

If one prefers to use commas as an array delimiter then the above may be achieved as follows:

```
A = a[ a[1, 2] , a[3, 4]]

//A resolves to:
//1  2
```

```
//3  4
```

15.7 Array concatenation and appending

In Concurnas, the semicolon operator `;` within the context of a n dimensional array element definition inside a pair of square brackets `[ ]` has special significance as a concatenation operator. To add a row to a matrix we do the following

```
m = [ 1 2; 3 4 ]
added = [m ; 5 6]

//added resolves to:
//1 2
//3 4
//5 6
```

This is known as vertical concatenation. Note that in conrunas for matrixes or other n> 1 dimensional arrays, same number of columns per row is not required in order to be able to perform a concatenation.

It is possible to concatenate arrays of higher dimensionality than 2. Note however that the the elements being concatenated cannot vary by more than one dimension degree, it's not possible to concatenate a matrix with a 4 dimensional array.

Concurnas also supports appending elements to arrays, also known as horizontal concatenation. The number of rows needs to match in order to do this however the number of columns per row does not..

```
m = [ 1 2 ; 3 4]
res = [m m]

//res ==
//1 2 1 2
//3 4 3 4
```

This also works for when one needs to add a single item to an array:

```
a = [ 1 2 3 ]
appended = [ a 34 ]

//appended resolves to:
//1  2  3  34
```

In cases where it is not possible to perform a horizontal concatenation, say due to array dimensionality, at attempt to mix primitive type arrays with object arrays or differing primitive type arrays, then a generalized append will take place:

```
a = [1 2 ; 3 4]
b = [5 6]

myAr = [ a b]//myAr is now of type Object[] as it's not possible to perform a
    horizontal concatenation
```

Compiler note: Concurnas automatically resolves the potential for ambiguity in cases where one wishes to concatenate a variable and an array, which would normally be interpreted as a array/list indexing operation:

```
a = 9
res = [a [1 7] ]
```

Nevertheless a array indexing operation will take precedence if it is semantically valid (i.e. a resolves to a list map or array and the expression contained within the bracket is suitable as an index).

Developer note: Using arrays as a data structure where a lot of adding/removing of elements is taking place is discouraged for performance reasons (a new array must be created for every `[ ... ;. ...]` call). Using an appropriate implementation of a list would be more sensible solution for these types of problem.

15.8 Array and List Lengths

The length of an array is obtainable be referencing the `length` synthetic field. Note that it is not possible to change the length of an array by attempting to assign a value:

```
myAr = [1 2 3]
size = mylist.length

//size == 3
```

The JDK Java List interface API specifies the length of a list is obtainable by calling the size method on the object:

```
mMylist = [1, 2, 3]
size = mylist.size()

//size resolves to: 3
```

15.9 Array and List indexing

Concurnas supports the same array indexing operations for both lists and arrays. These operators are useful for performing array assignment and extracting values from arrays. As par conventional programming languages array indexes start from zero:

```
mylist = [1,2,3]
myarray = [ 1 2 3]

mylist[1] = 99
myarray[1] = 99

fromlist = mylist[0] //obtain the first element of the list
fromarray = myarray[0] //obtain the first element of the array

//mylist == [1, 99, 3]
//myarray == [1 99 3]
//fromlist == 1
//fromarray == 1
```

Ranges of arrays and lists can be extracted as follows:

```
myarray = [1 2 3 4 5 6 7 8 9 10]//this applies to lists as well
```

```
range = myarray[ 3 ... 6]
prear = myarray[ ... 6]
postar = myarray[ 6 ...]


//range == [4 5 6 7]
//prear == [1 2 3 4 5 6 7]
//postar == [7 8 9 10]
```

More than one dimension must be specified if the array is of more than one dimension:

```
myar = [1 2 3; 3 4 5]
v = myar[1,1]

//v == 4
```

The expression selecting elements in the selector may also specify a complete subsection of a matrix using ;:

```
myar = [1 2 3; 3 4 5; 6 7 8]
v = myar[;,1]

//v == [2 4 7]
```

Ranges can be specified in this form:

```
myar = [1 2 3; 3 4 5; 6 7 8]
v = myar[1 ... ,1]

//v == [4 7]
```

15.10 The del statement

The del statement can be applied to a list in order to remove an item.

```
mylist = [1, 2, 3, 4]
del mylist[2]
//mylist == [1, 2, 4]
```

This is semantically equivalent to calling `mylist.remove(2)`.

15.11 Operations on arrays and lists

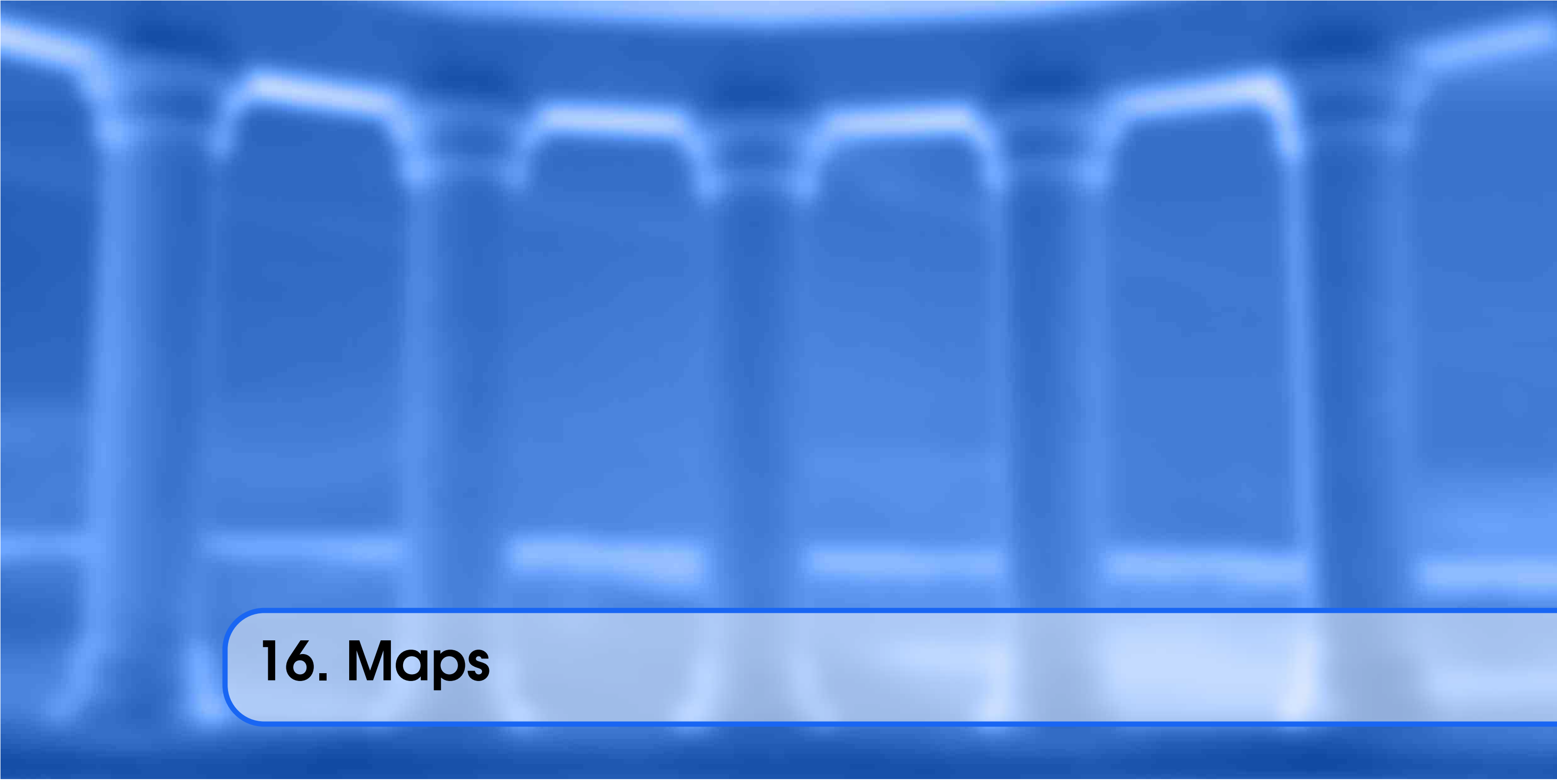
In addition to assignment and obtination of values in an array. All the Concurnas operators are provided for array and list elements.

15.11.1 Assignment operators

The assignment operators as well a pre/post dec/incrementors can be used on array and list elements, for example:

```
myarray = [1 2 3 4 5 6]
myarray[0] += 1
myarray[1] **=2
```

```
myarray[3]++
++myarray[4]
res = myarray[5]--

//myarray == [2 4 3 5 6 5]
//res == 6
```



16. Maps

Concurnas has first class citizen support for creating and working with maps. Maps are an extremely useful data structure which enable us to map from one value to another in a generally very efficient manner. First class citizen support for maps in Concurnas utilizes instances of `java.util.HashMap<Key, Value>` - they take advantage of generics in order to provide one uniform, efficient, implementation.

16.1 Creating maps

Concurnas offers a number of convenient mechanisms for creating maps. Starting with the most verbose:

```
mymap = new HashMap<String, int>()
```

Above, we have created a HashMap instance which maps from elements of type String to Integer (the primitive type int is auto boxed to be of object type Integer).

Concurnas comes pre packaged with an auto included map typedef which can be used in order to create a map in a more concise manner:

```
mymap = map<String, int>()
```

We can take advantage of Concurnas's usage based generic type inference mechanism in order to qualify the generic parameters of the map, saving us the need to define them when constructing the map initially:

```
mymap = map()
```

Then, based on usage of `mymap` (e.g. putting items into the map) Concurnas will infer the generic types, i.e. what type is the key and what type is the value.

16.1.1 Creating maps with initial values

In the case where we need to create a map having an initial set of key value associations, then the following syntax (similar to that used for lists and arrays) comes in handy:

```
mymap = {'taste' -> 10, 'colour' -> 4+1,'shape' -> 8}
```

Above we are defining a map as a comma separated list of key value pair maps via `->`. The individual keys and values are expressions. The generic Key type of the map is taken to be the most specific type of all the keys, and the generic Value type is taken as the most specific catch all type of the values. In this case the map is a mapping between Strings to Integers.

16.2 Operations on maps

Concurnas has first class citizen support for five key operations on maps: putting values, getting values, checking the presence of keys, removing values and iterating over keys:

16.2.1 Putting values

Values can be added to a map one at a time as follows:

```
mymap = map()
mymap['taste'] = 10
mymap['colour'] = 5
mymap['shape'] = 8
```

The above support allows us to avoid having to call the `put` method on the map in this way: `mymap.put('taste', 10)`

16.2.2 Getting values

Values can be obtained from a map:

```
mymap = map()
mymap['taste'] = 10

got = mymap['taste']//got now equals 10
```

The above support allows us to avoid having to call the `get` method on the map in this way: `got = mymap.get('taste')`

16.2.3 Checking the presence of keys

We can test to see if a particular key is present within a map via the `in`, `not in` keywords:

```
mymap = {'taste' -> 10, 'colour' -> 5,'shape' -> 8}
assert 'taste' in mymap
assert 'weight' not in mymap//'weight' is not present in the map
```

16.2.4 Removing values

Values can be removed from a map using the `del` keyword

```
mymap = map()
mymap['taste'] = 10

```

```
del mymap['taste']//value corresponding to 'taste' no longer exists
```

This saves us from having to write: `mymap.remove('taste')`

16.2.5 Iteration

We can iterate over the keys of a map as follows:

```
mymap = {'taste' -> 10, 'colour' -> 5,'shape' -> 8}

for(key in mymap){//iterate over the keys
   value = mymap[key]
}
```

The order of map keys is not guaranteed between differing iterations over them.

16.2.6 More operations

Maps in Concurnas are instances of HashMaps. As such they have access to all of the methods provided by the JDK on which you're running your Concurnas session.For example:

```
mymap1 = {'taste' -> 10, 'colour' -> 5,'shape' -> 8}

mymap2 = map()
mymap2.putAll(mymap1)//copy contents to new instance

assert mymap1 == mymap2
```

For an exhaustive list of methods which are invocable on instances of HashMaps for your JDK (e.g. Java 9) see here: JDK docs.

16.3 Default maps

When using maps there is often a need to have an initial default value returned for a key in the absence of one in cases where one is extracting a value from a map. Let's say we are creating a count all instances of a number. Normally we'd have to write something like this:

```
inputdata = [1,2,4,3,2,1,2,3,4,2,2,3,4,3,2,1]

counter = map()
for(x in inputdata){
   if(x not in counter){
      counter[x] = 0
   }
   counter[x]++
}

//counter now resolves to:
//{1->3, 2->6, 3->4, 4->3}
```

The above is quite verbose. When defining a map, one can use the default keyword mapped to a function which will be called in instances where a value for a requested key is not present within the map. This allows us to solve our previous problem in a far more elegant manner:

```
counter = {default -> def (a int) { 0 }}
counter[x]++ for x in inputdata
```

The default value output from the function above will be persisted within the map at the point of initial use.

16.3.1 Creating default maps

Default maps in Concurnas are instances of class: `concurnas.lang.DefaultMap<Key, Value>`. As with conventional maps there are a few ways in which they can be created, the first of which we have already seen above, where we pass a lambda taking one argument (the Key) and returning one default value (the Value):

```
counter = {default -> def (a int) { 0 }}
```

The above is useful in cases where one wishes to use the passed key in some manner so as to derive an appropriate default value for that key.

An alternative way to create a default map is to use a single expression as follows:

```
counter = {1->"value", default -> 0}
```

The above is useful in cases where the Key of the map derivable since there exists a non default mapping, above it is since we have a key value mapping from an Integer to a String.

A more verbose mechanism to create a default map, bypassing the first class citizen on creation support, can be achieved as follows:

```
counter = new concurnas.lang.DefaultMap<int, int>(def (a int) { 0 })
```

One can use the auto imported `map` function, with a lambda passed in order to create a default map like so:

```
counter = map(def (a int) { 0 })
```

Using the `map` function, along with Concurnas's support for anonymous lambda function definitions, and usage based generic type inference allows us to write some very succinct code:

```
counter = map(a => 0)
counter[0]++
```

16.4 String maps

Concurnas has special support for 'string maps'[1]. All maps in Concurnas have this support. This allows us to do two convenient things:

16.4.1 dot operator on keys

In the case where our maps keys are of type String (or originate from 'identifiers' as par above), we can use it in the following way:

```
mymap = {'akey' -> 12, 'another' => 14}

what = mymap.akey
//what now resolves to 12
```

In fact, we can chain maps together in a nested fashion:

[1]Inspired by javascript

```
mymap = {'another' -> {'thing' -> 'nested value'}}
what = mymap.another.thing
//what now resolves to 'nested value'
```

16.4.2 Identifiers as string keys and values

Concurnas will recognise identifiers defined in maps with initial values that do not otherwise refer to an identifiable element in scope such as a variable name, method name etc, as being strings. This allows us to easily make single word string value/key maps without having to use the `''` or `""` notation:

```
mymap = {akey -> 12}

what = myma.akey
//what now resolves to 12
```

Of course we may also to use the `''` or `""` notation for strings mixed with 'identifier' strings:

```
mymap = {akey -> 12, 'another' -> 1}

what = mymap.another
//what now resolves to 1
```

16.4.3 Object hashCode and equality

Most of the time we need not concern ourselves with the internals regarding how maps operate. But sometimes it is appropriate to understand this. Internal to a the maps described here are a set of key buckets into which keys are mapped to, this mapping is achieved by generating a hash code (a 32 bit integer) for the key object, this is then used to set the key value association into an appropriate bucket. Since more than one key object can have the same hash code[2] clashes can occur. When this occurs on extraction of a value from the map say, then values within the bucket are examined one by one to find the first match. This is achieved by checking the key object equality. Thus in the best case the computational complexity of our map when extracting a value is $O(1)$, however, in the worst case where all the key objects in our environment reside within the same bucket (perhaps due to an error in the hash code generation algorithm), then our computational complexity drops down to $O(n)$

When designing hash code generation algorithms it's important to ensure that the set of unique objects of one's class which are expected to be created and stored in a map have hashcodes that are as spread out as much as possible within the 32 bit integer space of the hashcode return value. This is to avoid clustering of values within a map bucket leading to our detrimental $O(n)$ worst case described above.

All objects in Concurnas are required to have a hashCode and equals method defined. By default Concurnas will automatically generate a `hashCode` and `equals` method for all classes. These methods will ensure that equality is implemented by value. These override the default provided by the JVM which provide referential equality.

For more details of the nature of the default auto generated `hashCode` and `equals` methods see "Automatically generated equals and hashcode methods".

[2]32 bits of information is not enough to enumerate the entirety of existence!

16.5 Map Gotchas

There are a few gotcha's associated with maps in Concurnas and maps in general which often catch out even seasoned pros.

16.5.1 Objects with non fixed hashcodes

Objects with non fixed hashcodes that are used as keys can be problematic. The main source of error stems from the fact that as the mutable state of an object used as a key changes the hashcode of said object, more of than not (depending upon the implementation) will change as well. This means that a key which previously mapped to one internal map bucket is very likely (but not 100% certain which can lead to false positives) to map to a different bucket post state mutation. This can lead to the following sorts of errors:

```
class Person(-name String, ~age int)


fred = Person('fred', 21)
mymap = {fred -> 9}

res1 = mymap[fred]
fred.age++
res2 = mymap[fred]

//res1 == 9, but res2 is null!
```

We can see above that we are using the same object as a key, yet in the second instance, post object mutation, when extracting a value nothing is found and we obtain null. There are two mitigations for this sort of problem, the first to use either a `java.util.IdentityHashMap` or to use only objects with immutable state as keys.

16.5.2 Implicit cast of map keys

Some care must be applied when one is using keys of a boxed primitive type. Since the `java.util.Map` specification defines the `get, put, remove` etc methods as taking a key argument of type object, when boxing a primitive type key element this boxing will be of the primitive types natural boxed counterpart (`Integer` for `int`, `Character` for `char` etc) - this is after all an Object type, but it may not be of the Generic key type defined for the map.

Hence for maps of a boxed primitive type we can end up with some surprising behaviour when auto casting and auto boxing primitive types:

```
mymap = map<Character, String>()
achar = 'g'
charAsInt = achar as int

mymap[achar] = "value"

res = mymap[charAsInt]//res is null!
```

Above, `res` resolves to `null` as when the `int charAsInt` is boxed to an `Object`, it is boxed to an `Integer` and not a `Character`.

The solution is to always explicitly cast into the primitive type which we have stored keys in:

```
mymap = map<Character, String>()
achar = 'g'
```

```
charAsInt = achar as int

mymap[achar] = "value"

res = mymap[charAsInt as char]//res is value as expected
```

Above, `charAsInt` resolves to `"value"` as expected.

17. Delete

Concurnas provides a mechanism where by a local variable can be removed from scope via the `del` keyword. This has the additional side effect for non array Object types of invoking the delete method on them if one is defined. This is incredibly useful for performing resource management and is used in supporting the Concurnas gpu parallel computing as well as the off heap memory frameworks. Additionally Concurnas offers first class citizen support for the `del` keyword when applied to maps and lists.

17.1 Removing values from maps

Values can be removed from a map as follows:

```
mymap = {taste -> 10}

del mymap['taste']//value corresponding to 'taste' is deleted
```

This saves us from having to write: `mymap.remove('taste')`

17.2 Removing values from lists

Values can be removed from a list as follows:

```
mylist = [5, 34, 2, 5, 11]
del mylist[1]//the second value, '34', is removed from the list
```

This saves us from having to write: `mylist.remove(1)`

17.3 Deleting Objects

Calling the delete operator on a variable in Concurnas does two things:

1. The variable is removed from scope
2. The delete method is called on the object pointed to by the variable

Here is an example of this in action:

```
deleteCalled = false
class DeleteMe{
   override delete() void => deleteCalled = true
}

todel = DeleteMe()
del todel
//todel is now out of scope and cannot be referenced
assert deleteCalled
```

Overriding the `delete` method as above will allows us to implement functionality to be invoked upon the `del` operator being applied to the object (or the `delete` method being called directly). For instance, closing or otherwise managing resources.

17.4 @DeleteOnUnusedReturn Annotation

The `@com.concurnas.lang.DeleteOnUnusedReturn` annotation can be used to denote that the delete method should be invoked on the return value of a method or function if it is unused by the caller (i.e. popped off the stack). This affords a degree of flexibility in API design around objects which require resource management as one does not need to worry about the caller correctly dealing with a return value which they may to be optional. This is used heavily in the support for Concurnas parallel GPU computing framework and is useful with off heap memory management. For example:

```
from com.concurnas.lang import DeleteOnUnusedReturn

@DeleteOnUnusedReturn
class ClassWithResource{
   def delete(){
      //...
   }
}

def doWorkAndGetClassWithResouce(){
   ret = ClassWithResource()
   //do some work here
   ret
}

doWorkAndGetClassWithResouce()//the delete method on the returned
    ClassWithResource object will be called
```

In the above example the delete method will be called on the returned `ClassWithResource` object as it is unused by the caller. If it were used by the caller (e.g. assigned to a value, or used in as part of a nested method or function invocation) then the delete method will not be called.

Notes:

1. The delete method will not be called if the returned object is null.
2. If a method is decorated with `@DeleteOnUnusedReturn` then instances in all subclasses will also inherit this decoration.
3. The annotation is not applied to refs to methods or functions which have been decorated with it.

Additionally, for refs which are returned from a function or method and are immediately extracted by the caller, these will also have the delete method called on the ref if the `@DeleteOnUnusedReturn` annotation is used. For example:

```
from com.concurnas.lang import DeleteOnUnusedReturn

@DeleteOnUnusedReturn
class ClassWithResource{
   def delete(){
      //...
   }
}

def doWorkAndGetClassWithResouce(){
   ret = ClassWithResource()
   //do some work here
   ret://returned as a local ref
}

got = doWorkAndGetClassWithResouce()//the delete method on the returned Local
    ref holding the ClassWithResource object will be called (but not on the
    ClassWithResource itself)
```

18. Functions

Functions are a major part of procedural programming. They allows us to split up our programs into subroutines of logic designed to perform specific tasks. They are an incredibly useful abstraction which is used at all levels of the computational process of transforming a human readable description of how to perform a task, right down to the level machine code that our computer CPUs can understand.

Broadly speaking, a function takes a set of inputs variables and returns an output (or outputs if one takes advantage of Concurnas' ability for functions to return tuples) having executed code specified in a block of code associated with the function. The function has a name in order to make it possible for other functions to call it. The input variables have their specified types as does the return value. The input variables and return type constitute the signature of the function and along with the name (and package path) must be unique.

Here is a function, the `def` keyword on its own is used to indicate that we are creating a function, followed by the name of the function, and any comma separated input parameters surrounded by a pair of parentheses `( )` and a (optional) return type:

```
def addTogether(a int, b int) int {
   return a + b
}
```

The /return/ keyword is used within a function in order to cease further execution and literally return the value on the right hand side of it from the function (or the innermost nested function if they are nested).

Now, the above is a perfectly acceptable way to define a function and although it is very verbose, is often the preferred method when writing complex code, or code for which the intended audience may require the extra verbosity in order to aid in their understanding of what is happening.

But there are a few refinements to the above which can make writing functions in Concurnas a quicker, more enjoyable less verbose experience with very little compromise to clarity.

Firstly, the type of the return value is usually inferable by Concurnas, in the above case it's `int` so we can omit this from the definition and leave it implicit:

```
def addTogether(a int, b int) {
   return a + b
}
```

Next we know that blocks are able to return values, so we don't need the `return` keyword at all:

```
def addTogether(a int, b int) {
   a + b
}
```

Now let us use the compact one line form of the `block`, via `=>`:

```
def addTogether(a int, b int) => a + b
```

The above is functionally identical to our first definition but far more compact. It's a matter of discretion in so far as the degree to which one wishes to compact one's functions definitions, sometimes a less compact, more verbose form is more appropriate.

18.1 Functions vs Methods

In Concurnas, a distinction is drawn between the concept of functions and methods.

Simply put, functions are defined at root source code level, methods are defined within classes and have access to the internal state of instance objects of their host class (and any parent nestor classes if relevant), via the `this` and `super` keywords. Methods are covered in more detail here See Classes and Objects section. For now lets look at a simple example highlighting the distinction:

```
def iAmAFunction() => "hi I'm a function"

class MyClass(id int){
   def iAmAMethod() => "hi I'm a method, my class holds the number: " + this.id
}
```

18.2 Calling functions and methods

Ordinarily, we require three things when calling a function, 1). the name of the function, 2). input arguments to qualify it's input parameters and 3). an understanding of the return type.

We can call our `addTogether` function defined previously as follows:

```
result int = addTogether(1, 1)
```

Also, we may use named arguments in order to call our function. This can often make method calls easier to read, especially where there are lots of arguments involved, some with default values some not etc. Named arguments do not have to be specified in the order in which they are defined in the function:

```
result = addTogether(a=1, b=1)
result = addTogether(b=1, a=1)
```

The above two calls to `addTogether` are functionally identical.

When it comes to calling (or invoking) methods, we need an additional component; an object to call the method on. To indicate that we are calling a function on a method, we need to use a dot `.`, for example:

```
class MyClass(state int){
   def myMethod(an input) => state += an; state
}

obj = new MyClass(10)
result = obj.myMethod(2)

//result == 12
```

If instead of whatever is returned from the method (if anything) we wish to return a reference to the object upon which we called the method, we can use the double dot notation: `..`:

```
obj = new MyClass(10)
result = obj..myMethod(2)..myMethod(10)

//result == 22
```

The double dot notation `..` is particularly useful when we need to chain together multiple calls on the same object and do not wish to do perform any sort of operation on the returned values from the intermediate method calls.

We can use named arguments when calling methods:

```
obj = new MyClass(10)
result = obj.myMethod(input = 2)

//result == 12
```

18.3 Input parameters

Functions specify a comma separated list of input variables consisting of a name and type. Each input variable name must be unique. They may optionally be preceded by `var` or `val`:

```
def addTwo(var a int, val b int) => a+b
```

18.4 Default arguments

Function input arguments may specify a default value to use in cases where the input argument is not specified by the caller. When this is done, the type of the variable does not have to be specified if you're happy for Concurnas to infer the input parameter type:

```
def doMath(an int, b int = 100, c = 10) => (an + b) * c
```

Then, when we call a function with default arguments, we do need to specify the arguments for which a default value has been defined:

```
res = doMath(5)

//res == 1050
```

18.5 Varargs

Function parameters may consume more than one input parameter if they are declared as a vararg. A varag input parameter is signified by postfixing ... to the type of the parameter - note that this converts the input parameter to be an array if it's a single value type, or an n+1 dimensional array if it's already an n dimensional array. For example:

```
def stringAndSum(prefix String, items int...){
   summ = 0L
   for( i in items){
      summ += i
   }
   prefix + summ
}
```

We can call a function with vararg parameters we can pass as many inputs to the vararg component as we need, seperated via a commas as par normal function invocation arguments:

```
result = stringAndSum("the sum is: ", 2, 3, 2, 1, 3, 2, 1, 3, 4, 2, 4)

//result == "ths sum is: 27"
```

The vararg may alternatively be passed as an array type (or n+1 dimensional array as eluded previously):

```
result = stringAndSum("the sum is: ", [2 3 2 1 3 2 1 3 4 2 4])

//result == "ths sum is: 27"
```

It's perfectly acceptable to not pass any input to the vararg parameter, e.g:

```
result = stringAndSum("the sum is: ")

//result == "ths sum is: 27"
```

An n dimensional array type can be used as a varrag:

```
def stringAndSum(prefix String, items int[]){
   summ = 0L
   for( i in items){
      summ += i
   }
   prefix + summ
}

result = stringAndSum("the sum is: ", 2, 3, 2, 1, 3, 2, 1, 3, 4, 2, 4)

//result == "ths sum is: 27"
```

18.6 Nested functions

Nested functions are appropriate in two cases:

1. It makes sense to break one's code down into a subroutine - for instance, in order to avoid what would otherwise be code duplication

2. One wishes for that sub function to only be callable within the nestor function - i.e. the nestor function is the only caller.

A nested function is simply a function defined within a function. The scope of that function is bound to the scope of the nestor function, it cannot directly be called by code outside of the nestor function. Example:

```
def parentFunction(apply int){
   result = 0L
   def dosomething(){
      result + (result + apply) * apply
   }
   //we wish to perform the above four times but avoid the code duplication, we
       also don't require any other code outside of parentFunction to be able to
       call it.

   result = dosomething()
   result = dosomething()
   result = dosomething()
   result = dosomething()
}
```

When it comes to using variables which are defined in the nestor function within the nested function, they are implicitly passed to the function but the nested function itself is defined as if it were separate from the nestor. For this reason, and by virtue of the fact that Concurnas uses pass by value for function arguments when calling functions the following is true:

```
def parentFun(){
   parentVar = 100

   def nestedFunc(){
      parentVar += 100
      parentVar
   }

   result = nestedFunc()
   assert result == 200
   assert parentVar ==100
}
```

We see above that although our `nestedFunc` has access to a copy of the value of the nestor variable `parentVar` (as it is implicitly passed into the function), changes made to that variable within the function do not apply to the one in scope of the `parentFun`.

But note that if we pass in an object, a copy of the reference to that object is passed to the nested function, so the behaviour is as follows:

```
class IntHolder(~an int)

def parentFun(){
   parentVar = IntHolder(100)

   def nestedFunc(){
      parentVar.an += 100
      parentVar.an
   }
   result = nestedFunc()
   assert result == 200
   assert parentVar.an == 200
}
```

As `parentVar` holds a reference to an object, the reference is copied, not the object itself, therefore the nestor function `parentFun` and nested function `nestedFunc` versions of the object referenced by variable `parentVar` are the same - they are shared.

18.7 Recursion

Recursion is the process by which a function, either directly or indirectly calls itself. Concurnas permits function recursion (except for within GPU functions and GPU kernels). The classic textbook example of this being factorial number calculation:

```
def factorial(n int){
   match(n){
      1 or 2 => 1
      else => factorial(n-1)+factorial(n-2)
   }
}

res = factorial(5)

//res == 5
```

It turns out that the above, for any value of n greater than 2 performs a lot of unnecessary repetitive work in terms of calling factorial for values already previously calculated. There are far better ways of calculating factorial numbers (some of which don't use recursion at all). Here is a different recursion example more likely to be seen in the wild, a tree traversal:

```
open class Node

class Branch(~children Node...) < Node
class Leaf(~value String) < Node

def create(){
   Branch(Branch(Leaf("a"), Leaf("z")), Leaf("c"))
}

def explore(node Node){
   match(node){
      Branch => String.join(", ", (explore(n) for n in node.children))
      Leaf => "Leaf: {node.value}"
   }
}

tree = create()
res = explore(tree)

//res == Leaf: a, Leaf: z, Leaf: c
```

One thing to bear in mind with recursion is the fact that as we recurse, with every direct or indirect self call, we deepen the call stack. This is not a big deal if we recurse to a small degree, but if we are recursing a lot[1], then we will be eating up our call stack which may result in us running out of stack causing a `java.lang.StackOverflowError` exception to be thrown. It is for this reason that some organizations restrict the use of, or even outright ban the use of recursion.

[1]*A lot* sounds vague but this is intentional because the stack size is platform specific, so really we cannot be more precise than this.

II

Object-oriented

She says she’s an 80’s baby
She goes for the bad boy type
She says she’s an 80’s baby
In the club she dance all night

- All Night from Synthetic Love LP by Trevor Something (2013).

19. Classes

Classes are the cornerstone of object oriented programming and are extensively supported within Concurnas. They are an extremely useful way of programming with data, and more specifically state, in an easily understandable way.

In short they a mechanism by which data in the form of state expressed via fields, and functionality expressed via methods can be encapsulated together in the form of an object of which a class acts as a template. A class hierarchy may be established via inheritance and functionality may be ’composed into’ classes via the use of traits.

Object orientation helps reduce the gap between the way machines operate and humans think by allowing us to think about data/state manipulation using everyday concepts, or at least concepts which more closely map to our problem domain than the abstract terminology of a computers machine code. Consider the following two representations of data:

```
peopleA Object[2] = [ ["dave" 45] ["freddy" 16] ["monica" 33]]

class Person(name String, age int){
  def incAge() => age++;;
}
peopleB = [Person("dave", 45) Person("freddy", 16) Person("monica", 33)]
```

`peopleA` is declared as a matrix, it’s a perfectly adequate way of representing our people data, but it is quite abstract - we’re not able to determine what `peopleA` really is representing without additional context. On the other hand `peopleB`, through its use of representing its individual elements via `People` instance objects, makes it immediately clear what it’s representing.

The case for classes is further strengthened when we consider manipulation of the state of a person. Let’s write some code to increment the age of each person by one:

```
for(person Object[] in peopleA){//approach 1
  person[1]++
}
```

```
for(person Person in peopleB){//approach 2
   person.incAge()
}
```

For ’approach 1’, manipulating the internal state of people held in the `peopleA` matrix is quite an abstract process. It’s not clear from looking at the code alone that we are incrementing the age of each person. Also, a more major hidden problem is that in the future if we wish to capture more information about a person, we need to change the structure of the data in the matrix - this would have a knock on negative impact on all instance of the code seen above, would the element in slot ’1’ of the array be the age still? With ’approach 2’ these problems are mitigated; we’re able to see that we are calling a method which increments the age of a person `incAge` by just looking at the code. Also, we’re free to change the internal structure of the person state without fear of breaking dependant code like this.

Object orientation does have some disadvantages. One major one being that it can be a very verbose process to write object oriented code (for legitimate reasons as we shall see here). Concurnas offers many useful syntactic and semantic tools for reducing this verbosity, which are described in detail later on. These include:

- One line class and superclass arguments via class declaration arguments.
- Automatic generation of setters and getters.
- Automatic redirection to getters and setters.
- Automatic generation of `hashCode` and `equals` methods to provide equality by value.
- Sensible accessibility modifier defaults.

19.1 Creating Objects of Classes

We can create instance objects of classes using the new operator:

```
class MyClass

inst = new MyClass() //create a new instance object of class 'MyClass'
```

But we can also choose to omit the `new` keyword all together:

```
class MyClass

inst = MyClass() //create a new instance object of class 'MyClass'
```

For classes which are defined in a separate source file from the one in which the new call is made (i.e. the case 90% of the time when programming in the wild) - the relevant class will need to be either referenced by its full class name, or imported and optionally given an associated name:

```
from java.util import ArrayList
from java.util import ArrayList as ARList

ar1 = ArrayList<String>()
ar2 = ARList<String>()
ar3 = java.util.ArrayList<String>()
```

All three of the above are equivalent. Note that we use generics when creating the above ArrayList instance objects, this is covered in more detail later.

19.2 Creating Classes

Classes are declared by use of the /class/ keyword followed by the name of the class. Usually one begins the name of a class with an Uppercase letter but this is not obligatory. The class may or may not have a block attached to it. For example:

```
class MyClass1//class with no block

class MyClass2{}//class with an empty block

class MyClass3{//class with a method defined
   def amethod() => 12
}
```

19.3 Fields

Classes allow variables, to be defined within themselves such that instance objects of those classes can maintain state within themselves. Variables like this are called fields. Example:

```
class MyClass{
   count int = 0
   name = "Fred"
}
```

Fields can be declared without initial assignment since it is anticipated that they will be assigned a value within a constructor call chain. If they are not assigned a value within a constructor call chain then a default value will be assigned contingent on the type of the field as follows:

Type	Value
`int`, `long`, `short`	0
`byte`	0b
`char`	”
`boolean`	false
`double`, `float`	0.0
`Object` or subclass of `Object`	null
n dimensional array of any component type	null
actors, tuples, etc	null
refs	a non-null instance of the ref type with no assigned initial value

By default fields are `var`’s, meaning that they don’t require initial assignment in a constructor and can be changed at any point (provided that the rules concerning accessibility of the field are respected). Declaring a field `val` both enforces initial assignment of the field within a constructor call chain and prohibits subsequent reassignment:

```
class Myclass{
   val name = "dave"//this field cannot be reassigned
   val age int//this field must be assigned a value in a constructor chain -
      otherwise it's a compilation error
   this(age int){
      this.age = age
   }
}
```

19.4 Overriding fields

If a field of the same name and supertype is defined in a superclass as that which we are defining in a subclass, then the subclass may optionally declare that field as being overridden by using the `override` keyword. This will ensure that a 'new' field is not created at the subclass level. For example:

```
open class SuperClass{
  aField = "super"
  anotherField = "super"
}

class Child < SuperClass{
  aField = "child"
  override anotherField = "child"
}

child = new Child()
asSuper = child as SuperClass
[child.aField child.anotherField ; asSuper.aField asSuper.anotherField]

//== [child child ; super child]
```

In the above example we see that when we override `anotherField` we do not create a new field at the `Child` class level, but in fact override the value of the field in the superclass. Note that the superclass field must be accessible to the subclass (i.e. not defined as private in the superclass).

The override keyword may also be used in class definition argument lists as follows:

```
open class SuperClass{
  aField = "super"
  anotherField = "super"
}

class Child(aField = "child", override anotherField = "child") < SuperClass

child = new Child()
asSuper = child as SuperClass
[child.aField child.anotherField ; asSuper.aField asSuper.anotherField]

//== [child child ; super child]
```

The effect is the same as our previous example.

19.5 Methods

Methods are functions declared within classes. Methods have all the same capabilities as functions. Additionally, code defined within methods has access to the keyword `this` which enables access their declaring classes internal state and all other accessible methods. For example:

```
class Person(name String, age int){
  def nameLength(){
    this.name.length()
```

```
   }

   def nameLengthandAge() (int, int) {
      this.nameLength(), this.age
   }
}
```

Note that the this keyword can often be omitted and it left implicit in cases where the right hand side dot does not resolve to something else:

```
class Person(name String, age int){
   def nameLength(){
      name.length()
   }

   def nameLengthandAge() (int, int) {
      nameLength(), age
   }
}
```

Code within methods also has access to the `super` keyword, this is described later in the inheritance section.

19.6 Setters and Getters

It turns out that one of the most popular uses of classes and object oriented programming in general is the concept of encapsulation of data within objects, and providing access to that data via means of access methods, which, via convention take the form of 'getters and setters'. The format is straight forward... To set a field in a class, a setter is used, with name `setX` where `X` is the name of the field being set with the first letter capitalized, and signature `(Y) void` where `Y` is the type of the field being set. To get a field in a class, a getter is used, with name `getX` (or `isX` if the type of the field is of type `boolean` or `Boolean`) where again `X` is the name of the field being set with the first letter capitalized, and signature `() Y` where `Y` is the type of the field being got. For example:

```
class Person(age int){
   def setAge(newAge int) void {
      this.age = newAge;
   }

   def getAge() int{
      return this.age
   }
}

//client code calling these methods:
person = new Person(20)
age = person.getAge()
person.setAge(23)
```

This approach is seen as being favourable to simply permitting direct access to the age variable because if in the future we wish to change the internal design of our `Person` class so as to make age a derived piece of information, we can easily substitute that logic in place to the `setAge` and `getAge` methods without having to change all the individual parts of code directly accessing the age field (wherever they may be in our codebase...) to accommodate this logic. For example, let's make age derived:

```
nowYear = java.time.Year.now().getValue()

class Person(yearOfBirth int){
   def setAge(newAge int) void {
      this.yearOfBirth = nowYear - newAge;
   }

   def getAge() int{
      nowYear - this.yearOfBirth
   }
}

//our client code calling these methods does not need to change...
age = person.getAge()
person.setAge(23)
```

It can sometimes feel very redundant to have to call `getAge`/`setAge` when all we are doing is getting or setting a field, using the standard assignment operator = can feel more natural. For this reason, Concurnas will automatically map to the appropriate getter/setter method where necessary. For example:

```
age = person.age
person.age = 23

//is internally mapped to and is is equivalent to...
age = person.getAge()
person.setAge(23)
```

Just as calling getters and setters can feel redundant/verbose, so to can writing them in the first place. In fact in languages such as Java, for data centric classes, it's not uncommon for the getter setter code to make up as much as 95% of the class body! To reduce verbosity here Concurnas provides a mechanism by which they can be automatically generated. Simply prefix the field with ~, - or +[1] in order to generate the appropriate setters or getters. Which getters and setters are generated is below:

Prefix	Generated Getter	Generates Setter
~	Y	Y
-	Y	
+		Y

These prefixes can be used as follows:

```
class Person1(-name String){
   ~age int
}

class Person2(name String){
```

[1]In looking at a standard QWERTY keyboard you will notice that whilst - requires only one keystroke, ~ and + require two (shift and the +/~ key). The choice of these characters for auto getter/setter generation is deliberate as the most common case for getters and setters is for fields to need only getters (with instantiation of state achieved via a constructor)

```
   age int

   def getName() String {
      return this.name
   }

   def getAge() int {
      return this.age
   }

   def setAge(age int)void {
      this.age = age
   }
}
```

19.7 Constructors

Constructors are used to create instance objects of classes. They are defined in the same way as methods (and so can have default and varag parameters) as follows:

```
class Person{
   name String
   age int
   this(name String, age int){
      this.name = name
      this.age = age
   }
}
```

Classes may define more than one constructor and constructors may call one another via use of the `this` keyword as follows:

```
class Person{
   name String
   age int
   this(name String, age int){
      this.name = name
      this.age = age
   }

   this(age int){
      this("fred", age)//call another constructor
   }
}

obj1 = new Person("dave", 23)
obj2 = new Person(56)
```

If a constructor calls another constructor this call must be the first entry in the constructor. This arrangement of constructor calling another constructor is known as a constructor chain. The top level (of which there may be more than one) being the constructor which calls a super constructor (via the `super` keyword seen later) or has no /this/ constructor call at all.

All classes must have at least one constructor. This constructor does not have to be publicly accessible. If at least one constructor is not provided then a publicly accessible zero argument constructor will be generated for the class automatically - if this were not the case then it would render the class uninstantiable in any situation, which is not ever useful.

19.8 Class Declaration Arguments

Often, particularly for data oriented classes with simple state and requiring only one or a straightforward set of constructors, having to declared a set of fields and constructors can lead to very verbose code. Concurnas has a remedy for this in the form of class declaration arguments. Here we combine both the field declaration and constructor set into one:

```
class Person(forename String, surname String, age int)
```

The above declaration of /Person/ is equivalent to the more long winded version as follows:

```
class Person2{
   forename String
   surname String
   age int

   this(forename String, surname String, age int){
      this.forename = forename
      this.surname = surname
      this.age = age
   }
}
```

In the above instance we have to write a tenth of the code in order to achieve the same effect (hence allowing us to be 10x more productive!).

19.9 Inheritance

When it comes to implementing class functionality and state it's often beneficial to to be able to establish a class hierarchy of sub and superclasses. Classes may subclass one another, and in doing so the superclass receives access to all the non private (and potentially package) fields and methods of the superclass. Classes may have only one superclass. The `extends` or < keyword is used in order to define a superclass for a class:

```
open class Animal{
   def livingState() => "is alive"
}

class Dog < Animal

class Cat extends Animal
```

All classes are considered closed by default. This means that they cannot be extended with subclasses. The `open` keyword must be used in order to denote that it's acceptable for a class to be subclassed.

Instance objects of `Dog` and `Cat` can be treated as instances of `Animal`. All of the accessible methods (and fields) of `Animal` are accessible on instance objects of `Dog` and `Cat` for example:

```
animals = [new Dog(), new Cat()]

res = animals^livingState()
```

```
//res == [is alive, is alive]
```

If a superclass has constructors then at least one must be invoked by the top level constructor chain of a subclass. Let's refine our previous example:

```
open class Animal{
   def livingState() => "is alive"
}

open class LeggedAnimal(legs int) < Animal

class Dog < LeggedAnimal{
   this(){
      super(4)
   }
}

class Bird < LeggedAnimal{
   this(){
      super(2)
   }
}
```

If a superclass has only a zero argument constructor defined then it is not necessary for subclass constructors to call it explicitly as this will be added implicitly. So for example, both the following are acceptable:

```
open class Animal{
   def livingState() => "is alive"
}

class Dog < Animal{
   this(){
      super()//this is ok, but unecsary
   }
}

class Cat extends Animal{
   int lives
   this(lives int){
      //super() is not called this is fine
      this.lives = lives
   }
}
```

If there are no constructors declared for a class which is explicitly marked as being a subtype of another then its generated zero argument constructor will be generated to call the referenced superclasses zero argument constructor if there is one accessible.

Where a superclass is undeclared, a class will implicitly be a subtype of `java.lang.Object`, an implicit super constructor to `Object` is added to all top level constructor chains. Both the following are equivalent:

```
class MyClass1{
   this(){
    super()
  }
}

class MyClass2
```

19.10 The `super` keyword

Similar to the `this` keyword the `super` keyword is used in two ways, 1). To enable a super constructor invocation (as we have already seen), 2). To allow a subclass method to access the fields and invocation of super class methods. Let's look at point 2...

```
open class SuperClass{
  def aMethod(a int) => a+1
}

class ChildClass < SuperClass{
  def anotherMethod(a int) => super.aMethod(a)
  override aMethod(a int) => super.aMethod(a)+1//using 'super' here lets us
      call the superclass version of 'aMethod' and not this method
}
```

Just like the `this` keyword, `super` may be omitted in instances where there is no other way to match what's on the right hand side of the dot. So we can write the above as:

```
open class SuperClass{
  def aMethod(a int) => a+1
}

class ChildClass < SuperClass{
  def anotherMethod(a int) => aMethod(a)
  override aMethod(a int) => super.aMethod(a)+1 //we cannot omit super here as
      this would result in an infinite stack call loop to 'aMethod'
}
```

19.11 Accessibility modifiers

Class methods and fields may be tagged with one of the following accessibility modifiers. These affect the way in which the field/method to which they are attached can be accessed/called/overridden:

Modifier	Effect on the field/method	Overridable
`public`	Can be accessed/called on any instance object anywhere.	Yes
`protected`	Can only be accessed/called by subclass.	Yes
`package`	Can be accessed/called by member of same package.	Yes
`private`	Can only be accessed/called by the instance object itself.	No

If no modifier is specified then the defaults are:

- Fields - `protected`
- Methods - `public`

The modifiers can be used as follows:

```
class MyClass{
   private field1 int = 9
   package field2 int = 9
   protected field3 int = 9
   public field4 int = 9
   field5 int = 9//defaults to protected

   private def aMethod1() => 12
   package def aMethod2() => 12
   protected def aMethod3() => 12
   public def aMethod4() => 12
   def aMethod5() => 12//defaults to public
}
```

19.12 Overriding methods

If we wish to override a method defined in a superclass we must use the `override` keyword. The keyword must be used so as to prevent us from accidentally overriding a method defined in a superclass. For example:

```
open class A{
   def aMethod1() => 0
   def aMethod2() => 0
}

class B < A{
   override def aMethod1() => 1000//we can choose to include the 'def' keyword
       or omit it
   override aMethod2() => 1000
}
```

If one wishes to prevent subclasses from overriding a method then postfixing its declared name with a dot `.` will prevent this from being possible. For example:

```
open class A{
   def aMethod.() => 0 //this method is now final and cannot be overridden
}

class B < A{
   override aMethod() => 1000//this will throw a compilation error
}
```

An overriding method may not narrow the accessibility of a method being overridden. So the following is not valid:

```
open class Parent{
   public def aMethod() => 12
}

class Child < Parent{
   private def aMethod() => 14//this is not permitted since the accessibility
       modifier is being narrowed.
}
```

As above we cannot override a public method to be private, but it is ok to override a private method to be public in a subclass.

19.13 Abstract classes

Abstract classes allow us to define ’intermediate’ or ’prototype’ classes which are designed to be extended by ’concrete’/non-abstract classes. They are partially implemented classes and can have methods and state. Abstract classes cannot be directly instantiated to make instance objects, only their (non abstract) subclasses can. Abstract classes can extend any class including abstract classes.

Since abstract classes cannot be instantiated, they cannot have constructors.

A class can be made abstract via one of two methods:

1. Using the `abstract` keyword:

   ```
   abstract class LeggedAnimal(legs int)
   ```

   When a class is declared abstract it’s not necessary to declared it being an `open` class, as this is implicit (the purpose of abstract classes is to be extended).

2. By having at least one abstract method defined. Abstract methods are defined in the same way as normal methods, but they do not have a body:

   ```
   abstract class LeggedAnimal(legs int){
      def howFeels() String //this is an abstract method
   }
   ```

   If an abstract class declares any abstract methods (which is the normal case), then these must be implemented in any concrete extending classes:

   ```
   open class Animal{//implicitly an abstract class since we have an
       abstract method declared below
      def reportFoodEaten() String //this method is abstract and must be
          implemented by all concrete sub classes of Animal
   }

   class Panda < Animal{
      def reportFoodEaten() => "bamboo"
   }
   ```

Note above that we do not need to use the `override` keyword when we define `reportFoodEaten`. This is because we are not overriding an existing implementation of `reportFoodEaten`.

Abstract methods may override concrete method implementations from super classes. The following is valid:

```
open class A{
   def aMethod() => 12
}

class AbstractClass < A{
   def aMethod() int
}
```

In the case above subclasses of `AbstractClass` must implement the `aMethod`. Note, direct subclasses of class `A` do not need to implement this method since it’s already been implemented.

An abstract class that extends an abstract class is not obliged to provide a concrete implementation of its abstract superclass’s abstract methods. So the following is perfectly valid:

```
abstract class Animal{
   def reportFoodEaten() String
}

abstract class FurryAnimal < Animal//not obliged to implement reportFoodEaten
```

19.14 Class Declaration Arguments with superclasses

We can use our one line approach to class declaration with class declaration arguments when we need to call a super constructor as follows:

```
open class Animal(favFood String, color String, age int)

class Panda(page int) < Animal('bamboo', 'blackAndWhite', page)
```

Above as we are implicitly creating the constructor for `Panda`, we are instructing Concurnas to add a super constructor invocation to the `Animal` superclass constructor with the body `('bamboo', 'blackAndWhite', page)`. Any expression is permissible in the super constructor argument list. In cases where a field name from the main class declaration arguments is referenced as a superclass super constructor argument, then this argument is not translated into a field for the class being declared. So for `Panda` above, field `page` is not created - it would if `page` were not referenced in the superclass super constructor argument list.

19.15 Automatically generated equals and hashcode methods

Concurnas implements equality by value by default for instance objects. It achieves this by auto generating a method `equals` with the following signature: `equals(Object)boolean` for every class declared. This makes the following code possible:

```
class Person(String name)

p1 = Person("dave")
p2 = Person("dave")

assert p1 == p2
```

Equality is tested via use of the == (and <>) operator. We see above that although `p1` and `p2` are separate instances objects, their value is the same, therefore they are considered equal.

Concurnas will also generate a `hashCode` method with signature `hashCode ()int` which is unique per the value of the class. This is useful for data structures such as HashSet's and HashMaps. For example:

```
class Person(String name)

p1 = Person("dave")
p2 = Person("dave")

pset = new java.util.HashSet<Person>()
pset.add(p1)
pset.add(p2)//p2 will not be added as its value matches p1 which is already in
    the set
```

```
assert p1.hashCode() == p2.hashCode()//hashcode's match
assert pset.size() == 1
```

One is of course free to implement the `equals` and `hashCode` methods manually and override the automatically generated versions. For example here is and example of referential equality:

```
class MyClass{
   override hashCode() => System.identityHashCode(this)
   override equals(an Object) => this &== an
}
```

19.16 `init` block

Concurnas provides an `init` block that can be useful in cases where one is using class definition arguments but needs to include extra code in the generated constructor. For example:

```
pCount = 0

class Person(String name){
   init{
      System out println "Created a Person with name {name}"
      pCount++
   }
}
```

The code within the `init` block is added to the end of any generated constructors, as such they have access to all the fields/methods of the class as par a normal method. More than one `init` block may be specified and they are executed in the linear order in which they were defined.

19.17 Nested Classes

Nested classes are classes defined within classes. They have access to all the methods and fields of their parent nestors scopes in which they are defined. For example:

```
class Outerclass{
   def outerMethod()=> 11

   public class InnerClass{
      def innerMethod(){
         outerMethod() + 1
      }
   }
}
```

If no accessibility modifier is provided when declaring a nested class then it will default to private (this is because most of the time nested classes are best suited for intra class algorithms which do not require exposure outside the parent nestor class).

Nested classes require a reference to their parent nestor in order to create instances of them, we may optionally use the `new` keyword:

```
oc = new Outerclass()
inst1 = oc.new InnerClass()
inst2 = oc.InnerClass()
```

In cases of multiple levels of nesting, the qualified `this` syntax is appropriate for dealing with situations in which we need an reference to a specific parent nestor. The qualified this syntax is as follows, `this[X]` where `X` is the class name for which we wish to obtain a reference to the corresponding parent nestor of, the return type will be of type `X`. For example:

```
class Outerclass{
   private variable = 'outer variable'
   def aMethod(){
      "outer method"
   }

   public class Innerclass{

      public class Innerclass2{ }
      private variable = 'inner variable'

      def aMethod(){
         "inner method"
      }

      def work(){
         outer = this[Outerclass] //here we use the qualified this syntax!
         inner = this[Innerclass] //and here!
         ""+[outer.aMethod(), outer.variable, inner.aMethod(), inner.variable]
      }
   }
}

outer = Outerclass()
inst = outer.Innerclass()
res = inst.work()

//res == [outer method, outer variable, inner method, inner variable]
```

All the other normal operations on classes can be performed on inner classes provided that an instance of the outer class is specified:

```
constructorRefernce = outer.new Innerclass&//constructor refence
classRefernce = outer.Innerclass&//class refence
actorInstance = outer.actor Innerclass()//actor of Innerclass
```

19.18 Local Classes

Local classes are classes defined within functions or methods, they are declarable just like ordinary classes. They are limited in terms of instantiation by the scope in which they are declared.

```
open class SuperClass

def makeLocalClass() SuperClass {
   class MyClass < SuperClass
   return new MyClass()
}
```

We cannot make use of `MyClass` directly (i.e. instantiate an instance of it via the `new` operator) outside the scope of the method in which it is defined, hence the type returned in the above example has to be either a superclass or trait which it implements.

19.19 Anonymous Classes

Anonymous classes provides a means by which classes can be both defined and instance objects derived from the in one relatively compact step. They are useful for implementing SAM types see here and when working with traits see here.

Anonymous classes are easy to define, here is an example:

```
class AbstractClass{
   def operation(a int) int//child classes must implement this method or be
       declared abstract
}

instance AbstractClass = new AbstractClass{ def operation(a int) => a*2 }

instance.operation(2)

//== 4
```

Anonymous classes may be defined without bodies

```
abstract class AbstractClass{
   def operation(a int) int => a
}

instance = new AbstractClass

instance.operation(2)

//== 2
```

Anonymous classes can be used any place where an expression is required:

```
class AbstractClass{
   def operation(a int) int
}

def doOperation(apTo int, an AbstractClass) => an.operation(apTo)

doOperation(2, new AbstractClass{ def operation(a int) int => a*2 })

//== 4
```

Anonymous classes may refer to variables and methods defined outside of the scope of the class:

```
class AbstractClass{
   def operation(a int) int
}

def doOperation(apTo int, an AbstractClass) => an.operation(apTo)

mul = 4
```

```
doOperation(2, new AbstractClass{def operation(a int) int => a*mul })

//== 8
```

Generics may be used in anonymous classes as normal:

```
class AbstractClass<X>{
   def operation(a X) X
}

def doOperation(apTo int, an AbstractClass<Integer>) => an.operation(apTo)

mul = 4
doOperation(2, new AbstractClass<Integer>{ def operation(a Integer) Integer =>
    a*mul })

//== 8
```

Note that for an anonymous class definition both a class (inaccessible) and object of the anonymous class are created. As such the anonymous class may only have a zero arg constructor defined (if at all), it may not use augmented constructors.

Black feathers in the storm, where will we fall?
Will we ever be home?
Let's let the wind decide, where we are going to fall
Soaring to a destination
She takes me away

- Black Feathers from Black Anima by Lacuna Coil (2019).

20. Traits

Traits allow us to define elements of reusable functionality (both methods and state) which can be mixed-in to classes. They are themselves not instantiable as, like abstract classes, they represent incomplete fragments of functionality. Unlike abstract classes however, a class may be composed of (aka mixed-in) more than one trait. Traits are useful as they encourage and facilitate software creation via composition as opposed to inheritance, which generally speaking is a better design pattern in object oriented programming.

Traits can be compared to interfaces with default methods such as in languages like Java. However, they differ in that they can be chained together, may extend non trait classes and allow classes to be composed of stacked trait super method calls.

20.1 Defining traits

Traits are defined in a similar way to classes. But they may not be nested as they don't have access to nested state. Like abstract classes, they may define both concrete and abstract methods.

```
trait Eater{
  def eat() String => "eating {favouriteFood()}" //concrete method
  def favouriteFood() String //abstract method to be implemented by composing
      class
}
```

Above we have defined a trait, Eater having both a concrete method implementation and an abstract method.

Like classes, traits may inherit from other traits, and call super etc as normal:

```
trait Animal < Eater{
  override def eat() String => "animal is " + super.eat()
  def age() => 12 //all animals are the same age in this example, we will
      improve this later on
}
```

Also like classes, traits may inherit from classes (either abstract or concrete):

```
abstract class NonAnimalEntity{
   def nonAnimalType() String
}

trait Plant extends NonAnimalEntity with Eater{
   def nonAnimalType() => "Plant"
   override def eat() String => "plant is" + super.eat()
   def favouriteFood() String => "carbon dioxide"
}
```

Note in the above example, for our Plant trait we are implementing the abstract class `LivingEntity`, and the trait Eater, by using the with keyword. We will look into using the with keyword (and it's abbreviation) in more detail below. We also choose to implement the abstract method, `favouriteFood` at this level, though it can of course be overridden again in any composing classes or traits.

Traits cannot be directly instantiated, rather they are composed or mixed-in to other concrete or abstract classes. For this reason they do not have constructors. They also cannot have `init` blocks.

20.2 Using traits

Following on from our previous examples. Let us now create a concrete class, composed of a trait, and make use of it. In order to compose a class of a trait we use the `with` keyword, the abbreviation of which being ~. Multiple different traits may be used in the definition of a class, by separating them with a comma `,`.

```
trait FourLegged ~ Animal

class Dog with Animal, FourLegged {
   def favouriteFood() => "sausages"
}

fido = new Dog()
fido.eat()//eat returns: "animal is eating sausages"
```

The following code is perfectly valid since an object of `Dog` class is composed of the `Animal` trait:

```
def animalFoo(animal Animal) => animal.eat()

fido = new Dog()
animalFoo(fido)

//returns: "animal is eating sausages"
```

We can use traits as types. Hence, we are able to use `is` and `as` with them:

```
something Object = new Dog()

ageIfAnimal = (something as Animal).age() if(something is Animal) else "not an
    animal"
```

When trait methods are called, they take precedence over matching superclass method definitions. They may also call superclass method instances. The exact order of method invocation is contingent on how the class upon which they are composed with has ordered its composed traits as par a feature called linearization described in more detail later on below.

```
trait SuperTrait{
   def foo() String => "superTrait method called"
}

trait ATrait < SuperTrait{
   override def foo() => "trait method called " + super.foo()
}//override keyword must be used since we are overriding the definition from the
    super trait: SuperTrait

class AClass with ATrait

AClass().foo()
//returns: trait method called, superTrait method called
```

As stated previously, traits may extend non trait classes, i.e. concrete and abstract classes. If a concrete class is composed of a trait which extends a non trait classes, and it itself does not extend a class, then it will implicitly extend the trait classes' non trait superclass.

```
abstract class AbstractClass{
   def foo() String => "superTrait method called"
}


trait ATrait < AbstractClass{
   override def foo() => "trait method called, " + super.foo()
}

class AClass ~ ATrait

aInstance AbstractClass = AClass()
aInstance.foo()
//returns: trait method called, superTrait method called
```

Above, `AClass` implicitly extends `AbstractClass`.

Care should be taken with traits that extend non trait classes as they bind all composing classes to the inheritance chain implied by the superclass which they extend. Since Concurnas does not permit multiple class inheritance via extension (so as to avoid the diamond pattern problem - which is itself solved by using traits) this means that usually only one non trait extending trait may be referenced in the composition of the composing class - assuming of course that the composing class does itself not explicitly extend an unrelated superclass.

Note that the above pattern can still work with a class which itself explicitly extends a superclass, or with more than one trait, provided that the inheritance hierarchy implied for the concrete class remains avoids multiple class inheritance. For example, consider the following cases:

```
abstract class Super1
abstract class Super2
abstract class Child1 < Super1
abstract class Child2 < Super2
```

```
trait Trait1 < Super1
trait Trait2 < Super2
trait Trait3 < Child2 ~ Trait2


class Concrete1 < Super1 ~ Trait2 //this cannot work as Super1 and Super2
    (implicit from Trait1) are not hierarchically related

class Concrete2 < Child1 ~ Trait1//this works as Child1 < Super2 (implicit from
    Trait2)

class Concrete3 ~ Trait2, Trait3//this works as Child2 (from Trait3) < Super2
    (from Trait2)
```

20.3 Generic traits

Traits, like classes, may make full use of Generics. For example:

```
trait Incrementor<X Number>{
   def operation(a X) => a.doubleValue()+1
}

class IntIncrementor ~ Incrementor<int>

new IntIncrementor().operation(12)
//returns: 13.0
```

It's possible to compose concrete classes with reference to a trait more than once in a class definition. Normally this is not a problem but with generics this can result in an incompatible generic type qualification for the trait in question. For example:

```
trait A<X>
trait B ~ A<int>//valid
trait C ~ A<double>//valid

abstract class Class1 ~ B//valid
abstract class Class2 ~ C//valid

abstract class Class3 ~ B, C//NOT valuid
abstract class Class4 < Class1 ~ C//NOT valuid
```

Classes `Class3` and `Class4` are not valid as in resolving the generic type qualification of trait A, we see that it has to be qualified as both `int` and `double`, which is not possible - only one generic qualification in an inheritance/composition tree is permitted.

20.4 Resolving ambiguous trait method definitions

Sometimes, in cases where one composes a class of multiple traits, an ambiguity can arise where two (or more!) traits or superclasses define methods with the same signature (name, and input parameters). In this case manual disambiguation is required by defining an implementation in the concrete class being composed and, if appropriate, directing the call flow to the required method(s). For example:

```
trait A{
   def foo() => "version A"
}
trait B{
   def foo() => "version B"
}

class FooClass with A, B{
   override def foo() => "version FooClass"
}

FooClass().foo()
//returns: version FooClass
```

An alternative strategy is to declare the offending method or holding class to be abstract - leaving resolution to a concrete class:

```
abstract class AbstractFooClass{
   def foo() => "version AbstractFooClass"
}

trait A{
   def foo() => "version A"
}
trait B{
   def foo() => "version B"
}

abstract class FooClass < AbstractFooClass with B, A

class RealFooClass < FooClass {
   override def foo() => "version RealFooClass"
}

RealFooClass().foo()
//returns: version RealFooClass
```

Often, we'd just like to call the rightmost defined trait version of foo by using the super keyword:

```
abstract class AbstractFooClass{
   def foo() => "version AbstractFooClass"
}

trait A{
   def foo() => "version A"
}
trait B{
   def foo() => "version B"
}

class FooClassA < AbstractFooClass with B, A{
   override def foo() => super.foo()
}

class FooClassB < AbstractFooClass with A, B{
   override def foo() => super.foo()
}
```

```
fc.foo() for fc in [FooClassA(), FooClassB()]
//returns: [version A, version B]
```

The above are fine solutions, but usually we'd like to call one, or more, versions from the offending method from the traits by referring to them explicitly. To this end we can use the qualified super syntax:

```
Super [ className ]
```

Using qualified super we may refer to any superclasses or traits referenced in the current class definition. For example:

```
abstract class AbstractFooClass{
   def foo() => "version AbstractFooClass"
}

trait A{
   def foo() => "version A"
}
trait B{
   def foo() => "version B"
}

class FooClass < AbstractFooClass with B, A{
   override def foo() => "" + [super[AbstractFooClass].foo(),
       super[A].foo(),super[B].foo()]
}

FooClass().foo()
//returns: [version AbstractFooClass, version A, version B]
```

20.5 Stateful traits

Traits may have state, i.e. fields. They must either be assigned a value when defined within the trait or be assigned a value within a class composed with the trait. E.g.

```
trait IndAncDec{
   -count int
   -countdown int =0
   def inc() => ++count + countdown
   def dec() => --countdown + count
}

class HasIncAndDec ~ IndAncDec{
   override count = 0
}

with(HasIncAndDec()){ inc(), inc(), dec(), count, countdown }
//returns: (1, 2, 1, 2, -1)
```

Note above that when a field in a trait is overridden in a subclass or trait, it must be declared as overridden using the override keyword (note that when classes `override` superclass fields the use of the override keyword is optional).

20.6 Local class definitions

Local classes may make use of traits in a similar manner to how they can extend superclasses:

```
trait IndAncDec{
   -count int
   -countdown int =0
   def inc() => ++count + countdown
   def dec() => --countdown + count
}

HasIncAndDec = class ~ IndAncDec{
   override count = 0
}

with(HasIncAndDec()){ inc(), inc(), dec(), count, countdown }
//returns: (1, 2, 1, 2, -1)
```

20.7 Anonymous class definitions

Anonymous classes can make use of traits as follows:

```
abstract class Operator{
   def operate(a int) int => a
}

open class ID < Operator{
   override def operate(a int) => a
}

trait PlusOne < Operator{ override def operate(a int) => super.operate(a)+1 }
trait Square  < Operator{ override def operate(a int) => super.operate(a)**2 }
trait MinusOne < Operator{ override def operate(a int) => super.operate(a)-1 }
trait DivTwo  < Operator{ override def operate(a int) => super.operate(a)/2 }

x = new ID ~ PlusOne, Square, MinusOne, DivTwo // (((x/2) - 1) ** 2) + 1
y = new ID ~ DivTwo, MinusOne, Square, PlusOne //reverse operator application
z = new ID ~ DivTwo, MinusOne, Square, PlusOne{
   override def operate(a int) => super.operate(a)+1000
}//reverse operator application with additional operation

inst.operate(10) for inst in [x y z]
//returns: [60, 17, 1017]
```

They can even implicitly be composed of traits as follows:

```
trait IndAncDec{
   -count int
   -countdown int =0
   def inc() => ++count + countdown
   def dec() => --countdown + count
}

xx = new IndAncDec{ //implicit composition
   override count = 0
}
```

```
"" + with(xx ){ inc(), inc(), dec(), count, countdown }

//returns: (1, 2, 1, 2, -1)
```

And, like all anonymous classes, may be used as expressions:

```
trait IndAncDec{
   -count int
   -countdown int =0
   def inc() => ++count + countdown
   def dec() => --countdown + count
}

"" + with(new IndAncDec{
   override count = 0
} ){ inc(), inc(), dec(), count, countdown }

//returns: (1, 2, 1, 2, -1)
```

20.8 Stacking traits

A really useful design pattern afforded by the implementation approach taken to traits in Concurnas is that of the stacking of trait-super method calls. Here is an example with a simple unary operator:

```
abstract class Operator{
   def operate(a int) int
}

open class ID < Operator{
   def operate(a int) => a
}

trait PlusOne  < Operator{ def operate(a int) => super.operate(a+1) }
trait Square   < Operator{ def operate(a int) => super.operate(a**2) }
trait MinusOne < Operator{ def operate(a int) => super.operate(a-1) }
trait DivTwo   < Operator{ def operate(a int) => super.operate(a/2) }
```

We can now create a concrete class which brings together the above traits:

```
class DMSP < ID ~ PlusOne, Square, MinusOne, DivTwo

DMSP().operate(10)
//returns: 17
```

Similarly, if we wish to apply the operations in a different order:

```
class class PSMD < ID ~ DivTwo, MinusOne, Square, PlusOne

PSMD ().operate(10)
//returns: 60
```

The neat thing about stacking traits, as we can see above, is the makeup of our concrete class can be very easily and dramatically changed by simply changing the order of composed traits - in fact in the above example where we have four trait references, this works out as `4! => (4*3*2)=> 24` variants!

20.9 Linearization

The exact mechanism by which method invocation chains are resolved such as in the above example is contingent on an algorithm called Linearization. This is more relevant for complex trait compositions. For example, let's look at a trait/class structure:

```
abstract class Abstract //implicitly extends Object
trait A < Abstract
trait B < Abstract
trait C < B

class Concrete < Abstract ~ A, C
```

From the perspective of our Concrete class, there are two ways to look at this structure. A Conventional inheritance hierarchy and the same from a linearized perspective:

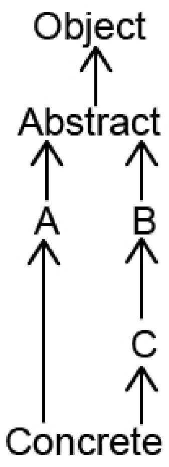


Figure 20.1: Inheritance hierarchy

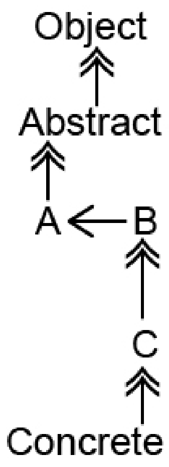


Figure 20.2: Linearization

When we are invoking a method with a super chain (if any) the following, flattened chain, will be checked in order to satisfy an invocation: Concrete, C, B, A, Abstract, Object.

Linearization algorithm. It's not necessary to fully understand the details of the linearization algorithm in order to make use of stacked traits, but the algorithm itself is interesting...

Let's define the Linearization function, acting on a class C as Lin:

$$Lin(C) = C, Lin(C_n) \leftarrow \cdots \leftarrow Lin(C_0)$$

Where $C_0 \ldots C_n$ denotes the superclass and composing traits of class C, defined on a left to right basis. The operator $\leftarrow$ is defined as follows:

$$\begin{aligned}\{a, A\} \leftarrow B &= a, (A \leftarrow B) \text{ if } a \text{ is not inside } B \\ &= A \leftarrow B \text{ if a is inside } B\text{, then } a \text{ is replaced by } B\end{aligned}$$

Let's now look at this for class Concrete as defined above:

$$Lin(Concrete) = Concrete, Lin(C) \leftarrow Lin(A) \leftarrow Lin(Abstract)$$

Let's now recursivity apply the *Lin* operator to each element on the right hand side of the above:

$$\begin{aligned} Lin(Abstract) &= Abstract, Lin(Object) \\ &= Abstract, Object \end{aligned}$$

$$\begin{aligned} Lin(A) &= A, Lin(Abstract) \\ &= A, Abstract, Object \end{aligned}$$

$$\begin{aligned} Lin(C) &= C, Lin(B) \\ &= C, B, Lin(Abstract) \\ &= C, B, Abstract, Object \end{aligned}$$

Putting this all together we arrive at the following:

$$Lin(Concrete) = Concrete, C, B, Abstract, Object \leftarrow A, Abstract, Object \leftarrow Abstract, Object$$

Finally, applying the $\leftarrow$ operator results in the following flattened, linearized, definition for Concrete:

$$Lin(Concrete) = Concrete, C, B, A, Abstract, Object$$

20.10 Using non Concurnas traits/interfaces

Traits may be defined and used within Concurnas to a limited extent. These may originate from other JVM languages such as Java, Kotlin etc. In the case of Java interfaces there are some caveats one needs to bear in mind when using them inside Concurnas:

- They may not have non static state of any kind.
- They cannot be stacked.
- Superclass methods take precedence over trait methods if defined.

21. Special Classes

There exist two forms of special class in Concurnas which can be used in some circumstances where a non standard approach to the sharing of state or persistence is required. These are shared classes and transient classes.

Bear in mind that one may choose to designate as class as being either shared or transient but not both.

21.1 Shared Classes

Classes, abstract classes and traits may be marked shared. This can be achieved by either using the `@com.concurnas.lang.Shared` annotation or shared keyword.

```
shared class MyClass{
   mylistInst = [1, 2, 3, 4]
}

obj1 = MyClass()
obj2 = {obj1}!//obj1 &== obj2
obj3 = obj1@//obj1 &== obj3
```

Marking a class or trait as being shared is most useful for read only or large data structures which are used by multiple iso's, or for classes which are not concerned with or implement their own multi threading (multi iso) support - for instance, all ref's a implicitly shared since their implement their own multi threading support.

Instances of objects the class of which is marked as shared or having a shared superclass or trait cannot be explicitly copied, the following kind of code results in a compilation error:

```
shared class MyClass{
   ~a =9
}
mc = MyClass()
c = mc@//this will result in a compilation error
```

But of course implicit copying is still possible (when an object is passed to an iso or transitivity, e.g. when a field of a class is of a shared class type) - here the 'copy' of the object will in fact just be a reference to itself.

Since marking a superclass, abstract class or trait as being shared will result in all implementing/composed/sub classes also becoming shared, so care should be taken with transient traits.

Classes may be tagged as either shared or transient but not both.

Care should be taken when declaring top level global variables, at module level, with a type of shared class with subsequent reassignment at module level (both directly or indirectly via a function/method etc). Since top level module code is run on import by an isolate, this has the effect of wiping out whatever was previously stored within the variable holding a shared class type every time an isolate which uses any aspect of the module is executed - thus defeating the point of the shared class. Here is an example of what to watch out for:

```
//in module com.myorg.code.conc
shared class Holder<X>(~x X)

public shared sharedvar = new Holder("initial value")//initial top level
    declaration

sharedvar = new Holder("reset value") //top level module core re-assigning a
    value to sharedvar - dangerous

//in module: com.myorg.othercode
from com.myorg.code import sharedvar, Holder

def operation(){
   sharedvar = new Holder("new Value")
   [sharedvar.getX() {sharedvar.x}!]//when the iso is executed sharedvar will be
       'reset' to 26
}

//== ['new Value:' 'reset value:']
```

Removing the `sharedvar = new Holder("reset value")` line will have the effect of allowing us to preserve the assigned value of `Holder("new Value")` within the `operation` method when the iso `{sharedvar.x}` is run:

```
//in module com.myorg.code.conc
shared class Holder<X>(~x X)
public shared sharedvar = new Holder("initial value")

//in module: com.myorg.othercode
from com.myorg.code import sharedvar, Holder

def operation(){
   sharedvar = new Holder("new Value")
   [sharedvar.getX() {sharedvar.x}!]
}

//== ['new Value:' 'new Value:']
```

21.2 Transient Classes

Classes, abstract classes and traits may be marked transient. This can be achieved by either using the `@com.concurnas.lang.Transient` annotation or transient keyword:

```
@com.concurmas.lang.Transient
class MyTransientClass

transient class AnotherTransientclass

transient trait MyTrait
```

Marking a class as being transient will render it uncopyable between iso's. I.e. referencing an object of a class marked as being transient in an iso which has been created outside of that iso's scope will result in a it being null, additionally, manually copying an object using the `@` operator will return null:

```
transient class MyClass

obj = MyClass()
obj2 = {obj}! //obj2 resolves to null
obj3 = obj @//obj3 resolves to null
```

Additionally, marking a class as transient results in it being non persistable off-heap.

Marking a superclass, abstract class or trait as being transient will result in all implementing/-composed/sub classes also becoming transient, so care should be applied with transient traits.

22. Accessibility Modifiers

Concurnas supports four key accessibility modifiers (in order of restrictability): `private`, `package`, `protected` and `public`. These are essential for supporting encapsulation from an object oriented programming perspective. Making diligent use of accessibility modifiers improves code readability (by clearly marking the accessibility of code), reduces maintenance costs of software (e.g. one can change the argument structure of a private function knowing that the only callers of said function are local to the definition), and generally leads to more structured, thought out code being written.

Modifiers can be applied at either the top level module level or more commonly to class elements (fields and methods). In total the following 8 types of referable/importable item can have accessibility modifiers: variables, class fields, functions, class methods, classes, enumerations, typedefs, annotations. The accessibility modifier keyword is placed first in the definition of said element. e.g.

```
//at the module level...
private raiser= 99
public mypow(a int) = a ** raiser

//at the class level...
class MyClass{
   private age int = 20
   public isOldEnough(test int) = test >== age
}
```

In the above example, the private variables and fields are not accessible outside of their defined module and class respectfully. However the publicly accessible function and methods are. `private`, `public` and `protected` are the most commonly used accessibility modifiers. A full table showing which items are assessable from what other parts of the code is as follows:

Accessible from	public	protected	package	private
Same Class/Module	Y	Y	Y	Y
Subclass	Y	Y	N	N
Package	Y	Y	Y	N
Global	Y	N	N	N

22.1 Default Accessibility

All of the aforementioned accessibility modifiable elements have a default accessibility applied where no accessibility modifier is specified. For example in the below case:

```
class MyClass{
   age int = 20
   isOldEnough(test int) = test >== age
}
```

The `age` field has a default accessibility applied of private and the `isOldEnough` method has a default accessibility applied of public. A table showing the default accessibility for each of the items, contingent on their defined context is as follows:

Item	Default
variable	private
field	private
function	public
method	public
class	public
nested class	protected
enumeration	public
annotation	public
typedef	public

23. Generics

Generics are an extremely useful part of modern object oriented programming. They allows us to create 'generic' data structures and work with data in a 'generic' fashion, that is to say, they allow us to write code (only once) to work with different classes of instance objects in a type safe manner. Let's look at an example from the pre generic programming days:

```
class Holder(~holds Object)
```

Above we have defined a `Holder` class which holds just one value. So that we can hold instance objects of any type of class, we declared the `holds` field (accessible via an auto generated getter and setter) to be of type Object. Now let's use it:

```
holder = new Holder("hi there")
got String = holder.getHolds() as String
```

Because `getHolds` returns a type Object, we must always cast the return value back to the type which we stored in the Holder. Note that we must remember what the type is, this is not automatic, this code is not type safe. The following would be perfectly valid code at compile time, but would throw a case exception at runtime:

```
holder = new Holder("hi there")

holder.setHolds(new Integer(123))//this is ok, Integer is a subtype of Object

got String = holder.getHolds() as String //this will throw a cast exception at
    runtime because the held value is an integer and not a String thanks to the
    previous setter call
```

Surely, there is a better way to do this. Enter generic programming which provides a typesafe mechanism to achieve this.

First we redefine our Holder class with a generic parameter `X` which is used in place of all instances of the type of our generic parameter:

```
class Holder<X>(holds X){
   def getHolds() X {
      return holds
   }

   def setHolds(newHolds X) void {
      this.holds = newHolds
   }
}
```

When defining generic types the diamonds <> are used and a comma separated list of generic type parameters are specified.

When specifying generic types it has become customary to use a single uppercase letter to denote the generic type. Not that this is not a requirement, the name used can any valid identifier - but watch out for clashes.

We shall now use the holder in a type safe manner:

```
holder = new Holder<String>("hi there")

holder.setHolds(123)//this now results in a compile time error!

held String = holder.getHolds() //no cast required

holder.setHolds("new value") //this is ok
```

Above we can see that we are creating a generic instance of `Holder` with the `new` operator by qualifying the generic parameters as a comma separated list inside a diamond pair <>.

We can use any object type and even primitive types as qualifiers for generic types. For instance the following are all valid variable declarations of the generic type `Holder`:

```
class Holder<X>(holds X)

inst1 Holder<String>
inst2 Holder<Object>
inst3 Holder<Integer>
inst4 Holder<int>
inst5 Holder<java.util.ArrayList<String>>
```

If a class has generic parameters, these must be defined when creating instance objects of them (unless the qualification is inferable, see below) and when referring the type. So for example, the below is not valid:

```
class Holder2<X>{
   ~holds X
}

aholder Holder2 //invalid as we are missing a generic type qualification
another = new Holder2()//also invalid as the generic type qualification is
    missing and it's not infereable from the constructor (since there are no
    arguments referencing the generic type)
```

23.1 Generic qualifications for super classes

Where a super class specifies generic types these can be qualified by using the diamond symbols <> with either another generic type or a concrete type. For example, the following are all valid:

```
open class SuperClass<X, Y>

class Child1 < SuperClass<String, Integer>//supertype qualified with concrete
    types

class Child2<A, B, C> < SuperClass<String, Integer>//generic types defined for
    subclass, unused as far as the superclass qualification is concerned

class Child3<A> < SuperClass<A, Integer>//we use a generic type from the
    definition of the sub class.

class Child4<A, B> < SuperClass<B, A>//all superclass generic parameters are
    taken from the subclass

class Child5<A, B> < SuperClass<java.util.HashSet<B>, java.util.HashMap<B,
    A>>//Here we are using the generic classes from the subclass within concrete
    classes qualifying the superclass generic types

typedef MyMap<X> = java.util.HashMap<String, X>
class Child6<A, B> < SuperClass<int, MyMap<A>>//now we are referring to a
    typedef. This code is equivalent to writing: class Child6<A, B> <
    SuperClass<int, java.util.HashMap<String, A>>
```

23.2 Inference of generic types

Concurnas is able to infer the generic type qualification of a generic type parameter in a generic class via two mechanisms. Constructor argument qualification and Via usage qualification:

23.2.1 Constructor argument qualification

The generic types of an instance of a generic class type can be inferred provided that they are referenced in a constructor creating an instance of the generic class type by examining the type of the arguments passed to that constructor. For example:

```
class Holder<X>(~holds X)

inferred = new Holder("hi there")
```

Above, `inferred` is inferred as being of type Holder<String>.

In the case where there are multiple potential types which could quality the generic type, the most specific one is used. For example:

```
class PairHolder<X>(a X, b X)

infered = new PairHolder(12, 56.22)
```

Above, `inferred` is inferred as being of type `PairHolder<Number>`, as the two arguments satisfying `X` are both subtypes of type `Number`.

23.2.2 Via usage qualification

Concurnas is able to infer generic types based on their localized usage, post construction in a local variable, a top level variable or class field. This is a really powerful mechanism and saves us lots of time when writing code which uses generics, especially when we are not initially calling a constructor which Concurnas can use as par above ("Constructor argument qualification"), in order to infer the constructed object's generic qualification(s). For a variable or field that is missing generic qualification(s) Concurnas is able to infer those generics based on (amongst other things) what methods are called on the variable, what method references are created based on the variable, what values it is later assigned, methods it is passed to as an argument, what generic fields are directly set etc. Here is an example:

```
class Holder<X>{
   ~holds X
}

holder = new Holder()//holder instance is inferred to be of type Holder<String>
    based on usesage below
holder.holds("hi there")
```

Above we see that the `holder` variable, when declared, it omitting its single genetic type qualification of `X`. Concurnas is able to infer the type of this though via the usage of `holder` - the `holds`'s method method is called upon it and this information is used to qualify the generic parameter `X` to `String`.

This is an especially useful feature which allows us to define and use some of the Concurnas built in (and auto included) functions as follows:

```
typedef set = java.util.HashSet

myset = set()//based on the usage below myset is inferred to be
    java.util.HashSet<Integer>

myset.add(1)
myset.add(2)
myset.add(3)
```

We see above that `myset` is inferred to be of generic type `java.util.HashSet<Integer>` based on the fact that calls to the `add` method pass in a value of (boxed object) type `Integer` which qualifies the `java.util.HashSet`'s single declared genetic type.

Partial inference

Concurnas is able to infer the full generic type qualification for multiple generic types in the case where one or more is qualified via a constructor, and one of more is qualified via usage like so:

```
class Both<X, Y>(x X){
   def takesOne(b Y){}
}

both = new Both(12)//based on constructor and usage below Both<X, Y> is
    inferred to: Both<Integer, String>
both.takesOne("ok")
```

Above we see that the generic qualifiers for the `Both` instance is inferable from the constructor call and subsequent usage of the object `bot` to be: `Integer, String`

23.3 Generics for Methods

So far we have seen generics being used in the context of classes to make generic classes. But we can also make generic methods and functions. Here are some examples:

```
class Holder<X>(~held X)

def extractFromHolder<X>(holder Holder<X>) X {
   return holder.getHeld()
}

def swap<X>(h1 Holder<X>, h2 Holder<X>) void {
   tmp = h1.held
   h1.held = h2.held
   h2.held = tmp
}

def duplicator<X>(todup X) X[]{
   return [todup todup]
}
```

We can call the above functions in just the same way as we ordinarily invoke functions:

```
res = duplicator<String>("x")//we must qulify the generic parameter type
//res => ["x" "x"]
```

23.3.1 Inference of Generics for Methods

Concurnas is able to infer the generic types at invocation point for a method having generic types defined. Carrying on from our previous example:

```
res1 = duplicator(12) //allow Concurnas to infer the generic type
//res => [12 12]
```

Nested methods/functions may also have generic types.

23.4 Generic lambdas

Generic Lambdas may be declared and used in the following manner:

```
lam = def <X>(inst X) X[] { [inst inst]}
res = lam(12)
//res == [12 12]
```

Concurnas is able to infer the generic types based upon usage of any objects returned from an object having otherwise unqualified local generic types provided that those types are referenced in the returned object:

```
def meMap<X, Y>() => new java.util.HashMap<X, Y>()

counter = meMap()//created without local generics
counter[0] = "val"
```

Above, the invocation of `meMap` can have its local generic types inferred as `<Integer, String>` based on the usage of the returned type of `java.util.HashMap<X, Y>` assigned to variable `counter`.

23.5 Generic Method References

Method references may be specified with generic type qualifications in a relatively unobtrusive manner. Where generic types are involved in method references they can be either qualified at reference creation time, or at invocation time:

```
def doubler<X>(x X) => [x x]

qualified = doubler<int>&
differed = doubler&

res1 = qualified(12)
res2 = differed('hi')//generic type qualification is infered
}
//res1 => [12 12]
//res2 => ['hi' 'hi']
```

23.6 Generic type restrictions

The following code is not valid:

```
class Holder<X>(~holds X)

myHolder Holder<Object> = new Holder<String>("something") //this is not valid
    code
```

When it comes to generic type qualification, the types must match on a 1:1 basis. Hence only the following is valid:

```
class Holder<X>(~holds X)

myHolder Holder<String> = new Holder<String>("something")
```

In Concurnas, as with most other languages which run on the JVM, generic types are by default, erased at runtime. That is to say that they are not available for reference at runtime. It is for this reason that the following code is not valid:

```
class Holder<X>(~holds X)

inst Object = new Holder<String>("")

check = inst is Holder<String>//we cannot check the generic qualification
```

This also means that we cannot overload methods with a type signature differing in generic type qualification only:

```
def aFunction(op1 java.util.ArrayList<String>){
   //do something
}

def aFunction(op1 java.util.ArrayList<Integer>){//this won't work!
   //do something else
}
```

The above will fail as the two functions are treated as: `def aFunction(op1 java.util.ArrayList){}` `def aFunction(op1 java.util.ArrayList){}` which is the same.

However, none of these disadvantages are major and there are some workarounds which we can use to make generic types even more useful. We shall explore some of these next...

23.7 Bounded generics

Concurnas permits generic type definitions to be bounded such that all qualifications of generic types must be equal to or a subtype of the specified bound. This applies to both class level and method level generic definitions. In fact all generic type definitions in Concurnas are bounded implicitly to Object by default.

This is a useful technique for cases where one wants to have some additional control over the types which are applicable for use within a class or method using a generic type. When defining a generic type one we need to add the type after the parameter name, For example:

```
class MyClass<T Number>(t T){
   def doPlus(arg T) => t.doubleValue() + arg.doubleValue()
}

mc = new MyClass(12.)
result = mc.doPLus(13)

//result == 25.
```

The above code is possible even though the `T` type is generic since it is bounded by Number, thus we know that instances of T must be a subtype of Number.

23.8 In out Generics

Generic types on the JVM are invariant, that is to say, `list<Object>` is not a supertype of `list<String>`. This can be a problem if we want to write code like the following:

```
strList = list<String>()
objList list<Object> = strList//this is not permitted
```

We cannot assign a variable declared as type `list<Object>` a value of type `list<String>`[1] as it would cause us problems later on when using said lists:

```
objs.add(1) //totally acceptable, as Integer is a subtype of Object
s = strList.get(0) //This now fails with a ClassCastException!
```

Above we are adding an `Integer` into an `Object` list which is actually a `String` list. When we come to obtain said value from the list, when it's referenced as a String list we end up with a `ClassCastException` as the previously added `Integer` is in our `String` list (and we expect there to be only objects of type `String`).

We can largely solve this problem though the use of in/out generics. These provide for a facility known as co- (`out`) and contra- (`in`) variance. A neat way to think about this is that `in` generic parameters may only be consumed by an object (used as inputs to methods), whereas `out` generic parameters may only be produced by an object (used in return values from methods).

[1] Of course we can use naughty code like: `objList list<Object> = strList as Object as list<Object>` to achieve this but it would be inadvisable to do so.

23.8.1 Use-site variance

Concurnas supports use site variance, this enables us to qualify individual generic parameters when declaring variables of generic types as being either `in` or `out`. This enables us to write code such as the following:

covariance `in`:

```
strList = list<String>()
objList list<out Object> = strList//objList may not have anyting added to it
```

contravariance `out`:

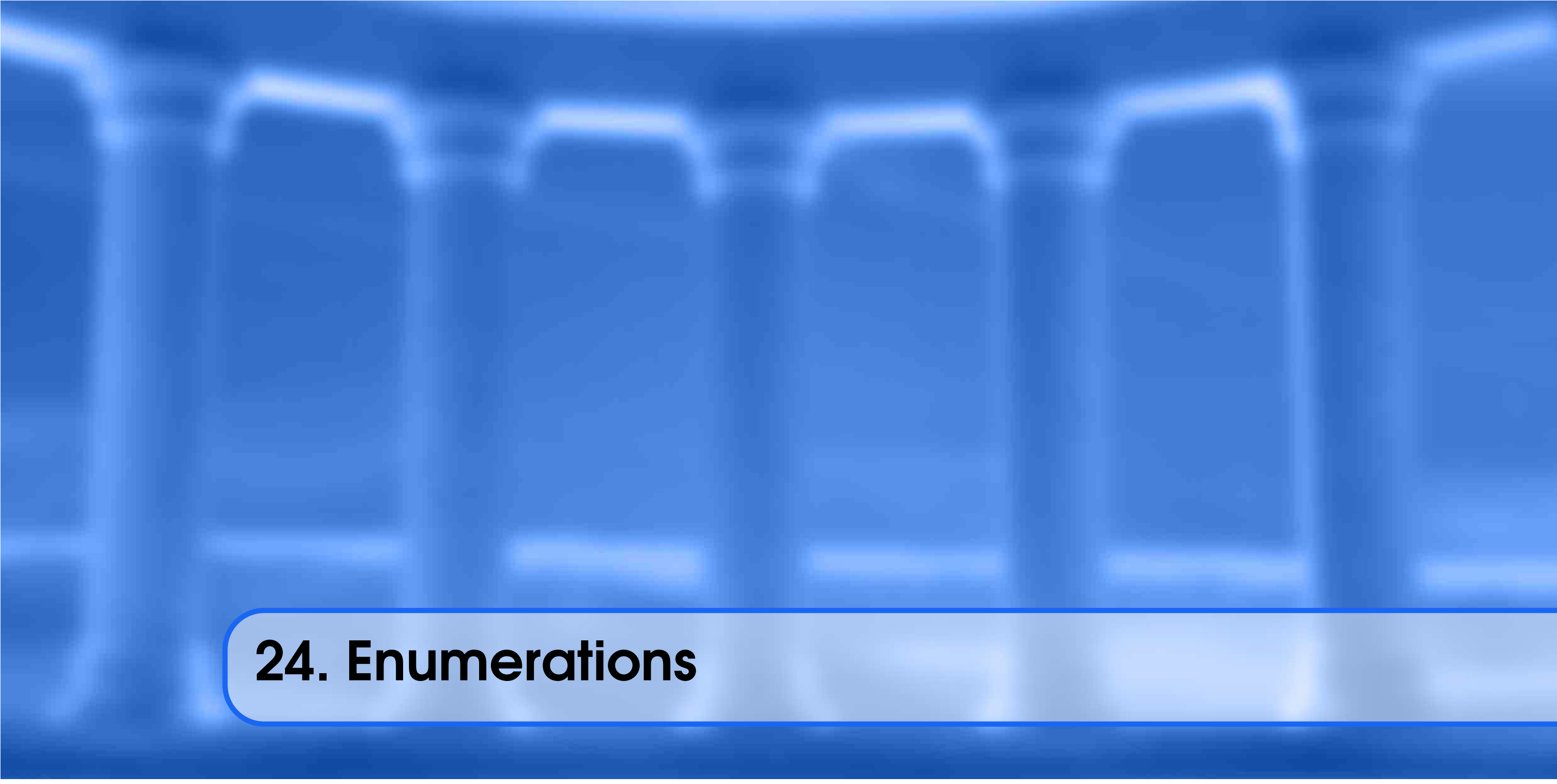
```
strList list<in String> = list<String>()//strList may only have elements added
    to it
objList list<out Object> = strList
```

And so via both these methods we are able to assign our `strList` to our `objList`. We could use this technique to say pass a `list<String>` instance object to a method expecting a `list<out Object>` type. Note that this all comes at the price of limiting what we can do with our generic objects.

23.9 Wildcards

As we have already seen, in Concurnas, where classes or methods require generic types, these must be qualified (or at least be inferable at compile time). However, sometimes if we want to write code which is agnostic to the type of the generic parameter is then using `?` to qualify the type will allow us do this:

```
from java.util import List, ArrayList

def stringMaker(anyItems List<?>) String {
   ret = ""
   for( x Object in anyItems){
      ret += "" + x
   }
   ret
}
```



24. Enumerations

Enumerations, or enum's for short are a handy way of representing a fixed set (at compile time) of related states wrapped up as an object type. All values/states of an enum are considered public. Values may not have accessibility modifiers applied to them.

When declaring an `enum`, a name and a comma separated list of values are required. The values do not have to be capitalized, but it is often conventional to do so. For example:

```
enum Color{ RED, GREEN, BLUE }
```

Enums can be used as follows:

```
aColor Color = Color.RED
anotherColor = Color.BLUE
```

Only one instance of an enum is created per isolate, thus the following holds true:

```
enum Color{ RED, GREEN, BLUE }

c1 = Color.RED
c2 = Color.RED

assert c1 &== c2 //c1 and c2 resolve to the same object
```

In other words, the same enum object is created only once and shared.

Enum items may have state associated with them, this can be initialized as follows:

```
enum Color(hexcode int){
   RED(0xFF0000),
   GREEN(0x00FF00),
   BLUE(0x0000FF)
}
```

We're able to assign state to enums because the individual entries are in fact subclasses of the enum holding them (`Color` in this example). This allows us to write code like this:

```
enum MyEnum(~a int, ~b int){
   ONE(9){
      this(a int){
         super(a,8)
      }
   },
   TWO(22, 33)

   override toString() => "[{a} {b}]"
}


res = [MyEnum.ONE MyEnum.TWO]

//res == "[[9 8] [22 33]]"
```

Given that the enum and enum items are themselves classes, we are afforded access to the likes of abstract methods etc. we're able to write code like the following:

```
public enum Operation {
   PLUS   { public def eval(x double, y double) { x + y; } },
   MINUS  { public def eval(x double, y double) { x - y; } },
   TIMES  { public def eval(x double, y double) { x * y; } },
   DIVIDE { public def eval(x double, y double) { x / y; } }

   def eval(x double, y double) double
}

res = Operation.PLUS.eval(1, 1)

//res == 2
```

Note that although enums can have state, it is not recommended that this state be mutable given that enum items are shared per isolate.

Enums specify two extremely useful methods:

valueOf. This method can be called on an enum type and enables us to find the item instance for a specified name as a String:

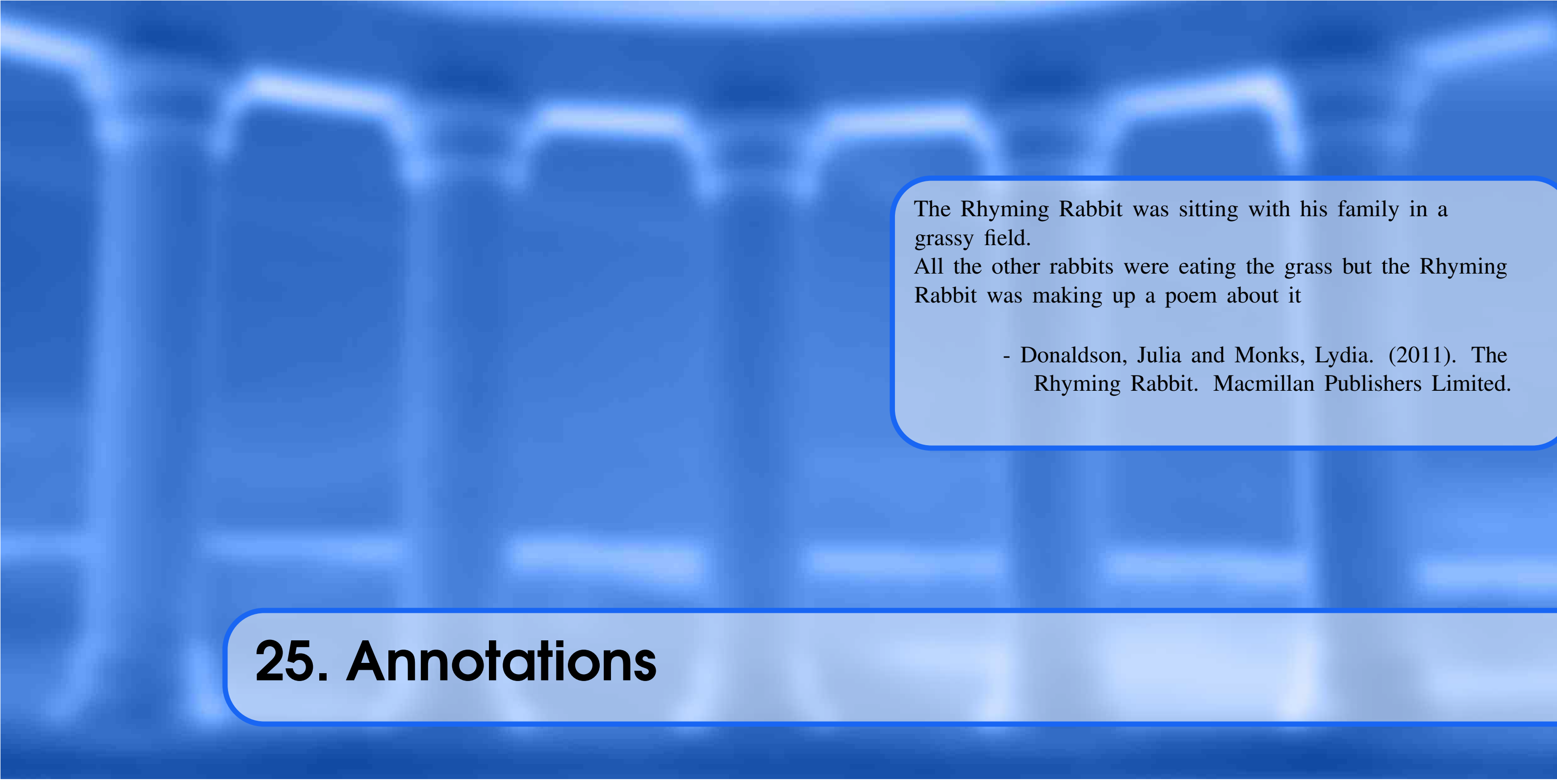
```
enum Nums{ONE, TWO, THREE, FOUR}

item Nums = Nums.valueOf('TWO')
```

A variant of this method is to use the one exposed on the class Enum as follows:

```
from java.lang import Enum

enum Nums{ONE, TWO, THREE, FOUR}

item = Enum.valueOf(Nums.class, 'TWO')
```

values. This method is useful for listing all items of an enum type. For example, we could rewrite our previous example as:

```
enum MyEnum(~a int, ~b int){
   ONE(9){
      this(a int){
         super(a,8)
      }
   },
   TWO(22, 33)

   override toString() => "[{a} {b}]"
}

res = MyEnum.values()
//res == "[[9 8] [22 33]]"
```



The Rhyming Rabbit was sitting with his family in a grassy field.
All the other rabbits were eating the grass but the Rhyming Rabbit was making up a poem about it

- Donaldson, Julia and Monks, Lydia. (2011). The Rhyming Rabbit. Macmillan Publishers Limited.

25. Annotations

Concurnas has support for annotations. They are a handy mechanism by which metadata can be attached to code to be interpreted either at compilation time or at runtime (or both). They are analogous to comments, but designed for our compiler and runtime to understand and act upon.

25.1 Using Annotations

Let's look at a simple example of a built in annotation for reference during and pre compilation and for documentation generation, `@Deprecated`:

This particular annotation is used to denote when a method of a class is deprecated and support for which is likely to be removed in future versions of said class. The annotation may be attached to a method by referencing it before the def keyword as follows. Note that `@Deprecated` it is an auto imported type.

```
class MyClass{
   @Deprecated
   def myMethod(){

   }
}
```

Annotations may be attached to the following items in Concurnas: classes, fields, functions, methods, parameters, annotations. For example:

```
annotation AnAnnotation

@AnAnnotation
def afunction() => 12

@AnAnnotation
class AClass(@AnAnnotation cdf int){
   @AnAnnotation
   def aMethod(@AnAnnotation arg int) => 44

   @AnAnnotation
   afield int = 99
}
```

25.2 Defining Annotations

Annotations may be defined in much the same way as classes, but since they represent static/constant data, they may not express methods and the types which they reference as fields may be of the following types only: primitive types (`int`, `long`, `float` etc), Class, String, enums, annotation or arrays of the aforementioned types. We create an annotation using the `annotation` keyword, the name of the annotation and an optional attached block containing any fields (defined in the same way as class fields):

```
annotation MapsTo{
   name String
   mapTo String
   repeat = 1
}
```

When defining fields we can choose to specify a default value for the field - such as in the case of the `repeat` field. If this is done then users of the annotation do not need to specify a value for `repeat`. On the other hand because the fields `name` and `mapTo` have no default value, they have to be specified by any users of the annotation.

We can use the above declared annotation by using the annotation symbol `@` and specifying any fields as a comma separated list of name and assignment as follows:

```
class MyClass{
   @MapsTo(name = "mappingName", mapTo = "anotherName")
   afield int =99
}
```

As with classes, we can also define annotations using annotation declaration arguments (aka the 'one liner' syntax). For example, the above annotation `MapsTo` can be more concisely declared as:

```
annotation MapsTo(name String, mapTo String, repeat = 1)
```

We can apply more than one annotation at an annotate-able location by separating them with a comma:

```
annotation Annot1
annotation Annot2

@Annot1, @Annot2
class MyClass
```

For annotations which have only one non-default field, we can omit the field name from the parameter declaration when we use it as follows:

```
annotation NameCount(name String, count = 0)
```

```
@NameCount("aName")//we don't have to write this as: @NameCount(name = "aName")
def myFunction() => 16
```

For annotations which have fields of type array, we do not have to specify the array declaration if we only wish to pass an array with a single value. For example:

```
annotation Ports(instances int[])

@Ports(8080) //only one value for field instances. This use call is translated
    into: @Ports(instances=[8080])
class HttpServer{
   a =2
}
```

25.3 Annotation Retention

The retention policy of an annotation denotes how accessible it is to different parts of the compilation/runtime process. There are 3 retention policies dictating how visible an annotation is when it's used:

- SOURCE - The annotation is visible to the reader of the source file only, it's not accessible to the compiler or at runtime.
- CLASS - The annotation is visible to the reader and it is also accessible to the compiler. It's not accessible at runtime.
- RUNTIME - the annotation is visible to the reader, the compiler and is accessible at runtime.

If no retention policy is specified then it will be implicitly defaulted to Runtime. Only one policy may be specified. Most annotations are most sensibly marked with policy of either SOURCE or RUNTIME.

We can specify the retention policy as follows:

```
from java.lang.\annotation import Retention, RetentionPolicy
//note that annotation is a keyword and therefore must be escaped

@Retention(RetentionPolicy.SOURCE)
annotation MYAnnotation1(a=1)

@Retention(RetentionPolicy.CLASS)
annotation MYAnnotation2(a=1)

@Retention(RetentionPolicy.RUNTIME)
annotation MYAnnotation3(a=1)

@MYAnnotation1
@MYAnnotation2
@MYAnnotation3
class SpecialClass
```

25.4 Restricting Annotation Use

By default, a declared annotation can be used at any annotate-able location (classes, fields, functions, methods, parameters, annotations). To restrict this a 'Target' may be specified for the use of the annotation. These are the valid targets and the restrict usage of the declared annotation to:

- ANNOTATION_TYPE - Other annotations
- CONSTRUCTOR - Constructors
- FIELD - Class Fields
- METHOD - Methods or functions
- PARAMETER - Method or function parameters
- TYPE - Classes

Target's can be specified as follows:

```
@Target(ElementType.ANNOTATION_TYPE)
annotation ForAnAnnotation

@Target(ElementType.TYPE)
annotation ForClasses
```

More than one target may be specified by listing the targets as an array:

```
@Target([ElementType.TYPE ElementType.FIELD])
annotation MultiUse
```

25.5 Accessing annotations via reflection

Annotations marked with retention policy RUNTIME (the default if none is specified) may be accessed at runtime via reflection. For example:

```
annotation Ports(instances int[])

@Ports(8080)
class HttpServer{
   a =2
}

res = HttpServer.class.getAnnotations()
//res == [@Ports(instances = [8080])]
```

Many JVM annotation frameworks use this mechanism (reflection) to implement functionality operating on annotations.

25.6 Annotations to class fields with getters and setters

If one is annotating a class field which has getters/setters automatically generated (denoted via use of a -, + or ~ prefix) we can use a special qualified form of the annotation symbol `@` to denote where we would like our annotation to be placed (either on the constructor parameter, the auto generated getter, setter or field itself or a multiple number of these locations). To use the qualified annotation symbol we use the following syntax `@[X]` where `X` is one or many of a comma separated list of: param, setter, getter or field. For example:

```
annotation MyAnnotation

class MyClass(
@MyAnnotation ~field0 int //no qualifier -> the annotation will be attached to
    the constructor parameter by default
@[param]MyAnnotation ~field1 int
@[setter]MyAnnotation ~field2 int
@[getter]MyAnnotation ~field3 int
@[field]MyAnnotation ~field3 int
@[setter, getter]MyAnnotation ~field3 int //attach the annotation to the setter
    and getter methods
){
   @MyAnnotation ~defaultedField1 = 100 //no qualifier -> the annotation will be
       attached to the field by default
   @[setter, getter]MyAnnotation ~defaultedField2 = 200
}
```

If no qualifier is used then the annotation will be attached in one of two ways:

1. If the location is a parameter as part of a class definition argument list, then the annotation will be attached to the constructor argument only.
2. If the location is a field in a class, then the annotation will be attached to that field definition only.

But the little things they make me so happy
All I want to do is live by the sea
Little things they make me so happy
But it's good it's good it's good to be free

- (It's Good) To Be Free from Definitely Maybe by Oasis (1994).

26. Copying Objects

Concurnas provides a convenient syntax for performing a default deep copy of any Object via the copy operator, the Syntax of which being:

```
copyOperator: expression @ copyDefinition?
```

When no copyDefinition is specified the copy operator will provide a deep copy of the expression on the left hand side. E.g.:

```
class Details(age int=21, firstname String="john", sirname String="doe",
    friendCount int){
   override toString() => "{(age, firstname, sirname, friendCount)}"
}

det = new Details(12, "dave", "whatson", 5)
copy = det@

//copy == 12, dave, whatson, 5
```

Immutable objects, composed of completely immutable objects are not copied since post creation they are unchangeable.

References are not copied since they have their own built mechanism for managing access (from different iso's) - this applies even when an explicit copy operation is performed.

26.1 Copy Definitions

The copy operator provides us with a deep copy of our object structure. Sometimes a deep copy is more than what we need (especially for complex data structures) and it can often be helpful to have control over which fields of an object are copied and how. Copy definitions allow us to control this, the syntax of which is:

```
copyDefinition: '('(cdComponent (','cdComponent)? )? (;
              ('nodefault'|'unchecked')*)? ')'


              cdComponent: fieldName ('='expression?)
                         |  fieldName (','fieldName)*
                        | fieldName @copyDefinition?
                        | 'super'@ copyDefinition?
```

26.1.1 Inclusion list

We can explicitly define which fields of our object we wish to copy. Any fields not included in the list will not be included in the copy and their default values will be used:

```
copy = det@(age)

//copy == 12, john, doe, 0
```

Note above that since we have defined a default value for the `firstname` and `sirname` fields, these default values are used for the copy. Otherwise, as in the case of `friendCount`, the default value for the type is used (`0` for integer).

26.1.2 Exclusion list

We can explicitly define an exclusion list, meaning "copy every field except for those defined in the list". Example:

```
copy = det@(<age>)//do not copy the age field instead populate with default value

//copy == 21, dave, whatson, 5
```

26.1.3 Overriding fields

We can override the value of an object field as follows:

```
copy = det@(age = 300, firstname = "fred")

//copy == 300, fred, whatson, 5
```

26.1.4 Nested copy definitions

We can control the way in which fields which are themselves objects are copied by using a nested copy definition referencing said field:

```
class Link(count int, details Details){
   def toString() => "{(count, details)}"
}
link = new Link(2, det)

lcopy = link@(count, details@(age=18))//override age of details field

//lcopy == 2, 18, dave, whatson, 5
```

26.1.5 Super copy definition

We can control the way in which the superclass of the object is copied by nesting a super copy definition as follows:

```
open class Parent(a int=12, b int=13)
class Child(a int, b int, c int) < Parent(a, b){
   override toString() => "{(super.a, super.b, c)}"
}

cc = new Child(1, 2, 3)
ccopy = cc@(super@(a=89, b), c)

//ccopy == 89, 2, 3
```

26.1.6 Empty Copy

We can choose to not specify any contents to the copy definition. In this instance we are instructing the copy operator to copy the object with all fields being of their default value:

```
copy = det@()

//copy == 21, john, doe, 0
```

26.1.7 Ignoring default values

As we have seen previously, where a default value for a field is defined, this will be used if the field is excluded (either explicitly or implicitly) from the copy operation. To suppress this behaviour and instead use the default value for the type of the field, the `nodefault` keyword can be used:

```
copy = det@(;nodefault)

//copy == 0, null, null, 0
```

26.1.8 Unchecked copies

By default Concurnas will check the copy definition to ensure that the fields referenced exist on object to be copied. In order to suppress this behaviour, the `unchecked` keyword can be applied:

```
asObject Object = det
copy = asObject@(age = 66; unchecked)

//copy == 66, dave, whatson, 5
```

I am not the villain in this story.
I do what I do because there is no choice.

- Samuel Hayden. Doom. Id Software (2016).

27. Object Providers

Concurnas has first class citizen support for dependency injection which we term, Object Providers. Readers familiar with frameworks in other languages such as Spring, Google Guice and Google Dagger will no doubt be sold on the benefits of dependency injection and how they become essential for the structuring of large, or even small to medium sized projects. Let's now look at why dependency injection is so useful...

27.1 The case for Dependency Injection

First let us define what we mean by a dependency. A dependency is considered a unit of related functionality that another unit of functionality in a system relies upon. For example, let's say we have a `MessageProcessor` function, which takes a message, potentially performs some processing before passing it on somewhere else.

One way of writing this `MessageProcessor` would be as follows (of course, in real life this would be much more complex, but this example is to illustrate dependencies and the case for dependency injection):

```
class MessageProcessor {
  public def processMessage(){//processing
    sendMessage(getMessage()) //probably some more complex processing here in
        real life...
  }
  private def getMessage() String {//obtination
    return "A message"
  }
  private def sendMessage(msg String) void {//deliverance
    System.out.println(msg)
  }
}
```

The above serves its purpose from a function perspective. However, there are a number of problems, which would be magnified in a real life situation.

1. **Reasoning**. The Message "Processor" above actually contains both the obtination and deliverance functions, which makes reasoning about the functionality harder than it needs to be.
2. **Testing**. The above is very hard to test, since we have no way of easily mocking up the message obtination and message sending mechanism above we have no way of testing the `processMessage` functionality in isolation. Furthermore, in real systems where there are side effects, these are unavoidably triggered when we attempt to test the message processing functionality.
3. **Reusability**. Perhaps we'd like to reuse of the three elements of functionality (obtination, processing, deliverance) could be reused elsewhere in the our overall system we're likely building, but this is extremely difficult with the above design.

We can rewrite the above example, splitting out the three components of functionality which make up the overall function (obtination, processing and deliverance) as follows:

```
class MessageProcessor(obtainer MessageGetter, sender MessageSender){
   public def processMessage(){//processing
      this.sender.sendMessage(this.obtainer.getMessage())
   }
}

trait MessageGetter {
   public def getMessage() String
}

trait MessageSender{
   public def sendMessage(msg String) void
}

class SimpleMG ~ MessageGetter {
   def getMessage() String => 'A message'
}

class MessagePrinter ~ MessageSender{
   def sendMessage(msg String) void => System.out.println(msg)
}

//to be used as follows:
getter = SimpleMG()
printer = MessagePrinter()
mp = new MessageProcessor(getter, printer )
mp.processMessage()
```

If we take `MessageProcessor` above as being the central component of interest, we can say that the `MessageGetter` and `MessageSender` are dependences of the `MessageProducer`. The above design is nice as it solves all the problems previously identified:

1. **Reasoning**. It's clear what all the above components do. And there is no pollution of concerns, senders send, getters get and processors process.
2. **Testing**. By using traits for our `MessageGetter` and `MessageSender` we can provide mock implementations when we are testing our `MessageProcessor` which allows that testing to take place in isolation, side effect free and with controlled inputs and outputs which we can validate against.

3. **Reusability**. We can easily reuse the `MessageProcessor` functionality above by simply defining different `MessageGetter` and `MessageSender` implementations.

Whilst the above is a nice design approach, one disadvantage is that one has to manually create a lot of objects and do a lot of "wiring"/"plumbing" every time one wishes to create a `MessageProcessor`. This of course decreases our ratio of useful domain specific work to non domain specific work, and in practical programs the number of lines of code responsible for this wiring can number in the thousands... But lucky for us, with Concurnas we can use Object Providers to make this much easier for us, as we can automatically inject these dependencies and skip all the plumbing! Here is what we need to change in order to use this:

```
inject class MessageProcessor(obtainer MessageGetter, sender MessageSender){
   public def processMessage(){//processing
      this.sender.sendMessage(this.obtainer.getMessage())
   }
}

provider MPProvider{
   provide MessageProcessor //provide objects of this type
   MessageGetter => new SimpleMG()//dependency satisfaction for MessageProcessor
   MessageSender => new MessagePrinter()//dependency satisfaction for
       MessageProcessor
}

//to be used as follows:
mpProvider = new MPProvider()
mp = mpProvider.MessageProcessor()
///business as usual...
```

Now not only do we have all the advantages outlined above, but we have eliminated the plumbing which use would have had to have done every time we wish to create a new `MessageProcessor` instance, instead we can simply use an instance of the `MPProvider`.

We will now look in detail at this new object provider mechanism...

Note that there other Dependency injection frameworks which are written in Java and are therefore compatible with Concurnas. Some rely on separate configuration files coded in XML, some rely upon runtime reflection and some avoid this. All of these solutions are however library based. Concurnas on the other hand has dependency injection built in and treated with first class citizen support. This of course means that we are able to perform the plumping associated with dependency injection at compile time, via generated code, which makes for a very efficient runtime implementation. This is particularly handy in cases where one is building large complex systems, creating thousands or even millions of objects (and so requires an efficient dependency injection implementation to create those). An additional benefit of providing first class citizen support is that it makes that it easy to track down how dependencies are being injected at compilation and runtime. With library based solutions relying upon reflection, this can be challenging.

27.2 Injectable Classes

In order to be able to render a class compatible with Object Providers we must tag at most one constructor with the keyword `inject` before the accessibility modifier (`public`, `private`, `protected`, `package`, or nothing - which will default to `public`). This has to be done even if we have a zero argument constructor. For example:

```
class MyClass{
   inject this(proc Processor){
        //...
    }
    //...
}
```

In cases where no constructors are explicitly defined (for instance, when we are defining class definition level arguments), then we can tag the class itself with inject:

```
inject class MyClass(proc Processor)
```

27.3 Injectable elements

In addition to constructors, both fields and methods having injectable arguments can be marked as inject (and again are implicitly marked as being publicly accessible):

```
inject class MessageProcessor{
    inject obtainer MessageGetter
    private sender MessageSender
    inject MSSetter(sender MessageSender){ this.sender = sender }
}
```

In the above case, the `MessageSender` is now considered a dependency since it's an argument of an injectable method and the `MessageGetter` is also a dependency as it's the type of an injectable field.

At first glance it would seem clumsy so as to require dependencies to be explicitly marked with the inject keyword. But it's actually incredibly useful as firstly it gets one thinking early on in the construction of one's software from the perspective of dependency injection and how that software will be tested so as to validate its function, and secondly because it makes the expected dependencies of a class very explicit - thus improving readability for whomever will be using and supporting the software in the future.

27.4 Providers

Now that we have marked our classes as being injectable, and tagged our dependencies as appropriate above (whether they be passed in via constructors, methods or directly as fields), we can now move on to defining the Object Providers themselves.

Object Providers are made up of two components, objects to provide, and dependency qualifiers to satisfy those dependencies of the the objects being provided.

```
provider MPProvider{
    provide MessageProcessor //provide objects of this type
    //dependency qualifiers for MessageProcessor...
    MessageGetter => {
        new SimpleMG()
    }//a block may be used
    MessageSender => new MessagePrinter()//a single line may be used
}
```

Providers may provide many Objects of differing type, but they must provide at least one. Also, only non-array object types may be provided. In the above example we're providing one Object of type `MessageProcessor`. In exampling the dependency tree of `MessageProcessor` we see that it have two injectable dependencies on objects of type `MessageGetter` and `MessageSender`. These are qualified via dependency qualifiers.

Dependency qualifiers are type names which are not prefixed with the keyword `provide` and which use => to resolve to an expression which must return something equal to or a subtype of the dependency type being qualified. In the above example it's `new SimpleMG()` and `new MessagePrinter()` qualifying `MessageGetter` and `MessageSender` respectfully. All declared dependency qualifiers must be used in the dependency tree of the objects being provided.

Note that although it is possible to perform complex computation within the dependency qualifier (as any valid expression or block is permitted), it is inadvisable to do so since then one would be mixing computation with one's dependency injection mechanism and this can make reasoning about system behaviour challenging.

At compilation time, the provider block is transformed into a class with generated code to satisfy the object graphs of the defined providers. In this example the name of the provider is `MPProvider` and so a class of that name is created and can be used just like a normal class. As such all the usual restrictions regarding class names being unique per module etc apply. Note that the class is a subtype of `com.concurnas.lang.ObjectProvider`.

The specified provide instances are exposed in this provider (as a class) in the form of a series of public methods returning an instance of the class being provided. So for the above provider we can obtain a new provided instance of a `MessageProcessor` by using code like the following:

```
mpProvider = new MPProvider()
mp = mpProvider.MessageProcessor()
```

Note that all calls to `MessageProcessor` will by default provide a new instance of the `MessageProcessor`. If we want to provide just one unique instance for all calls, then we can use a scoped provider described below.

We can override the name of the method by prefixing the class name with our choice of name. For example:

```
provider MPProvider{
  provide normalMP MessageProcessor//chance name of method to normalMP
  MessageGetter => new SimpleMG()
  MessageSender => new MessagePrinter()
}

mpProvider = new MPProvider()
mp = mpProvider.normalMP()//method is now called normalMP instead of
    MessageProcessor
```

This extends mechanism extends us a tremendous amount of flexibility in terms of automatically generating the wiring/plumbing code for our object dependency graphs. For instance, when it comes to testing, we need simply define a provider with our mock instances in place of our real dependencies for the functionality we wish to focus our testing on.

27.5 Qualified Providers

It can often be useful to fully qualify a provider and cut out the dependency injection mechanism all together, additionally, since dependency qualifiers themselves cannot be exposed as external methods using a provider can be a nice solution to this. Note that provide instances themselves may be used as dependency qualifiers by the Object Provider if appropriate. For instance we could do the following if needed:

```
provider MPProvider{
   provide MessageProcessor => new MessageProcessor(new SimpleMG(), new
       MessagePrinter())
}
```

Providers may be marked as private in order to suppress public method generation for them (note that the associated method will still be generated, but it will be private).

27.6 Transient dependencies

In the examples previously explored we have seen that the dependencies of `MessageProcessor` have been directly qualified in the object provider. But this can also be achieved on a transient basis, or in other words, indirectly provided that the intermediate classes involved are injectable. For example:

```
trait MessageSender{
   public def sendMessage(msg String) String
}

inject class MessageProcessor(obtainer SimpleMG, sender MessageSender){
   public def processMessage(){//processing
      this.sender.sendMessage(this.obtainer.getMessage())
   }
}

inject class SimpleMG(theMessage String){
   def getMessage() String => theMessage
}

class MessagePrinter ~ MessageSender{
   def sendMessage(msg String) String => msg
}

provider MPProvider{
   provide MessageProcessor
   MessageSender => new MessagePrinter()
   String => 'a message'
}
```

Above we see that there is no dependency qualifier for `SimpleMG`. But this is ok, because `SimpleMG` is itself injectable and all of its dependencies (one String) are fully qualified within the provider.

Another way to think about the dependency injection supported by Concurnas Object Providers is as a forest, the providers being the trunk of the trees, the branches the intermediate injectable classes (and type only dependency qualifiers), and the leaves the fully qualified dependencies.

27.7 Provider specific dependency qualifiers

The dependency qualifiers we've seen in the previous examples have all been 'global' qualifiers. Meaning that they apply for all providers and to satisfy all dependencies of those respective provider graphs within the Object provider. If we want to be more specific and define dependency qualifiers which are for use only by only one provider we can do so by specifying some or all of its dependencies in a block as follows:

```
provider MPProvider{
   provide normalMP MessageProcessor{
      MessageGetter => new SimpleMG()
      MessageSender => new MessagePrinter()
   }
}

mpProvider = new MPProvider()
mp = mpProvider.normalMP()//method is now called normalMP instead of
    MessageProcessor
```

This block may contain only dependency qualifiers or type only dependency qualifiers, not provide instances.

27.8 Named dependency qualifiers

Dependency qualifiers may specify a parameter name string to which they will bind their dependencies. This further specializes what dependency they qualify. This is particularly useful in instances where we need to qualify a dependency of the same type but used for different purposes. The named qualifier is defined as follows:

```
inject class User(firstName String, sirName String)

provider UserProvider{
   provide User
   'firstName' String => "freddie"
   'sirName' String => "Brown"
}
```

In the above example when User is provided, `firstName` is mapped to the qualified String resolving to `"freddie"` and `sirName` to `"Brown"`. Note that the named dependency maps to the argument name of the injected constructor, the same applies to injectable method arguments. In the case of fields the field name is used.

This behaviour of mapping the dependency qualifier name to an argument, can be overridden by using the `@Named` annotation (which is an auto import in Concurnas) on the field or constructor/method argument name. For example:

```
inject class User(@Named('The first name') firstName String, sirName String)

provider UserProvider{
   provide User
   'The first name' String => "freddie"
   'sirName' String => "Brown"
}
```

Providers being used to satisfy dependencies may also specify a qualification String as follows:

```
provider MCProvider{
   provide 'aString' String => "A String"
}
```

27.9 Type only dependency qualifiers

In some cases, instead of qualifying a dependency using a dependency qualifier, it can be preferable to direct the dependency to a subtype of that needing qualification. This is particularly the case if one has a trait type which needs qualifying and where there are [potentially] multiple different implementing that trait which would be suitable and which themselves support injection. Let's look at an example of this:

```
trait MessageGetter {
   public def getMessage() String
}

trait MessageSender{
   public def sendMessage(msg String) String
}

inject class MessageProcessor(obtainer MessageGetter, sender MessageSender){
   public def processMessage(){//processing
      this.sender.sendMessage(this.obtainer.getMessage())
   }
}

inject class SimpleMG(theMessage String) ~ MessageGetter {
   def getMessage() String => theMessage
}

class MessagePrinter ~ MessageSender{
   def sendMessage(msg String) String => msg
}

provider MPProvider{
   provide MessageProcessor
   MessageSender => new MessagePrinter()
   MessageGetter <= SimpleMG//type only dependency qualification
   'theMessage' String => 'a message'
}

//to be used as:
mpProvider = new MPProvider()
mp = mpProvider.MessageProcessor()
```

We see above that the `MessageProcessor` is injected with an instance of a `MessageGetter` trait. We've made the `SimpleMG` class injectable and qualified its only dependency (argument name `theMessage` of type String) is qualified to a String. The Object Provider knows which type to qualify the `MessageGetter` with since we provide a type only dependency qualifier linking this as: `MessageGetter <= SimpleMG`. Type specific dependency qualifiers are of the form: `type <= type`. The type on the right hand side of the definition must be equal to or a subtype of the left hand side type.

Type only dependency qualifiers may have their own specific dependency qualifier blocks just like provider declarations. For example:

```
trait MessageGetter {
   public def getMessage() String
}
```

```
trait MessageSender{
   public def sendMessage(msg String) String
}

inject class MessageProcessor(obtainer MessageGetter, sender MessageSender){
   public def processMessage(){//processing
      this.sender.sendMessage(this.obtainer.getMessage())
   }
}

inject class SimpleMG(theMessage String) ~ MessageGetter {
   def getMessage() String => theMessage
}

class MessagePrinter ~ MessageSender{
   def sendMessage(msg String) String => msg
}

provider MPProvider{
   provide MessageProcessor
   MessageSender => new MessagePrinter()
   MessageGetter <= SimpleMG{
      'theMessage' String => 'a message'
   }
}

mpProvider = new MPProvider()
mp = mpProvider.MessageProcessor()
```

27.10 Object Provider arguments

Object providers behave a lot like normal classes. As such we are able to provide arguments to them at the point of creation, these arguments can be used within the individual dependency qualifiers or any fully qualified provider. For example:

```
provider MPProvider(theMessage String){
   provide MessageProcessor
   MessageSender => new MessagePrinter()
   MessageGetter <= SimpleMG
   SimpleMG => new SimpleMG(theMessage)
}

//used as:
mpp = new MPProvider("My message")
mpp.MessageProcessor()
//as normal...
```

27.11 Scoped providers

Concurnas provides two mechanisms where by objects can be scoped, via the `single` and `shared` keywords. Using either of these keywords will result in a singular instance of an object being provided or injected as a dependency by an object provider. These keywords can be used as both dependency qualifiers and provide instances.

How the scopes differ is in terms of the 'lifetime' of the identicality of objects provided. For cases where the `single` keyword is used, all calls to the provider will resolve to the same provided/injected object, throughout the lifetime of the provider itself. For the `shared` keyword, the same object will be provided/injected for the duration of the external call to the provider only - i.e. the object graph will be populated with the same instance of an object for that call only.

Provider specific dependency qualifiers may be scoped, that is to say, the `single` and `shared` keywords may be used within Provider specific dependency qualifier blocks.

27.11.1 single

Where the `single` keyword is used, all calls to the provider will resolve to the same provided/injected object, throughout the lifetime of the provider itself. This can be applied to both dependency qualifiers and provide instances. Simply prefix the entity with the keyword `single`. For example:

```
inject class AgeHolder(age Integer)
inject class User(name String, ah AgeHolder)

provider UserProvider{
   single provide User
   String => "freddie"
   AgeHolder => new AgeHolder(22)
}

up = new UserProvider()
inst1 = up.User()
inst2 = up.User()

assert inst1 &== inst2//true, both User objects are the same
```

The above will resolve true as both variables point to the same object.

We can also apply this to dependency qualifiers as follows:

```
inject class AgeHolder(age Integer)
inject class User(name String, public ah AgeHolder)

provider UserProvider{
   provide User
   String => "freddie"
   single AgeHolder => new AgeHolder(22)
}

up = new UserProvider()
inst1 = up.User()
inst2 = up.User()

assert inst1 &<> inst2 //true, the two User instances are different objects
assert inst1.ah &== inst2.ah//true, the two AgeHolders resolve to the same object
```

Above, the `User` objects returned from the provider above are unique, but their dependant `AgeHolder` instance objects are the same across both instances.

We can apply the `single` keyword to a dependency even without a qualification on the right hand side as follows:

```
inject class Bean{
   count = 0
   def increment() void => count++
}

inject class BeanCounter(-red Bean, -blue Bean)

provider CounterProvider{
   provide BeanCounter
   single Bean
}

bcProvider = new CounterProvider()
bcInst1 = bcProvider.BeanCounter()
bcInst2 = bcProvider.BeanCounter()

assert bcInst1.red &== bcInst1.blue //both Bean instances of BeanCounter are the
    same
assert bcInst2.red &== bcInst1.red //all Bean instances of BeanCounter are the
    same across all instances
```

27.11.2 shared

Where the `shared` keyword is used, the same object will be provided/injected for the duration of the external call to the provider. In other words, all instances of the object in the object graph returned from the provider will be identical. However, unlike the `single` keyword, subsequent calls to the provider will provide a different object. As with the `single` keyword, `shared` can be applied to both dependency qualifiers and provide instances. Simply prefix the entity with the keyword `shared`.

For example, a provide expression may be tagged as being shared - this is useful when the provide expression itself is called by another provide expression in the provider:

```
inject class Bean{
   count = 0
   def increment() void => count++
}

inject class BeanCounter(-red Bean, -blue Bean)

provider CounterProvider{
   provide BeanCounter
   shared Bean => new Bean()
}

bcProvider = new CounterProvider()
bcInst1 = bcProvider.BeanCounter()
bcInst2 = bcProvider.BeanCounter()

assert bcInst1.red &== bcInst1.blue //same bean for single object
assert bcInst2.red &<> bcInst1.red //the two beans on separate invocations of
    the provider differ
```

Above we see that a single instance of the `BeanCounter` class has the same `Bean` instance objects, but different `BeanCounter` instance objects have different `Bean` instance objects (if we were using the `single` keyword then all the `Bean` instance objects would be the same).

We can apply the `shared` keyword to a dependency qualification as follows:

```
inject class Bean{
   count = 0
   def increment() void => count++
}

inject class BeanCounter(-red Bean, -blue Bean)

inject class PairOfBeans(-left BeanCounter, -right BeanCounter)

provider CounterProvider{
   provide PairOfBeans
   single Bean => new Bean()
}

bcProvider = new CounterProvider()
bcInst1 = bcProvider.BeanCounter()
bcInst2 = bcProvider.BeanCounter()

assert bcInst1.red &== bcInst1.blue //resolves to true
assert bcInst2.red &<> bcInst1.red //resolves to true, the two beans on seperate
    invokations of the provider differ
```

The effect is the same as our previous example.

We can apply the `shared` keyword to a dependency qualification without a right hand side qualification as follows:

```
inject class Bean{
   count = 0
   def increment() void => count++
}

inject class BeanCounter(-red Bean, -blue Bean)

inject class PairOfBeans(-left BeanCounter, -right BeanCounter)

provider CounterProvider{
   provide PairOfBeans
   single 'red' Bean
}

bcProvider = new CounterProvider()
bcInst1 = bcProvider.BeanCounter()
bcInst2 = bcProvider.BeanCounter()

assert bcInst1.red &== bcInst1.blue //resolves to true
assert bcInst2.red &<> bcInst1.red //resolves to true, the two beans on seperate
    invokations of the provider differ
```

Again, the effect is the same effect as the previous two examples.

27.12 Providers for Actors and refs

Actor instances and refs, being object types, can be provided by Object Providers, both as provided instances and dependencies. Additionally, for refs which themselves are injectable, their dependencies can also be satisfied by a provider.

27.13 Generics

Generics may be used within Object Providers in any place where you would normally use generics in relation to the use of types. For example:

```
class GenericHolder<X>(xxx X)

inject class MyClass<X>(gh GenericHolder<X>)

inject class MyGenericThing<X>(an X)

provider ManyProvides<X>(item X){
   provide java.util.ArrayList<X> => new java.util.ArrayList<X>()
   provide java.util.Set<String> => new java.util.Set<String>()
   provide MyClass<X>{
      GenericHolder<X> => new GenericHolder<X>(item)
   }
   provide MyGenericThing<String>{
      String => "a string"
   }
}
```

Localized generics are also permitted by postfixing the provide keyword with the list of generic types:

```
provider ARProvider{
   provide<Y> java.util.ArrayList<Y> => new java.util.ArrayList<Y>()
}
```

Generic types may be qualified with `in out` and upper bounds etc as normal.

27.14 Special Types

There are three classes for which special behaviour/support is provided:

27.14.1 Lazy variables

Lazy variables which are dependencies can be assigned to lazily in Object providers, therefore maintaining their lazy binding quality. Here is an example:

```
avar = 88

class MyClass{
   inject this(){}
   inject lazy an String
   override toString() => "" + [avar an avar]
}

provider MCPRovider{
   provide MyClass
   lazy String => {avar = 99; "ok"}
}
```

```
apu1 = MCPRovider()
res = "" + apu1.MyClass()
}

//res == [88 "ok" 99]
```

It's not necessary to explicitly define the String above as being lazy. As the lazy type can be considered a transient dependency. As such we can simplify our provider as follows:

```
provider MCPRovider{
   provide MyClass
   String => {avar = 99; "ok"}
}
```

Note that if the lazy String dependency qualifier above was marked as `single` then the lazy String dependency would be qualified with only one lazy String instance, and the code in the associated block executed only once upon first unassignment of the lazy variable.

27.14.2 Provider type

The provider type is handy if you wish to produce more than one instance of an object instead of just one injected. The `com.concurnas.lang.types.Provider<X>` type takes a function reference or lambda as its single input and invokes this on every call to its `get() X` method. For example:

```
cnt = 0

typedef Provider<X> = com.concurnas.lang.types.Provider<X>

inject class MyClass(an Provider<String>){
   inject an2 Provider<String>
   private an3 Provider<String>

   inject def SetThingamy(an3 Provider<String>){
      this.an3 = an3
   }

   override toString() => "" + [cnt an.get() an2.get() an3.get() cnt]
}

provider MCPRovider{
   provide MyClass
   Provider<String> => new Provider<String>(def () {cnt++; "ok"} )
}

apu1 = MCPRovider()
res = apu1.MyClass() + ""

//res == [0 ok ok ok 3]
```

Again, just like with lazy variables, the provider generic type qualification only need be specified:

```
provider MCPRovider{
   provide MyClass
   String => {cnt++; "ok"}
}
```

Also, as with lazy types if the qualifier is marked as being `single` then only one instance of the qualifier block will ever be executed. No matter how many times get is called.

27.14.3 Optional type

If you are running Concurnas on an Oracle JVM greater than or equal to version 1.8 then the `java.util.Optional<X>` type (https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) may be used to denote an object which will either contain an instance of type `X` or `null`. Dependencies which are of Optional type may have their dependencies omitted in Object Providers. Here is an example:

```
from java.util import Optional

inject class MyClass(an Optional<String>){
   override toString() => "" +an.isPresent()
}

provider MCPRovider{
   provide present MyClass{
      String => "hi"
   }
   provide notPresent MyClass
}

apu1 = MCPRovider()
inst1 = apu1.present()
inst2 = apu1.notPresent()

res = ""+ [inst1 inst2]

//res == [true false]
```

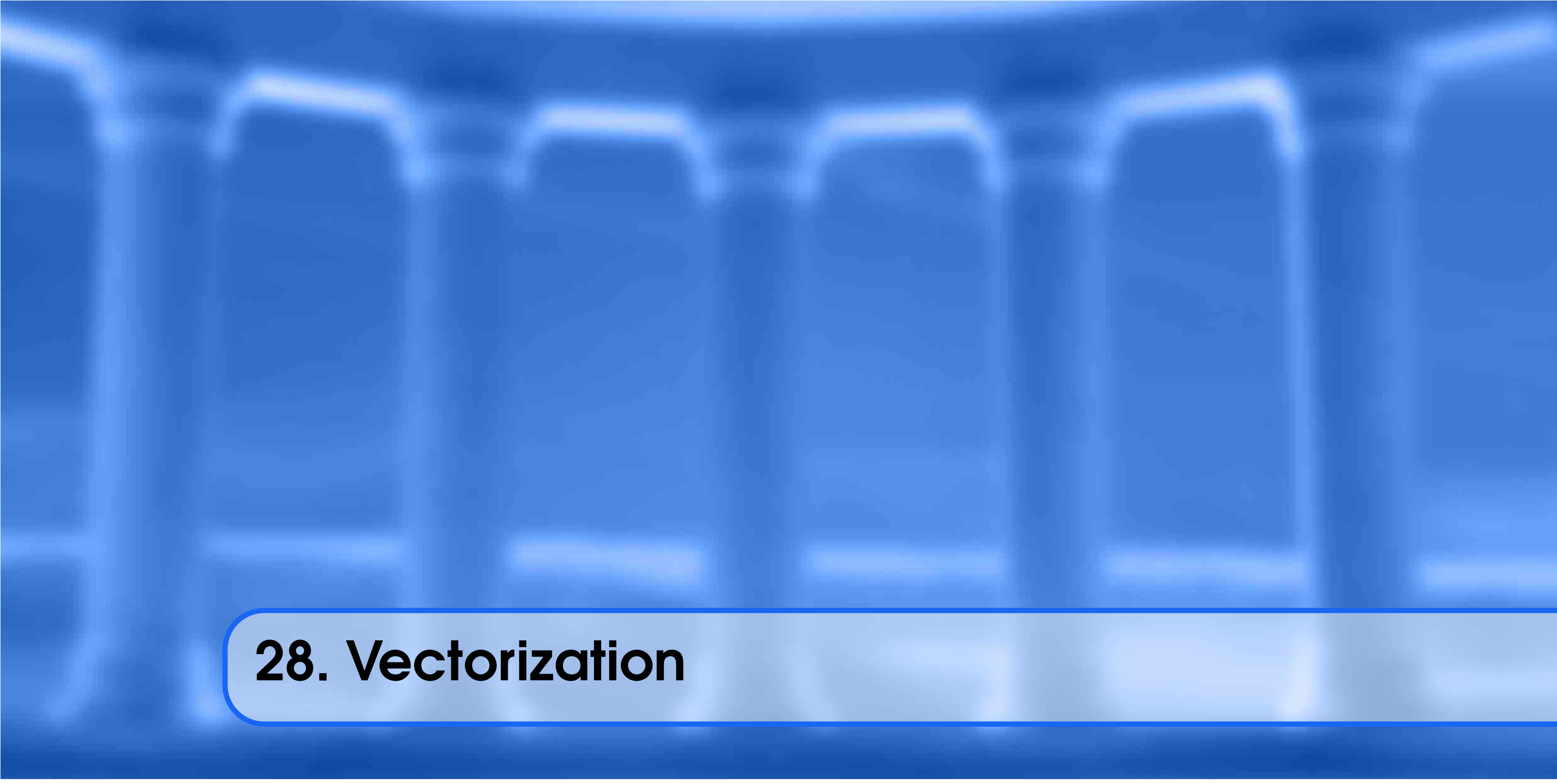
27.15 Object Providers with Java Classes

If you have Java classes or classes in a standard Java format produced by another JVM compatible language (such as Clojure, Scala or Kotlin) then these can be made compatible with Object Providers by ensuring the following: At least one public constructor is decorated with the `com.concurnas.lang.Inject` Annotation. Any public methods with arguments requiring injection or public fields needing injection are marked with the `com.concurnas.lang.Inject` Annotation.

In order to use Named dependencies, the injected arguments/fields must be decorated with either the `@Named` or `@FuncParam` annotation.

III

Advanced Techniques



28. Vectorization

Concurnas contains special support for performing what are known as vectorized operations. This is incredibly useful in cases where one wishes to apply an operation (see the Operators chapter) or function to each of the elements of an array, matrix or other n-dimensional array or list, without having to change the function or apply the operator explicitly to each element.

Say we wish to apply the sine function to every element of an array, we pass the array into our function as normal, except appended with a hat^:

```
from java.lang.Math import sin//sin expects our single value input to be
    expressed as radians

myarry = [ 1 2 3 4]
appliedsin = sin(myarry^)//applies the sin function to every element of the
    array, returning another array
```

In the above example the result of the vectorized call is of type `double[]`double[], if `myarry` were a list then the result would be of type `List<Double>`.

28.1 In place vectorization

The above will return a new array, if we wish to apply the vectorized operation in place to the passed array we can append a hat hat: `^^` to the call:

```
from java.lang.Math import sin//sin expects our single value input to be
    expressed as radians

myarry = [ 1 2; 3 4]
sin(myarry^^)//applies the sin function to every element of the matrix in place.
    myarry will now contain the result
```

28.2 Element wise method execution

We can call an instance method on every element of an array or list as follows:

```
class MyClass{
   def anOperation() = 'result!'
}

myAr = [MyClass() MyClass()]

res = myAr^anOperation()//apply anOperation to each element of the array

//res == [result! result!]
```

Function references can be created in a similar manner:

```
class MyClass{
   def anOperation() = 'result!'
}

myAr = [MyClass() MyClass()]

funcrefs= myAr^anOperation&()//create a funcref to anOperation
res = x() for x in funcrefs

//res == [result! result!]
```

28.3 Element wise field access and assignment

Object field access can be vectorized as follows:

```
class Myclass(public field int)

A = [Myclass(1) Myclass(2) Myclass(3)]

res =A^field //extract the value of field from each object of array A

//res == [1 2 3]
```

And fields can be assigned via vectorization as well:

```
class Myclass(public field int){
   override toString() => 'Myclass: {field}'
}
A = [Myclass(1) Myclass(2) Myclass(3)]

A^field = 99//set value of each field in object in array A to 99

//A == [Myclass: 1 Myclass: 2 Myclass: 3]
```

Access and assignment to fields can be chained using the vectorization operator ^:

```
class FieldCls(valu int){
   override toString() => 'FieldCls: {valu}'
}

```

```
class Myclass(field FieldCls){
   override toString() => 'Myclass: {field}'
}

A = Myclass( [FieldCls(1) FieldCls(2) ; FieldCls(3) FieldCls(4)]^)

res1 = A^field^valu
A^field^valu = 99

//res1 == [1 2 ; 3 4]
//A   == [Myclass: FieldCls 99 Myclass: FieldCls 99 ; Myclass: FieldCls 99
    Myclass: FieldCls 99]
```

28.4 Chained Vectorization

We can chain together vectorized calls as well:

```
from java.lang.Math import sin, toRadians
mymat = [ 80 170; 260 350]
sins = sin(toRadians((mymat^ + 10) mod 360))
```

For cases where we wish to chain together a vectorized expression and have the result written in place, we may specify a vectorized variable within a nested function invocation as the destination for this in place writing:

```
from java.lang.Math import sin, toRadians
myarry = [ 80 170; 260 350]
funcres = sin(toRadians((myarry^^ + 10).))//The result of the entire vectorized
    expression will be written to myarry
```

Not all expressions referenced in the call chain need to be arrays or vectors. For example:

```
from java.lang.Math import sin, toRadians

def rsin(item double, torad bool) => sin((toRadians(item) if torad else item)

myarry = [ 80 170; 260 350]

def getToRad() = true

rsin(myarry^^, getToRad())//applies the sin function to every element of the
    array in place with radian conversion
```

Note that in the above function invocation, the second expression (`getToRad()`) bound to the torad parameter is executed for each element of myarry.

Our invocation, chained and/or containing vectorized operator calls, may reference more than one array, however both the dimensions and number of elements per dimension should match those of the first referenced array. If subsequent arrays exceed the first referenced array in elements then the excess will not be processed. If subsequent arrays do not contain enough elements then an array access exception may (depending on the first underlying structure) be thrown. A null pointer exception maybe be thrown if null elements are present.

```
toinvert = [-1 -2 -3 4]
choices = [false false true true]
```

```
def inverter(arg int, choice boolean) => -arg if choice else arg

applied = inverter(toinvert^, choices^)//apply inverter function to each element
    and return result to applied
inverter(toinvert^^, choices^)//result stored in place within choices array

//applied and toinvert == [-1 -2 3 -4]
```

28.5 Disambiguation of vectorized calls

In cases where a method/function having vectorized arguments has been overloaded, and where that overloading means that the vectorized version of a function call can match more than one version of the function, differing only in the degree of the vectorized expansion, then the version which expands the vectorized arguments to the smallest degree will be chosen:

```
myarray = [1 2 ; 3 4]

def foo(a int[]) = '2d: ' + a
def foo(a int) = '1d: ' + a

res = foo(myarray^)//maps to foo(a int[])

//res == [2d: [1, 2], 2d: [3, 4]]
```

28.6 Vectorizable expressions

In addition to function and methods being vectorized as we have previously seen. All the operators may be vectorized (see the Operators chapter). Examples:

```
myAr = [1 2 3 4 5]
gt = myAr^ > 5
addOne = myAr^ + 1
power = myAr^ ** 1
altpow = myAr^ * myAr^
toString = myAr^ + '' //toString is a String[]

assortment = ['aString' Integer(1) Float(22) 'anotherString']
whichStr = assortment^ is String //=> [true false false true]

wanted = [ 2 3]
subarray = myAr[wanted^]//=> [3 4]
subranges = myAr[0 ... wanted^]//=> [1 2] [1 2 3]
```

In place assignment can also be vectorized:

```
myAr = [1 2 3 4 5]
myAr^ += 1 //increment by 1 assignment
myAr^++ //add 1 as a postfix increment operation
```

Constructor and actor invocations as well as method references can be vectorized:

```
class IntPair(a int, b int){
    override toString() => "(IntPair: {a} {b})"
}
def inca(a int, b int) => a+b

ar = [1 2 3]

objs = new IntPair(12, ar^)//an array of IntPair's
incrementees = inca&(ar^, _ int)//an array of function refences of type: (int)
    int

res = x(10) for x in incrementees

//objs == [ (IntPair: 12 1) (IntPair: 12 2) (IntPair: 12 3) ]
//res  == [11 12 13 ]
```

Array access and assignment operations can be vectorized:

```
A = [ 1 2 3 ; 4 5 6 ; 7 8 9 ]
B = A@//copy of A
C = A@//copy of A

firstOfEach1 = A^[0]
firstOfEach2 = B[ [ 0 1 2]^, 0]
oneOfEach  = C^[ [ 0 1 2]^ ]

A^[0] = 99
B[ [ 0 1 2]^, 0] = 99
C^[ [ 0 1 2]^ ] = 99

//firstOfEach1 == 1 4 7]
//firstOfEach2 == [1 4 7]
//oneOfEach   == [1 5 9]

//A == [ 99 2 3 ; 99 5 6 ; 99 8 9 ]
//B == [ 99 2 3 ; 99 5 6 ; 99 8 9 ]
//C == [ 99 2 3 ; 4 99 6 ; 7 8 99 ]
```

28.7 No hat needed

Note that it's not always necessary to use the hat ^ notation. A neat time saving feature of Concurnas is its ability to implicitly vectorize call chains if the input array argument and input arguments of function being called suit such a call structure. This means we can write the following:

```
from java.lang.Math import sin
myarry = [ 1 2; 3 4]
appliedsin = sin(myarry)
```

The above call is automatically converted to the form: `appliedsin = sin(myarry^)` as the sin function only takes a scalar input.

In cases where a function or method is overloaded and either a array input (with matching dimensionality) or a scalar input can be matched against an array input in the call, the array input will take precedence. In order to explicitly route the call to the scalar input function (and vectorize the call) one must explicitly use the dot as above. Example:

```
def addone(a int) = a+1 //version 1
def addone(a int[]) = [a 1] //version 2

myar = [1 2 3]

addone(myar) //implicitly will be routed to version 2
addone(myar^) //explicitly routed to version 1
```

Compiler note: Concurnas contains optimizations for vectorized call chains to avoid unnecessary matrix creation and thus save memory and creation time, take for example the following:

```
A = [1 2]
B = [3 4]
C = [5 6]

Result = A + B + C
```

A naive implementation would create a temporary matrix used to hold the calculated value of A+B before performing the +C component. Concurnas avoids the creation of the temporary matrix and just creates one holding the result.

29. Ranges

Concurnas has native support for numerical ranges. For example:

```
range = 0 to 10 //integer range from 1 to 10 inclusive
```

Above, range is now of type `IntSequence`. In Concurnas, sequences implement the `java.util.Iterable` interface, meaning that they are able to be used within for loops and anywhere else where an iterator is appropriate. Let's extract the values of the above range:

```
result = x for x in range

//result == [0, 1, 2, 34, 5,6 7, 8, 9, 10]
//note that the range is inclusive of the start and finishing items specified
```

The above example denotes a integer sequence. A long sequence is created when either of the range bounds specified are of type long:

```
range LongSequence = 0L to 10
```

29.1 Steps

Sequences can be created with specific increments via the step method:

```
stepped = 0 to 10 step 2
result = x for x in stepped

//result == [0, 2, 4, 6, 8, 10]
```

29.2 Decrementing sequences

So far we have only explored ascending sequences, we can create descending sequences by inverting the boundary arguments:

```
descending = 10 to 0 step 2
result = x for x in descending

//result == [10, 8, 6, 4, 2, 0]
```

29.3 Reversed sequences

As an alternative to decrementing sequences, a reversed sequence can be created as follows:

```
norm = 0 to 4
rev = norm reversed

//norm == [0, 1, 2, 3, 4]
//rev == [4, 3, 2, 1, 0]
```

29.4 Infinite sequences

Infinite sequences can be created simply by omitting a to argument:

```
infi = 0 to

//infi => 0, 1, 2, 3, ...
```

And they can be stepped as follows:

```
infi = 0 to step 10

//infi == 0, 10, 20, 30,...
```

Note that adding a step also enables us to create infinitely decreasing sequences:

```
infi = 0 to step -1

//infi == 0, -1, -2, -3,...
```

Infinite sequences cannot be reversed.

29.5 In

Sequences have direct support for the in operator (without requiring calculation of the entire contents of the range). For example:

```
range  = 0 to 5
cont1 = 4 in range //cont1 resolves to true as 4 is in the range
cont2 = 88 not in range//con2 resolves to true as 88 is not in the range
```

29.6 Char, double, float sequences

Concurnas doesn't have direct support for non `int`/`long` sequences, however the effects can be easily achieved. For example, a char sequence:

```
chars = x as char for x in 65 to 70

//chars == [A, B, C, D, E, F]
```

29.7 Under the hood

Ranges are implemented via a clever use of both extension functions,expression lists, operator overloading and auto importing of the relevant extension functions and sequence classes. See: `com/concurnas/lang/ranges.conc` for more details.

30. Datautils

Concurnas includes a number of commonly used data related utility typedefs and functions which can be used to make writing programs using the common data structures: list, map and set easier and more succinct. These are defined in: `com.concurnas.lang.datautils` and **are automatically imported**. Most of these data related utility functions are generic and so we're able to take advantage of Concurnases advanced usage based type inference functionality to enable us to write concise programs where this generic type information can largely be omitted and left to the compiler to infer.

30.1 Lists

The `list` functions and typedefs defined within `datautils` create instances of `java.util.ArrayList`

```
alist list = list()//generic type infered as Integer
alist.add([1 2 3]^)
//alist is equvilent to: [1,2,3]

another = list<String>()//generic type information provided
```

30.2 Sets

The `set` functions and typedefs defined within `datautils` create instances of `java.util.HashSet`

```
aset set = set()//generic type infered as Integer
aset.add([1 2 3]^)

another = set<String>()//generic type information provided
```

30.3 Maps

The map functions and typedefs defined within datautils create instances of java.util.HashMap. The map function may consume a lambda which can be used in order to to populate the map with a default value for missing keys.

```
amap map = map()//generic type infered as String, Integer
amap["auspicious"] = 108
//amap is equvilent to: {"auspicious" -> 108}

another = map<String, Integer>()//generic type information provided

counter = map(a => 0)//with default value mapping
counter[0]++
counter[0]++
counter[10]++

//counter == {0 -> 2, 10 -> 1}
```

30.4 Other utilities

Concurnas offers a few basic utilities for working with collections of data, these basic utilities are often used as the basis of more complex and specific data related algorithms.

30.4.1 Sorted

The sorted function may be used to sort an ordered collection (e.g. a list) in place or as a copy. It can consume an optional comparator - itself a SAM class thus permitting the definition of a lambda in its place:

```
xyz1 = [1, 4, 3, 2, 5, 4, 3, 2]
xyz2 = xyz1@//deep copy of xyz1
xyz3 = xyz1@
xyz4 = xyz1@

xyz1sorted = sorted(xyz1)

sorted(xyz2, inplace=true)

xyz3reversed = sorted(xyz3, java.util.Collections.reverseOrder())
xyz4reversed = sorted(xyz4, (o1 , o2) => o2-o1)
```

30.4.2 Reversed

The reversed function reverses an ordered collection and can act in place or return a copy of the input collection:

```
xyz1 = [1, 4, 3, 2, 5, 4, 3, 2]
xyz2 = xyz1@//deep copy of xyz1

xyz1reversed = reversed(xyz1)
reversed(xyz2, inplace=true)
```

30.4.3 Enumerations

The `enumerate` function adds counter to a `java.lang.Iterable<X>` instance object (including lists, sets, ranges etc) and returns it as a list of tuples. The returned type is: `java.util.List<(Integer, X)`.

```
mylist=[1, 2, 400, 4, 5]

res = enumerate(mylist)
//res == [(0, 1), (1, 2), (2, 400), (3, 4), (4, 5)]
```

30.4.4 Zips

The `zip` function creates a `java.lang.List<(X, ..., X)>` of n dimensional tuples from the provided n `java.lang.List<X>` instances. The i-th tuple contains the i-th element from each of the input arguments. The length of the returned list equals the length of the first input argument list. The `zip` function may consume up to 10 input arguments.

```
x = zip([1, 2, 3, 4], [4, 3, 2, 1])
y = zip([1, 2, 3, 4], [4, 3, 2, 1],[4, 3, 2, 1],[4, 3, 2, 1] )

//x == [(1, 4), (2, 3), (3, 2), (4, 1)]
//y == [(1, 4, 4, 4), (2, 3, 3, 3), (3, 2, 2, 2), (4, 1, 1, 1)]
```

31. Pattern Matching

Concurnas has support for pattern matching. Through the use of pattern matching one can check a value against series of patterns, expressed as `case`'s. The functionality found in Concurnas is similar to, and inspired by, that found in the likes of the purely functional languages such as Haskell . Use of patterns generally results in a considerable saving in terms of quantity of code, and increased readability relative to the next best alternative, which would be in writing very long and terse blocks of if then else statements.

31.1 Syntax

The pattern matching expression has the keyword `match` to which a value is passed at at least one case statement with attached block

```
def matcher(n int){
   result = "unknown"
   match(n){
      1 => result = "one"
      2 => result = "two"
      3 => result = "three"
   }
   result
}

matcher(1) // -> returns "one"
matcher(2) // -> returns "two"
matcher(3) // -> returns "three"
matcher(4) // -> returns "unknown"
```

Optionally, an else statement may be used:

```
def matcher(n int){
   result = "unknown"
   match(n){
      1 => result = "one"
      2 => result = "two"
      3 => result = "three"
      else => result = "got value: {n}"
   }
   result
}

matcher(1) // -> returns "one"
matcher(2) // -> returns "two"
matcher(3) // -> returns "three"
matcher(4) // -> returns "got value 4"
```

The following slightly more verbose syntax, with the full block statement form, is equivalent to the above:

```
def matcher(n int){
   result = "unknown"
   match(n){
      case(1){ result = "one" }
      case(2){ result = "two" }
      case(3){ result = "three" }
      else{ result = "got value: {n}" }
   }
   result
}

matcher(1) // -> returns "one"
matcher(2) // -> returns "two"
matcher(3) // -> returns "three"
matcher(4) // -> returns "got value 4"
```

We shall stick to the abbreviated block statement from now on...

The else statement may also take the form of a catch all case statement:

```
def matcher(n int){
   result = "unknown"
   match(n){
      1 => result = "one"
      2 => result = "two"
      3 => result = "three"
      x => result = "got value: {x}"
   }
   result
}

matcher(1) // -> returns "one"
matcher(2) // -> returns "two"
matcher(3) // -> returns "three"
matcher(4) // -> returns "got value 4"
```

As with all other control flow expression and blocks in Concurnas, they may return values. Note that in this case, where we wish to return a value, an else or catch all case must be provided.

```
def matcher(n int){
   result = match(n){
      1 => "one"
      2 => "two"
      3 => "three"
      x => "got value: {x}"
   }
   result
}

matcher(1) // -> returns "one"
matcher(2) // -> returns "two"
matcher(3) // -> returns "three"
matcher(4) // -> returns "got value 4"
```

31.2 Pattern case on types

We can pattern match against types. The match value is cast to the type it is matched against for the bounds of the attached case block. This is analogous to an `isas` type check

```
def matcher(obj Object){
   result = match(obj){
      String   =>   "a String of length: {a.length();}"
      int      =>"an int"
      Object   => "something else: {obj}" //a catch all
   }
   result
}

matcher(1)     // -> returns "an int"
matcher("string") // -> returns "a String of length: 6"
matcher(23.34f) // -> returns "something else: 23.34f"
```

We can match against multiple types:

```
def matcher(obj Object){
   result = match(obj){
      String or int => "a String or int"
      Object => "something else: {obj}" //a catch all
   }
   result
}

matcher(1)      // -> returns "a String or int"
matcher("string") // -> returns "a String or int"
matcher(23.34f)  // -> returns "something else: 23.34f"
```

Note that within the case block for the individual match instances, the value will automatically be cast to the type of interest, making this sort of code easy to write:

```
class Person(-yearOfBirth int, -name String)

def matcher(an Object){
   match(an){
      Person => "Person. Born: {an.yearOfBirth}"//as is treated cast to Person
      x => "unknown input"
   }
}

matcher(Person(1945, "dave"))
//returns: person born: 1945
```

31.3 Pattern case with variable assignment

We can assign to a case variable. The variable will be scoped to exist within the bounds of the attached case block.

```
def matcher(obj Object){
   match(obj){
      str String => "a String with length: {str.length()}"
      x Object  =>  "something else: {x.getClass()}" //a catch all
   }
}

matcher("string") // -> returns "a String with length: 6"
matcher(23.34f) // -> returns "something else: Float.class"
```

31.4 Pattern case conditions for objects

We can go one step beyond pattern cases on types and examine the contents of those types by expressing match conditions on the fields of the object type being matched, within the type pattern case declaration. These conditions must resolve to type boolean, and act upon accessible fields (either direct or via a getter method) and be separated by commas:

```
class Person(-yearOfBirth int, -name String)

def matcher(an Object){
   match(an){
      person Person(yearOfBirth < 1970) => "Person. Born: {person.yearOfBirth}"
      x => "unknown input"
   }
}

[matcher(Person(1945, "dave")), matcher(Person(1982, "freddie"))]
//returns: [Person. Born: 1945, unknown input]
```

Note that we may choose to omit the 'person' variable declaration above as an will be automatically cast to type Person within the body of the catch block:

```
class Person(-yearOfBirth int, -name String)

def matcher(an Object){
   match(an){
      Person(yearOfBirth < 1970) => "Person. Born: {an.yearOfBirth}"
      x => "unknown input"
   }
}

[matcher(Person(1945, "dave")), matcher(Person(1982, "freddie"))]
```

```
//=> [Person. Born: 1945, unknown input]
```

These types of object field content matches can be applied on a recursive basis to fields of objects as follows:

```
class Favourites(-number int, -word String)

class Person(-yearOfBirth int, -name String, -favs Favourites)

def matcher(an Object){
   match(an){
      person Person(yearOfBirth < 1970, favs(number == 12)) => "Person, born:
          {person.yearOfBirth}. Favourite word: {person.favs.word}"
      x => "unknown input"
   }
}

matcher(Person(1945, "dave", Favourites(12, "Panda")))

//=> Person, born: 1945. Favourite word: Panda
```

31.5 Pattern case conditions

We can apply expressions resulting in boolean results in order to attach conditions to case's. The operator implicitly takes the matched value as input.

```
def matcher(n int){
   match(n){
      <10 =>   "less than 10"
      else => "more than or equal to 10"
   }
}

matcher(2) // -> returns "less than 10"
matcher(99) // -> returns "more than or equal to 10"
```

and and or may also be used:

```
def matcher(n int){
   match(n){
      >5 and <10 =>  "greater than 5 but less than 10"
      <10 =>   "less than 10"
      else => "more than or equal to 10"
   }
}

matcher(2) // -> returns "less than 10"
matcher(8) // -> returns "greater than 5 but less than 10"
matcher(99) // -> returns "more than or equal to 10"
```

The full list of compatible operators which can be used in this manner is as follows:
`and, or, ==, <, <>, &==, &<>, >, >, <==, in, not, not in`
Where an expression element is provided without an open attached operator, the matched value

is compared for equality against it

```
def resolvesTrue() = true
def matcher(n int){
   x=4
   match(n){
      x =>"special value " //implicit equality comparison, equivalent to /n==x/
      >5 and <10=>  "greater than 5 but less than 10"
      <10 =>   "less than 10"
      else => "more than or equal to 10"
   }
}

matcher(4) // -> returns "less than 10"
matcher(2) // -> returns "special value"
matcher(8) // -> returns "greater than 5 but less than 10"
matcher(99) // -> returns "more than or equal to 10"
```

Note that any normal expression element is appropriate for a case pattern, all will be checked for equality

```
def resolvesTrue() = true
def matcher(n int){
   x=4
   y=100
   match(n){
      x if resolvesTrue() else y => "special value " //implicit equality
          comparison, equivalent to /n==(x if resolvesTrue() else y)/
      >5 and <10 =>  "greater than 5 but less than 10"
      <10 =>   "less than 10"
      else => "more than or equal to 10"
   }
}

matcher(4) // -> returns "less than 10"
matcher(2) // -> returns "special value"
matcher(8) // -> returns "greater than 5 but less than 10"
matcher(99) // -> returns "more than or equal to 10"
```

Case patterns are checked in a serial manner. Thus one can expect the following behaviour:

```
def matcher(n int){
   match(n){
      <10 =>   "less than 10" }
      >5 and <10 =>  "greater than 5 but less than 10" }//will never be returned
      else => "more than or equal to 10"
   }
}

matcher(2) // -> returns "less than 10"
matcher(8) // -> returns "less than 10"
matcher(99) // -> returns "more than or equal to 10"
```

We can also make use of normal expressions for our case condition:

```
def matcher(n int){
   match(n){
      n<10 =>   "less than 10"
      else => "more than or equal to 10"
   }
}

matcher(2) // -> returns "less than 10"
matcher(99) // -> returns "more than or equal to 10"
```

31.6 Pattern case conditions with type check

We can apply pattern case conditions to values which have first been type checked by separating the assignment and type check from the conditions with a `;`:

```
def matcher(a Object) {
   match(a){
      int; ==1 or ==2 => "first case"
      else => "other case"
   }
}

matcher(1) // -> returns "first case"
matcher(2) // -> returns "first case"
matcher(3) // -> returns "other case"
```

31.7 Pattern case conditions with type check and assignment

As above if we perform an assignment in addition to a case type check then we can apply pattern conditions as follows

```
def matcher(a Object) {
   match(a){
      n int; n==1 or n==2 => "first case"
      else => "other case"
   }
}

matcher(1) // -> returns "first case"
matcher(2) // -> returns "first case"
matcher(3) // -> returns "other case"
```

Note that the expressions post the `;` must be fully formed expressions, the value being matched will not be checked for equality against the resulting value.

31.8 Pattern case conditions with additional checks

Sometimes it may be necessary to perform additional checks using expressions which we do not want to check against the value being matched. In this case, the `also` keyword can be used:

```
class Grouper(size int, avoid int[]){
collector = java.util.ArrayList<int>()

private result = ""
```

```
   def getResult(){
      if(not collector.isEmpty()){
         result += ""+collector
      }
      result
   }

   def addItem(a int){
      match(a){
         not in avoid also collector.size()>==size {
            //cluster our elements together when we hit the size except for when
                == to something in avoid
            result += ""+collector
            collector.clear()
         }
      }
      collector.add(a)
   }
}


grp = Grouper(3, [4])
for(a in [1,2,3,4,5,6,7,8,9, 10]){
   grp.addItem(a)
}
grp.result //resolves to: [1, 2, 3, 4][5, 6, 7][8, 9, 10]
```

31.9 Bypassing pattern case conditions

If one doesn't want to make use of pattern cases, but still needs to check for conditions on a match value, the also syntax in isolation may be used:

```
def isPalendrome(a String){
   upp = a.toUpperCase()
   upp == ""+StringBuilder(upp).reverse()//reverse used to return
       AbstractStringBuilder
}

def matcher(inp String){
   match(inp){
      also isPalendrome(inp) => "is"
      else => "is not"
   } + " a Palindrome"
}

matcher('ava') //returns 'is a Palindrome'
matcher('dave') //returns 'is not a Palindrome'
```

31.10 Match with variable assignment

Similar to try with resources, we can assign a local variable, scope bound to the attached set of case blocks.

```
def reverse(a String) = new StringBuffer(string).reverse().toString()

def matcher(str String){
   result = match(rev = reverse(str)){
      rev.length() > 3 => result = "input string: '{str}' reversed string:
          '{rev}'"
      else => result = "too short to be reversed!: '{str}'" //a catch all
   }
   result
}

matcher("stressed") // -> returns "input string: 'stressed' reversed string:
    'desserts'"
matcher("an") // -> returns "too short to be reversed!: 'an'"
```

The assignment within the match block may be declared `val` or `var`.

31.11 Match against enum elements

When matching a value known to be of an enum type, it is not necessary to specify the entire enum name qualifier. The short name of the enum elements may be matched against.

```
enum MyEnum{ CASE1, CASE2, CASE3, CASE4, CASE5 }

def matcher(a MyEnum) {
   match(a){
      MyEnum.CASE1 => "case 1"//use the full name of the enum element
      CASE2 => "case 2"//use the short name of the enum element
      CASE3 or MyEnum.CASE4 => "case 3 or 4"//use both the short and long names
      else => "another case: {a} "
   }
}

matcher(MyEnum.CASE1)//returns "case 1"
matcher(MyEnum.CASE2)//returns ""case 2""
matcher(MyEnum.CASE3)//returns "case 3 or 4"
matcher(MyEnum.CASE4)//returns "case 3 or 4"
matcher(MyEnum.CASE5)//returns "another case: CASE5"
```

31.12 Match against tuples

Concurnas has special support for performing matches against tuples.

Firstly if we don't already know our input matched against to be of type tuple we can test it as a tuple type and access the dereferenced inner elements as follows:

```
def matcher(n Object){
   match(n){
      case( (a int, b int, c int) ) { "tuple of 3: {a}, {b}, {c}"}
      case( (a int, b int); a > b ) { 'tuple 2: {a}>{b}'}//with a test
      case( (a int, b int); a < b ) { 'tuple 2: {a}<{b}'}//with a test
      case( (int, int, int, int) ) { "4 item int tuple"}
      case( (int, int, double, int) ) { "4 item int tuple with a double"}
      case( (a , b, c, d, e) ) { "5 item tuple with Object type"}
      x => 'catch all: x'
   }
}
```

If we know the type being matched against is a tuple, then the syntax is more succinct as follows:

```
def matcher(n (int, int)){
   match(n){
      (0, 0) => 'both zero'//test against both elements
      (0, ) => 'first zero'//test only one element
      (, >2) => 'second above 2'//perform a test against second tuple element
      (a, 1) => 'second is 1, first is: {a}'//extract an element
      (a, b); a>b => '{a} > {b}'//extract both elements and perform a test on
          them
      (a, b) => 'all others: {a}, {b}'
   }
}
```

32. Method References

Method references are an extremely useful part of functional programming which are included in Concurnas. They allow one to pass a reference to a method around one's program in the same way that one would pass data via objects or primitive types. Note that in this section we use the terms 'function' and 'method' interchangeably as for the most part function and method references behave identically.

32.1 Basic Method References

We create a function or method reference by using the `&` operator:

```
def myfunction(an int, bn int) => an + bn

funcRef1 (int, int) int = myfunction&(int, int)
```

Above, `funcRef1` is a method reference type `(int, int)int` because we have chosen to not bind either of the two input arguments to the function when making our reference. We can call the method reference, like a normal function:

```
result = funcRef1(1, 2)

//result == 3
```

We can even make method references to method references:

```
frefTofref = funcRef1&(int, int)
```

We can choose to bind any or all of the inputs arguments to the function as follows:

```
partial (int) int = myfunction&(int, 10)
full () int = myfunction&(2, 2)

//now lets use them...
```

```
result = [partial(10), full()]

//result == [20 4]
```

Notice how above the type returned from `&` is contingent on which input arguments have been bound. Bounded input arguments do not show up in the method reference type.

If there exists only one function matching the name of the function we're trying to make a method reference for in scope, and we wish to not bind any input arguments (if there are any), then we can forgo having to specify the types to leave unbound and simply create our method reference as follows:

```
funcRef (int, int) int = myfunction&
```

Sometimes there is ambiguity in terms of type names and variable names. Though this is bad practice one can resolve this ambiguity by using an `?` to indicate that we wish to leave the argument with matching typename unbounded:

```
class MyClass()

def bounce(an MyClass) => an
def bounce(an int) => an

MyClass = 99

ref = bounce(? MyClass)//directed call to first version of function
```

If we hadn't used the `?` above then the variable `MyClass` would have attempted to have been passed to the function reference.

Another neat approach we can take when defining method references (particularly for overloaded method definitions differing only in the number of arguments they have) is to simply use a comma to indicate that we wish a parameter to remain unbounded:

```
def myFunction(a int, b int, c int) => a+b*c

fref (int, int) int = myFunction&(, 45, )

call = fref(1, 3)//equvilent to calling: myFunction(1, 45, 3)
```

32.2 Method references for instance objects

Things become slightly more complex when we are dealing with method references on instance objects. We must decide if we wish to bind the method reference to a specific instance object at the point of definition of the method reference or not. We call these two forms, bounded and unbounded method references. The key difference is that only bounded method references may be invoked. Say we have the following class:

```
class MyClass(cnt int){
   def incMany(bywhat int){
      cnt += bywhat
   }
}
```

Let's create a bounded method reference:

```
instObj = new MyClass(10)
boundedMethodRef = instObj.incMany&
```

We can see above that when creating `boundedMethodRef` we are referencing an instance object `instObj` of `MyClass` - as such the method reference held by variable `boundedMethodRef` is said to be bound to object `instObj`. When we call `boundedMethodRef` it is though we are calling `incMany` on `instObj`.

We can create an unbounded method reference in the following way:

```
methodRef = MyClass.incMany&
```

The above method reference `methodRef` cannot be called by itself as it is not bound to an instance object of type `MyClass`. Attempting to invoke `methodRef` in its unbound state will result in an `com.concurnas.bootstrap.lang.LambdaException` exception being thrown.

In order render `methodRef` callable, it first needs to be transformed into a bounded method reference. This is achieved by calling `bind` on the method reference:

```
instObj = new MyClass(10)
methodRef.bind(instObj)
```

Now we can invoke `methodRef`.

32.3 References to Constructors

In Concurnas, method references are not limited to just methods, but they can be applied to constructors as well. Let's take a class:

```
class MyClass(cnt int){
   this(an int, ab int){
      this(an + ab)
   }
   def incMany(bywhat int){
      cnt += bywhat
   }
   override toString() => "MyClass({cnt})"
}
```

We can create a constructor reference, which looks very much like a method reference, in the normal manner as follows:

```
refToCon = MyClass&(int, int)
instanceObj MyClass = refToCon(12, 13)
```

But, what if we wish to defer the choice of constructor called to the caller of the reference? In this case we can use the following syntax in order to create a constructor reference, with special type: `(*) X` where `X` is the type of the instance object being created. Example:

```
refToCon ( * ) MyClass = MyClass& //this will defer the choice of constructor to
    call until later
"result: " + refToCon(12, 13)//at this point the constructor to call is
    determined
```

This may seem to be of little use, since in the example above one could just call `new MyClass(12, 13)` to have the same effect. But consider the application with locally defined classes - which by nature cannot have instance objects of them created via the new operator outside of their defined scope. We can use this feature of Concurnas to create instance objects of locally defined classes outside of their defined scope:

```
def creator(){
   class MiniClass (a String){
      this(a int) { this(""+a) }
      override toString() => "MiniClass: " + a
   }

   //MiniClass cannot be created outside of the scope of creator, unless we use
       a constructor reference as par below...
   return MiniClass&
}

miniCRef = creator()

istObj = miniCRef('hi')
```

32.4 Lambdas

Lambdas are a nice feature of Concurnas from functional programming which allow us to create functions which do not have identifiers. When a lambda is created its type is that of a Method reference. Lambdas are created in the same way as functions but they have no identifier (no name). For example:

```
plusOne (int) int = def (a int) int { return a + 1 }
```

They can be invoked just like normal method references:

```
res = plusOne(2)

//res == 3
```

We can compact the lambda definitions in the normal manner, the following are all equivalent:

```
plusOne (int) int = def (a int) int { return a + 1 }
plusOne = def (a int){ return a + 1 }
plusOne = def (a int){ a + 1 }
plusOne = def (a int) => a + 1
```

To see how these are useful, lets define our own map function operating on an array of integers:

```
def myMap(opOn int[], func (int) int) =>  func(opOn^)

data = [1 2 3 4]
res = myMap(data, def (a int) => a+1)

//res == [2 3 4 5]
```

32.4.1 Zero argument lambdas

Concurnas has special additional support for zero argument lambdas. An expression which evaluates to the return type of a zero argument lambda may be used in place of a lambda definition. For example, this is perfectly valid code:

```
athing () int = {5**2}

res = [athing(), athing(), athing()]

//res == [25, 25, 25]
```

In the above case the `5**2` expression block will be automatically "upgraded" to a lambda, taking no arguments and returning an `int` so as to match the left hand side assignment type. Note that the expression will be fully evaluated on every call to the lambda. To see this in action see the following example:

```
counter = 0
athing () int = 5**counter++
res = [counter, athing(), athing(), athing(), counter]

//res == [0, 1, 5, 25, 3]
```

Here we see that the counter is incremented on every call.

The above automatic "upgrading" may occur at any point where the zero argument lambda is required. For instance, in a function call:

```
counter = 0
def perform(athing () int) => [counter, athing(), athing(), athing(), counter]

res = perform(5**counter++)

//res == [0, 1, 5, 25, 3]
```

32.5 Anonymous lambdas

Anonymous lambdas provide a convenient shorthand for defining lambdas. The following definitions are functionally identical:

```
mul2v1 = def (a int) int => a*2 //expanded 'normal' lambda definition
mul2v2 (int) int = (a int) => a*2 //compact lambda definition with return type
    inference

mul2v3 (int) int = a => a*2    //fully compact lambda definition with return and
    argument type inference
```

With `mul2v2` we see a more compact form of the same lambda definition as `mul2v1`.

The final definition `mul2v3` is most interesting as only the input variable names to the lambda are defined. Their types, along with the return type, are left to be inferred based on the context in which the lambda is defined, which, in this case is on the right hand side of an assignment statement for a function type taking one integer and returning another.

Another common context in which anonymous lambdas are defined are in arguments to function invocations:

```
class MyNumberHolder(~a int){
   def apply(operation (int) int) => operation(a)//apply takes a lambda and
       applies to to the held value of a
}

mnh = MyNumberHolder(12)
res = mnh.apply(a => a+100)//we define a lambda in compact form

//res == 112
```

Note that we must be able to infer the type of the lambda in order to be able to use the compact form. The following will resolve in a compile time error since we don't know what they type of `mul` is:

```
mul = a => a.operation(5, "n")
```

32.5.1 SAM types

SAM types, or Single Abstract Method types are traits (or interfaces if referencing Java code) which define only one single abstract method. Concurnas performs a neat trick where we can map a lambda we have created, in compact form, to an instance of a SAM type. Some alternative methods to using a lambda are to implement our solution as either an instance object, or an anonymous class, but as you will see, the lambda is the preferred approach for its compactness.

```
trait Operator{//This is a SAM type as there is only one method defined which is
    abstract
   def perform(arg int, arg2 int) int
}

class MyNumberHolder(~a int){
   def apply(b int, operator Operator) => operator.perform(a, b)
}

mnh = MyNumberHolder(12)

res = mnh.apply(50, a, b => a + b)// second parameter is used to generate an
    Operator instance

//res == 62
```

In the above example, an instance of the Operator mixin is generated from the addition lambda defined in order to satisfy the second argument of the apply method. Note that we don't have to use the compact lambda form, the full form is acceptable for this purpose as well.

This makes using the Java sdk stream library possible in Concurnas. For example:

```
mylist = [1, 2, 3, 4, 5, 6, 7, 8]

res = mylist.stream().map(a =>a+10).collect(java.util.stream.Collectors.toList())

//res == [11, 12, 13, 14, 15, 16, 17, 18]
```

Without the compact syntax and SAM type support we would have to write code like the following:

```
mixin Operator{//This is a SAM type as there is only one method defined which is
    abstract
   def perform(arg int, arg2 int) int
}

class MyNumberHolder(~a int){
   def apply(b int, operator Operator) => operator.perform(a, b)
}

mnh = MyNumberHolder(12)

myOperator = class ~ Operator{//define a class implementing the mixin Operator
   def perform(arg int, arg2 int) int = >arg + arg2
}

res = mnh.apply(50, new myOperator())

//res == 62
```

The compact lambda definition is far more convenient than this alternative, and the compact form comes with no performance penalty!

32.5.2 SAM types with a zero argument abstract method

Just as with zero argument lambdas Concurnas have special additional support for SAM types whose single abstract method takes no arguments. An expression which evaluates to the return type of method can be used in place of a lambda definition as above. As such we are able to write code such as the following:

```
trait ExeCounter{//this is a SAM type
   counter int
   def toexe() int//zero arg abstract method
   public def invoke() int[] => [counter++ toexe()]
}

athing ExeCounter = 5
res = [athing(), athing(), athing()]

//res == [[0 5], [1 5], [2 5]]
```

33. Off Heap Memory

Concurnas provides support for managing data off heap. Since Concurnas is an Object oriented garbage collected language, data, in the form of Objects, is managed in a subsection of the RAM of the machine upon which it's operating called the heap. This is generally only a portion of the RAM available to and a fraction of the persistable storage available (SSD's, disk drives etc) to the machine.

33.1 Overview

The off heap memory management functionality provided by Concurnas affords us three key advantages:

- We are able to work with datasets which are significantly larger than what is possible to store on heap. For instance, we may be running our program with a with a 8GB heap, 128GB of RAM and many terabytes of physical disk based storage - with off heap memory management we can reside and work with our data in this RAM and physical disk seamlessly.
- We can perform our own memory management. This is often preferable in cases where we are working with large datasets which are resident in memory for large durations of time and/or have access patterns which we are aware of and in control of ahead of runtime. This frees up our garbage collector to focus on other areas of our program's operation.

This functionality is provided in the form of off heap stores and key value pair maps which can be backed in either RAM or disk. These are intuitive and easy to use data structures and are already a very popular industry approved means of working with large datasets.

33.2 Serialization of objects

All objects in Concurnas are sterilizable to and from binary format - this extends naturally to object graphs (i.e. the total data structure referenced by an object and all its fields, and it's fields fields etc). Concurnas is even able to cater for cycles in serialized object graphs by default. It is through this mechanism of serialization that objects are able to be managed off heap, being marshalled back into an object form when they are required in heap memory, and marshalled into binary form for storage/persistence off heap.

This default serialization scheme is added to all classes executed by Concurnas at runtime.

33.2.1 sizeof

We can make use of the `sizeof` keyword in order to determine the amount of bytes a serialized object graph will consume off heap:

```
anArray = [1 2 3 4]

sz = sizeof anArray //sz == 37
```

This can be useful for when when working with large objects in an environment where we have a limited amount of off heap memory.

33.2.2 Custom Serialization

Sometimes the default serialization mechanism is not adequate. In these instances a custom user defined serialization mechanism may be defined. This may take the form of either implementing the `Serializable` interface or `Externalizable` interface.

Implementing the `Serializable` interface

A class may implement the `java.io.Serializable` interface, in doing so it will use the default serialization strategy employed by Java in order to serialize its graph. For example:

```
class MyClass(firstName String, secondName String) ~ java.io.Serializable
```

Optionally, as par Serializable one may define a pair of `writeObject` and `readObject` methods in order to perform the serialization. For example, on a custom ArrayList:

```
class MyArrayList ~ java.io.Serializable{
   items String[]
   highwatermark = 0

   this(){
      items = new String[10]
   }

   this(startsize int){
      items = new String[startsize]
   }

   def add(what String){
      if(highwatermark >== items.length){
         newitems = new String[Math.ceil(items.length * 1.2)as int]
         System.arraycopy(items, 0, newitems, 0, items.length)
         items = newitems
      }
      items[highwatermark++] = what
   }

   private def substr(){
      items[0 ... highwatermark] if items <> null else 'its null'
   }
   override toString() => "" + substr()

   private def writeObject(s java.io.ObjectOutputStream) void{
      s.defaultWriteObject()
      s.writeInt(highwatermark)
      for (i=0; i<highwatermark; i++) {
         s.writeObject(items[i])
      }
   }

   private def readObject(s java.io.ObjectInputStream) void {
      s.defaultReadObject()
      highwatermark = s.readInt()
      items = new String[highwatermark]

      for (i=0; i<highwatermark; i++) {
         items[i] = s.readObject() as String
      }
   }
}
```

Implementing the `Externalizable` interface

A class may implement the `java.io.Externalizable` interface, in doing so we are obliged to define a pair of `writeExternal(outx java.io.ObjectOutput)` and `readExternal(inx java.io.ObjectInput)` methods which perform our serialization and deserialization. For example, on a custom ArrayList:

```
class MyArrayList ~ java.io.Externalizable{
   items String[]
   highwatermark = 0

   this(){
      items = new String[10]
   }

   this(startsize int){
      items = new String[startsize]
   }

   def add(what String){
      if(highwatermark >== items.length){
         newitems = new String[Math.ceil(items.length * 1.2)as int]
         System.arraycopy(items, 0, newitems, 0, items.length)
         items = newitems
      }
      items[highwatermark++] = what
   }

   private def substr(){
      items[0 ... highwatermark] if items <> null else 'its null'
   }

   override toString() => "" + substr()

   public def writeExternal(outx java.io.ObjectOutput){
      outx.writeInt(highwatermark)
```

```
        for(n=0; n < highwatermark; n++){
            outx.writeUTF(items[n])
        }
    }


    public def readExternal(inx java.io.ObjectInput){
        highwatermark = inx.readInt()
        items = new String[highwatermark]

        for(n=0; n < highwatermark; n++){
            items[n] = inx.readUTF()
        }
    }
}
```

33.2.3 Unserializable Objects

All objects in Concurnas are serializable with exception of:

- Actors.
- Any class marked as transient. See here for more details.

33.3 Transient fields

Classes may have transient fields declared within them. These behave like regular fields except that when serialized via the default strategy provided by Concurnas, or the default `java.io.Serializable` strategy, they are not converted to/from binary format. A field may be declared transient by using the `transient` keyword as follows:

```
class MyClass(transient firstName String, secondName String){
    transient yearOfBirth int
}
```

Upon serialization and deserialization transient fields will not be populated, thus in a deserialized object any non primitive, non array type transient fields will have a default value of `null` attributed to them, and primitive types the equivalent of `0`. It is because of this behaviour that non primitive, non array type transient fields are always nullable.

This can be useful in instances where a local resource is tied up to a Object which needs to be persisted or otherwise managed off heap, for instance a database connection. Note that excessive use of transient fields can be a *code smell* indicating unorthodox design.

33.3.1 Default Transient fields

When it comes to the Serialization of transient fields with default values the behaviour differs contingent upon the variant of serialization used. For example:

```
class MyClass(transient firstName String = "dave", secondName String){
    transient yearOfBirth int = 1970 //transient field with default value
}
```

The fields `firstName` and `yearOfBirth` will be deserialized to their respective default values if either:

- The default serialization strategy offered by Concurnas is used.

- Explicit defaulting of the fields is within the appropriate methods of a class extending either `java.io.Externalizable` or `java.io.Serializable`

33.4 Off Heap Stores

Concurnas offers Off Heap Stores, these may reside either in memory via the `com.concurnas.lang.offheap.storage.O` class or on disk via the `com.concurnas.lang.offheap.storage.OffHeapDisk` class - these are subtypes of the `com.concurnas.lang.offheap.storage.OffHeapPutGettable` class. They allow us to store object graphs off heap and provide us with objects we can use in order to interact with those off heap objects.

33.4.1 Creating Off Heap RAM Stores

In order to create an off heap RAM store we must specify the size of our off heap stores in bytes, for instance, 10MB is:

```
10meg = 10 * (1024**2)
```

This can then be used within our `OffHeapRAM` store, with generic qualification to store an array of Strings as follows:

```
from com.concurnas.lang.offheap.storage import OffHeapRAM
from com.concurnas.lang.offheap import OffHeapObject

msg1 = ["hello" "world"]
msg2 = ["nice" "day"]

10meg = 10 * (1024**2)

offHeapRamStore = new OffHeapRAM<String[]>(10meg)
offHeapRamStore.start()

offHeapObj1 OffHeapObject<String[]> = offHeapRamStore.put(msg1)
offHeapObj2 OffHeapObject<String[]> = offHeapRamStore.put(msg2)

gotMsg1 = offHeapRamStore.get(offHeapObj1)
gotMsg2 = offHeapRamStore.get(offHeapObj2)

offHeapRamStore.close()

assert gotMsg1 == msg1
assert gotMsg2 == msg2//equal by value

assert gotMsg1 &<> msg1
assert gotMsg2 &<> msg2//different by reference
```

There are a few things going on with the `OffHeapRAM` store above:

1. The `OffHeapRAM` store is explicitly started via a call to the `start` method.
2. Then we store objects within them via the `put` method, this returns to us an object reference of type `OffHeapObject` which we can use in order to obtain a copy of the object from the store.
3. We then obtain a copy of the stored objects from the store using the `get` method. Note that these objects are copies, so they are (by default) equal by value, but different by reference.

4. We then shut down the `OffHeapRAM` store using the `close` method. It is important that this is done so as to avoid a memory/resource leak.

Using OffHeapObject's

The returned `OffHeapObject` object references from our object store can be passed around our program as par normal objects. They may be deleted by using the `del` keyword or calling the `delete` method - this will remove their referenced object from the object store. Similarly, when `OffHeapObject` object references go out of scope and become garbage collected, the object to which they refer is removed from their host object store - however, it is still best practice to explicitly delete the object when it is known to not be of use. Here is an example:

```
del offHeapObj1
offHeapObj2.delete()
```

When working with `OffHeapObject` objects it is not necessary to have immediate knowledge of the object store to which they reference, since they have a `getManager` method which can provide this information, additionally, the `get` method may be called in order to obtain a copy of the object to which the `OffHeapObject` object refers, for example:

```
gotMsg1a = offHeapObj1.getManager().get(offHeapObj1)
gotMsg1b = offHeapObj1.get()

assert gotMsg1a == gotMsg1b //equal by value
assert gotMsg1a &<> gotMsg1b//different by reference
```

33.4.2 Creating Off Heap Disk Stores

The Off Heap Disk Store is a good mechanism for storing large amounts of temporary data off heap outside of RAM. The Off Heap Disk Store is backed by a memory mapped file which greatly enhances performance as spatially localized data is cached in memory.

As with the `OffHeapRAM` store, the `OffHeapDisk` store must be provided with a store size. Additionally however, a file path may be provided. This file will be used to store the data held in the `OffHeapDisk` store. If a file path is not provided, a temporary file will be created. If the file path exists already it will be erased, additionally, the file will be removed upon the `close` method of the store being called.

The `OffHeapDisk` store exposes an additional method `setPreallocate` which if called with `true` before the store is started, will result in the temporary file used to back the data of the store being fully allocated on disk. If this is not set then the file will grow as required.

Here is an example of the `OffHeapDisk` store in action, it is very similar to the `OffHeapRAM` store:

```
from com.concurnas.lang.offheap.storage import OffHeapDisk
from com.concurnas.lang.offheap import OffHeapObject

msg1 = ["hello" "world"]

10meg = 10 * (1024**2)

offHeapRamStore = new OffHeapDisk<String[]>(10meg)
offHeapRamStore.start()

offHeapObj1 OffHeapObject<String[]> = offHeapRamStore.put(msg1)
```

```
gotMsg1 = offHeapRamStore.get(offHeapObj1)

offHeapRamStore.close()

assert gotMsg1 == msg1//equal by value

assert gotMsg1 &<> msg1//different by reference
```

The Off Heap Disk Store is not designed for permanent Object persistence since at the point of shutdown of a process with a `OffHeapDisk` store the necessary handles (such as `OffHeapObject` objects) are lost. For true persistence, an off Off Heap Disk Map is recommended, since they provide a key reference that can be used in order to refer to objects post process shutdown and resumption.

33.4.3 Managing Off Heap Stores

We can examine the amount of space we have allocated and have remaining in the store via the `getCapacity` and `getFreeSpace` methods respectfully.

Over time an object store may become fragmented (see: Defragmentation). As such although there may appear to be plenty of space available for an object allocation, there in fact may not be due to fragmentation. In this situation Concurnas will automatically compact and defragment the store in order to free up space. This is a slow operation, for this reason it is recommended that stores be monitored for becoming close to capacity, a good rule of thumb is to over allocate space by 50% more than what is expected to be required.

We can adjust the amount of space allocated to the store by calling the `setCapacity(size long)` method. This method may be called on the store before or after it has been started via a call to the `start` method. If it is called after the `start` method, and the amount of space reduced, the store will be compacted and defragmented so as to ensure that it can fit into the newly allocated reduced space. If it cannot an exception will be thrown.

If a `OffHeapRAM` or `OffHeapDisk` store is closed (by calling of the `close` method) all outstanding `OffHeapObject` objects are invalidated. Attempting to extract an object referenced by a `OffHeapObject` will result in an exception. Additionally, Objects may not be persisted after the store has been closed.

Off heap stores and `OffHeapObject`'s cannot be shared between isolates. In cases where shared access to an off heap map is required, it is recommended that an actor be used in order to achieve this.

33.5 Off Heap Map

Now let us examine the core of the off heap memory support in Concurnas, off heap key value pair maps. These may reside either in memory via the `com.concurnas.lang.offheap.storage.OffHeapMapRAM` class or on disk via the `com.concurnas.lang.offheap.storage.OffHeapMapDisk` class. Both of these implementations implement the `java.util.Map` interface. They both behave in a very similar manner to the Off Heap stores we have previously explored.

The `OffHeapMapDisk` class offers all the same functionality as the `OffHeapMapRAM` class but with the added benefit of being disk backed, enabling permanent persistence of off heap objects. Although the `OffHeapMapDisk` implementation it is not as fast as the RAM based backing since it is disk backed, it does make use of memory mapped files in order to improve access times to data.

33.5.1 Creating Off Heap RAM Maps

In order to create an off heap map we must specify the size of our off heap structure in bytes, for instance, 100 MB is:

```
100meg = 100 * (1024**2)
```

This can then be used within our `OffHeapMapRAM` store, with generic qualification to map from a String as key to String[] as value as follows:

```
from com.concurnas.lang.offheap.storage import OffHeapMapRAM

msg1 = ["hello" "world"]
100meg = 100 * (1024**2)

offHeapRamStore = new OffHeapMapRAM<String, String[]>(100meg)
offHeapRamStore.start()

offHeapRamStore.put('msg1', msg1)

gotMsg1 = offHeapRamStore.get('msg1')

offHeapRamStore.close()

assert gotMsg1 == msg1//equal by value
assert gotMsg1 &<> msg1//different by reference
```

As with the Off heap stores there are a few things above going on:

1. The `OffHeapMapRAM` map is explicitly started via a call to the `start` method.
2. Then we store objects within the map via the `put` method, this returns a copy of the previous object persisted if any.
3. We then obtain a copy of the stored objects from the store using the `get` method. Note that these objects are copies, so they are (by default) equal by value, but different by reference.
4. We then shut down the `OffHeapMapRAM` map using the `close` method. It is important that this is done so as to avoid a memory/resource leak.

Notice that, unlike off heap stores, `OffHeapObject` objects are not returned from the `put` method calls. We do not need `OffHeapObject` objects because we can use the keys we have referenced.

33.5.2 Using Off Heap Disk Maps

The Off Heap Disk Map is a good mechanism for storing large amounts of temporary data off heap outside of RAM. The Off Heap Disk Map is backed by a memory mapped file which greatly enhances performance as spatially localized data is cached in memory.

As with the `OffHeapMapRAM` store, the `OffHeapMapDisk` store must normally be provided with a map size. Additionally however, a file path may be provided. This file will be used to store the data held in the `OffHeapMapDisk` store. If a file path is not provided, a temporary file will be created. If the file path is populated and points to a file which already exists, this file will be used as the backing store, any previously persisted mappings within the file will be accessible. It is through this means that persistence of data may be achieved.

The `OffHeapMapDisk` store exposes a number of additional methods:

- `setPreallocate` - if called with `true` before the store is started, will result in the temporary file used to back the data of the store being fully allocated on disk. If this is not set then the file will grow as required.

- setRemoveOnClose - if called with true, will result in the backing file used for the map being removed upon the close method being called on the map.
- setCleanOnStart - if called with true before the store is started, will result in the file used to back the data of the store being erased when the map is started.

Here is an example of the OffHeapMapDisk store in action, it is very similar to the OffHeapMapRAM store:

```
from com.concurnas.lang.offheap.storage import OffHeapMapDisk

msg1 = ["hello" "world"]
100meg = 100 * (1024**2)

offHeapRamStore = new OffHeapMapDisk<String, String[]>(100meg)
offHeapRamStore.start()

offHeapRamStore.put('msg1', msg1)

gotMsg1 = offHeapRamStore.get('msg1')

offHeapRamStore.close()

assert gotMsg1 == msg1//equal by value
assert gotMsg1 &<> msg1//different by reference
```

33.5.3 Off heap map management

As with off heap stores, the same points regarding management apply to off heap maps...

We can examine the amount of space we have allocated and have remaining in the map via the getCapacity and getFreeSpace methods respectfully.

Over time an object map may become fragmented (see: Defragmentation). As such although there may appear to be plenty of space available for an object allocation, there in fact may not be due to fragmentation. In this situation Concurnas will automatically compact and defragment the map in order to free up space. This is a slow operation, for this reason it is recommended that maps be monitored for becoming close to capacity, a good rule of thumb is to over allocate space by 50% more than what is expected to be required.

We can adjust the amount of space allocated to the map by calling the setCapacity(size long) method. This method may be called on the map before or after it has been started via a call to the start method. If it is called after the start method, and the amount of space reduced, the map will be compacted and defragmented so as to ensure that it can fit into the newly allocated reduced space. If it cannot an exception will be thrown.

Objects may not be persisted to a Off heap map after it has been closed.

Off heap maps cannot be shared between isolates. In cases where shared access to an off heap map is required, it is recommended that an actor be used in order to achieve this.

33.6 Schema evolution

One of the strongest points of the Concurnas off heap map implementation is its support for schema evolution. We term schema evolution as being changes to a class after it has been persisted. For instance, adding a new data field, changing a type etc. This turns out to be a surprisingly normal operation performed in enterprise computing and unfortunately the subsequent required data migration is something which consumes a lot of time and effort. Traditionally this would cause a problem for us upon deserialization since the persisted version of the class code would not match that of the current 'live' version, but Concurnas is largely able to account for these sorts of evolutionary changes.

Additionally, Concurnas is able to store multiple evolved versions of the same class within an off heap data structure (either the map or store objects above). In this way Objects which have been serialized in a previous format usually do not require explicit migration to a new format.

33.6.1 Supported evolutions

Concurnas is able to support the following evolutions to Objects in isolation and in combination:

Removing a field

```
class MyClass(firstName String, sirName String)
//Later version:
class MyClass(firstName String)
```

When we deserialize a class having an evolved definition with a removed field, this removed field will simply be ommitted from the deserialized object.

Adding a field

```
class MyClass(firstName String)
//Later version:
class MyClass(firstName String, sirName String)
```

When we deserialize a class having an evolved definition with an additional field, this additional field will be set to its default/initial value, the equivalent of `0` for a non array primitive type, and `null` otherwise.

If a default value/initial value for the new field is specified then this value will be populated for the new field in deserialized objects.

Changing the type of a field

```
class MyClass(firstName String, sirName String, userid byte)
//Later version:
class MyClass(firstName String, sirName String, userid long)
```

When we deserialize a class having an evolved definition with a field with a different type from that of its persisted version the behaviour we encounter is contingent upon the variant of type evolution employed. This is summarized below:

Variant	Example	Behaviour
Boxed? primitive to Boxed? primitive	`int -> double`	Equivalent to a cast operation
Boxed? primitive array to Boxed? primitive array	`int[] -> Double[]`	Equivalent to a cast operation
Any array to scalar	`Integer[] -> Integer`	Cannot be converted
Any scalar to array	`Integer -> Double`	Cannot be converted
Any type to String	`MyClass -> String`	Equivalent to `toString`
Any array to String array	`MyClass[] -> String[]`	Equivalent to `array^toString`
Class to trait	`Child -> MyTrait`	Equivalent to a cast operation
Subclass to superclass	`Child -> Parent`	Equivalent to a cast operation
Superclass to subclass	`Parent -> Child`	Cannot be converted
Unrelated classes	`Myclass -> MyOtherClass`	Cannot be converted

In situations in which a value cannot be converted, the default value for the type (`0` for a non array primitive type, and `null` otherwise) will be used unless a default value/initial value for the field is specified.

Oh darlin'
In a sky full of people, only some want to fly
Isn't that crazy
In a world full of people, only some want to fly
Isn't that crazy, crazy
In a heaven of people there's only some who want to fly
Ain't that crazy, oh babe, oh darlin'
In a world full of people there's only some want to fly
Isn't that crazy, isn't that crazy, isn't that crazy, isn't that crazy

- Crazy from Seal by Seal (1991).

34. Expression lists

Concurnas provides expression lists, this is a neat feature which enables a more natural way of writing expression related code that would otherwise have to be written as a set of chained together calls using the dot operator and/or function invocation brackets. Example:

```
class Myclass(b int){
   def resolve(a int) => (a+b)*2
}

res = Myclass 4 resolve 4 //expression list
//res now resolves to 16
```

Concurnas interprets `mc doit 4` to resolve to `mc.dotit(4)`. Without expression lists the above would look like: `mc.dotit(4)`. Concurnas will evaluate all possible interpretations of the defined expression list. If more than one valid interpretation is possible then this will be flagged up as an ambiguous compilation error which will require disambiguation (e.g. explicitly using the dot operator or function invocation arguments).

Combined with extension functions this affords us some very concise and powerful domain specific syntax.

```
def int min() = this*60*60

10 min//resolves to 3600 - which is the number of seconds in 10 minutes.
```

34.1 Double dot and direct dot Operator

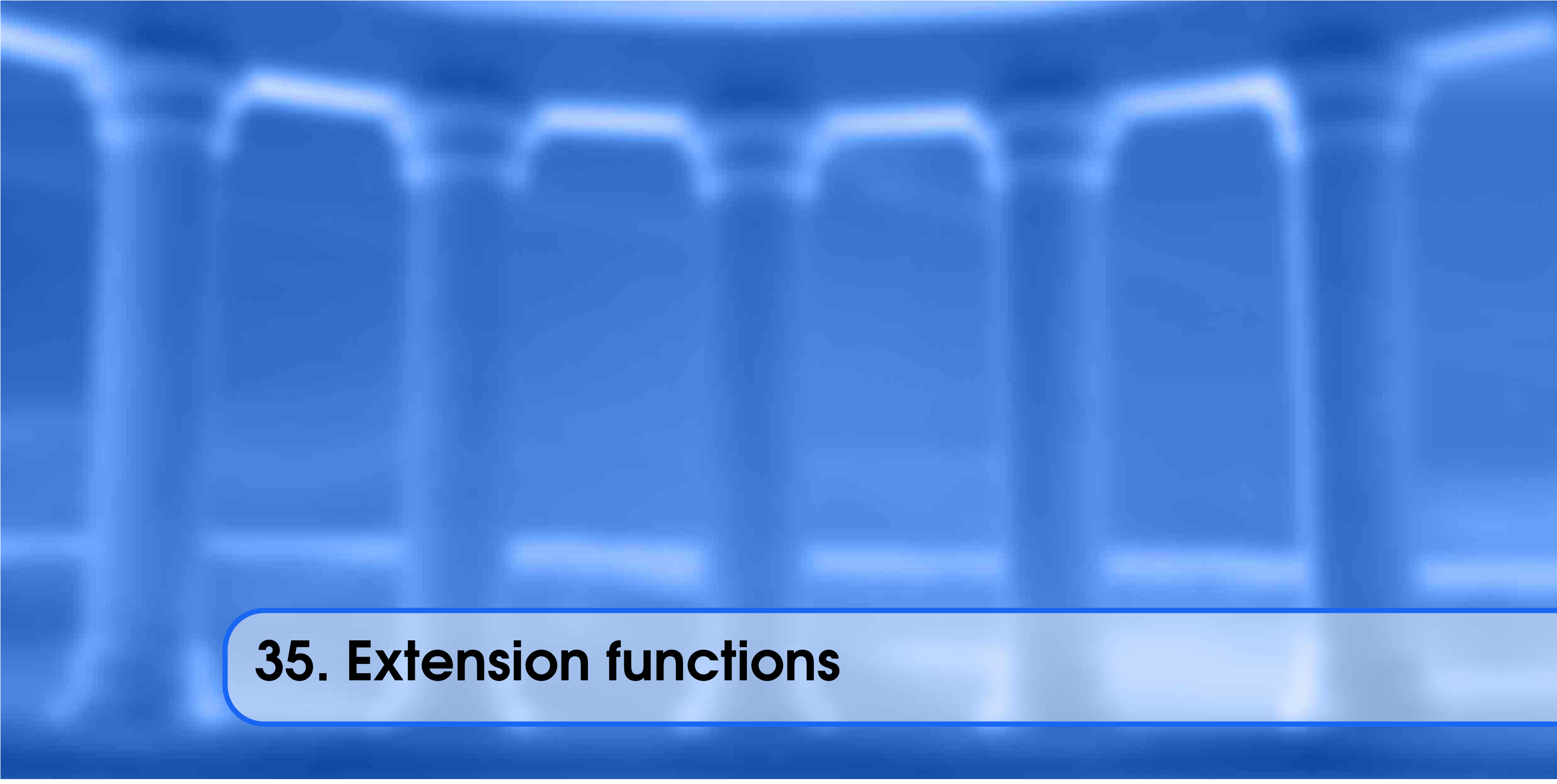
Concurnas is able to interpret expression lists to make use of the double dot `..` and direct dot `\.` operators in addition to the single dot `.`:

```
class MyClass{
   public answers = [0,0]
   def oneCall() = answers[0]=10
   def twoCall() = answers[1]=99

}

mc = new MyClass()
mc oneCall twoCall answers //equvilent to: 'mc..oneCall()..twoCall()\.answers'.
    Resolves to: [10, 99]
```

34.1.1 Element Precedence

The elements of the expression list are themselves expected to be valid expressions. E.g. The below resolves to 16 and not 14 as the plus operator defined in the 2+2 expression is valid and takes precedence over the expression list.

```
class Myclass(b int){
   def doit(a int) = (a+b)*2
}

mc = Myclass(4)
res = mc doit 2+2
//res is now 16
```



35. Extension functions

Concurnas has support for extension functions. These allow for functionality to be added to classes without needing to interact with the class hierarchy (e.g.extending the class etc). They are a convenient alternative to having to use utility functions/methods/classes which take an instance of a class and often permit a more natural way of interacting with objects.

```
def String repeater(n int){//this is an extension function
   return String.join(", ", java.util.Collections.nCopies(n, this))
}

"value".repeater(2) // returns: 'value, value'
```

Public properties or methods of the extended class may be referenced inside the body of the extension function. Non public items may not be referenced:

```
class MyClass(public name String){
   def amethod() = 'MyClass Method'
   private unseen = 1//not accessible from the extension function below
}

def MyClass extFunc(){
   "" + [this.name, this.amethod()]
}

new MyClass('dave').extFunc() //resolves to '[dave, MyClass Method]'
```

35.1 Extension Methods

Extension methods are the same as extension functions, just defined within a class. For example:

```
class MyClass(public name String){
   def amethod() = 'MyClass Method'
}

class AnotherClass{
   def MyClass extFunc(){
      " + [this.name, this.amethod()]
   }

   def dotask(){
      new MyClass('dave').extFunc()
   }
}

new AnotherClass().dotask()//resolves to '[dave, MyClass Method]'
```

Extension methods may only be defined as private, protected.

35.2 This keyword

The `this` keyword can be used inside extension functions/methods and refers to the instance of the class being extended. In the case of an extension method it may be qualified in order to refer to the containing class:

```
class MyClass(){
   def amethod() = 'MyClass Method'
}

class AnotherClass{
   def amethod() = 'AnotherClass Method'

   def MyClass extFunc(){
      "" + [this.amethod(), //this defaults to refer to the MyClass instance
      this[MyClass].amethod(),//explicitly directed to MyClass
      this[AnotherClass].amethod()]//explicitly directed to AnotherClass
   }

   def dotask(){
      new MyClass().extFunc()
   }
}

new AnotherClass().dotask()//resolves to '[MyClass Method, MyClass Method,
    AnotherClass Method]'
```

Note that in cases where an extension method/function has the same input signature as the method associated with the object being invoked then the extension function will take priority:

```
class MyClass{
   def countdown(a int) => "MyClass instance: {a}"
}

def MyClass countdown(a int){
   ""+ 0 if a == 0 else "{a}, {this.countdown(a-1)}"
}

res = new MyClass() countdown 5
```

```
//res == "5, 4, 3, 2, 1, 0" and NOT "10, MyClass instance: 9"
```

35.3 Super Keyword

The super keyword can be used inside extension functions/methods in the same way as the this keyword, however, unlike the this keyword subsequent method invocations always refer to the instance of of the class being extended. This thus explicitly avoids an otherwise recursive call in the case of a matching input signature between the extension function/method and method associated with the object upon which the invocation is taking place.

```
class MyClass{
   def countdown(a int) => "MyClass instance: {a}"
}

def MyClass countdown(a int){
   ""+ 0 if a == 0 else "{a}, {super.countdown(a-1)}"
}

res = new MyClass() countdown 5

//res == "10, MyClass instance: 9" and NOT "5, 4, 3, 2, 1, 0"
```

35.4 Resolution of extension functions vs instance methods

In cases where an extension function overrides the definition of a method of the class it's extending, the extension function will be invoked in place of the instance method:

```
class MyClass(){
   def amethod() = 'instance method'
}

def MyClass amethod(){
   'Extension function'
}

new MyClass().amethod() //resolves to 'Extension function'
```

35.5 Extension functions with generics

Class or local generics may be used as par normal for extension functions.

```
class MyClass<X>{
   def amethod() = 12
}

fun  MyClass<X> repeater<X>( ) {//extension function with extendee qualified via
    local generic type
   ""+ [amethod(), this.amethod()]
}

new MyClass<String>().repeater<String>()// resolves to: '[12, 12]'
```

As per usual, qualification of generic types may also be inferred:

```
class MyClass<X>{
   def amethod() = 12
}

fun  MyClass<X> repeater<X>( ) {//extension function with extendee qualified via
    local generic type
   ""+ [amethod(), this.amethod()]
}

new MyClass<String>().repeater()// resolves to: '[12, 12]' as local generic is
    qualified to String via inference
```

35.6 Extension functions as operator overloaders

Extension functions can also be used in order to provide operator overloading. For example, in order to overload + plus for an `Arraylist`:

```
from java.util import ArrayList

fun ArrayList<X> plus<X>(toAdd X) {
   this.add(toAdd)
   this
}

ar = new ArrayList<int>()
ar = ar + 12 //operator overload
ar//resolves to '[12]'
```

35.7 Extension functions on primitive types

In addition to object classes, extension functions can be defined for primitive types.

```
def int meg() => this * 1024 * 1024

12 meg //-> 12582912
```

Auto boxing/unboxing is performed automatically:

```
def Integer meg() => this * 1024 * 1024

12 meg //-> 12582912
```

35.8 References

References to extension functions/methods can be made as par usual:

```
def String repeater(n int){//this is an extension function
   return String.join(", ", java.util.Collections.nCopies(n, this))
}

rep = "value".repeater&(2) //creates a reference
```

```
rep() // returns: 'value, value'
```

Note that bindings to extension functions are created at compile time, not runtime. Thus changes to class hierarchies etc will require a recompilation of the extension function and related code in order to be incorporated. In practice this is not a significant problem, but it is something to be aware of nevertheless.

He was kind of foreign, you know? Like the Hub people. Bald as a coot. I remember thinking, 'you look like a young man, mister, but you look like you've been a young man for a long, long time if I'm any judge.' Normally I wouldn't have any man there, but he sat and talked to her in his foreign lingo and sang her songs and little poems and soothed her, and back she came, out of thin air, and I was ready and it was one, two, done. And then she was gone. Except that she was still there, I think. In the air.

- Pratchett, Terry. (2001). Thief of Time. Harper Collins.

36. Language Extensions

A really useful and exciting component of Concurnas is its support for language extensions. These enable us to embed code defined in other programming languages directly within Concurnas. This provides us with the following advantages:

- **Convenience** - By using language extensions we avoid the need to switch to other tools/-toolchains, products etc. this greatly simplifies our working environment and build process.
- **Appropriateness** - By using language extensions we're able to use the right language for the task at hand, without having to try and fit our solution around the language we're mainly programming in.
- **Allows for compile time checking of code** - It's always better to know about errors ahead of runtime. Language extensions may be integrated such that they can report errors back to the core Concurnas compiler for highlighting at compile time.
- **Compiles down to bytecode** - Language extensions are compiled to bytecode and are thus executed at the same speed as 'native' high performance Concurnas code.
- **Easy to integrate** - With Concurnas language extensions we need only concern ourselves with the tokenization, parsing and some semantic analysis of our guest language. We can avoid bytecode or machine code generation, as we instead output Concurnas code. This generated Concurnas code is checked and treated as par normal Concurnas code in the subsequent analysis and compilation stages of the core Concurnas compiler. This removes the majority of the hard work from the process of supporting guest languages and allows us to focus on the more interesting part concerning language structure and semantics.

36.1 Using language extensions

We can make use of pre defined language extensions by importing them in much the same way as conventional `import` statements. We simply need use the `using` keyword to import the appropriate language extension class. The following is valid:

```
using com.myorg.langs.mylang
using com.myorg.langs.mylang as MyLanguage

from com.myorg.langs using mylang
from com.myorg.langs using mylang as MyLanguage

from com.mycode using *
using com.mycode.*
```

In addition to the above, language extensions may be imported using the regular `import` syntax and used as regular classes.

36.1.1 Using imported language extensions

The syntax for using an imported language extension within a language extension expression is as follows:

```
LangExtentionName '||'code '||'
```

For example, a lisp like language may be used within a language extension expression as follows:

```
using com.myorg.langs.mylisp

result = mylisp||(+ 1 2 3 )||
```

The code expressed within the pair of `||`'s is to be defined in that of the guest language extension, in the above example, `mylisp`. At compilation time, this code is passed to an instance of the `mylisp` language extension which itself (after any appropriate error checking and other operations it performs) returns a String output of Concurnas code - this Concurnas code is then handled as normal by the Concurnas compiler.

Language extensions may be used at any point within a Concurnas program, provided that the guest language extension supports it. I.e. at top level, within a function, within an extension function/method, at field level of a class and within a method. Language extensions may only be referenced as names as par above - they may not include parameters since they are not treated as variables/classes etc.

As with Strings when defining code in a guest language the backslash may be used as an escape character, for instance to escape the pipe operator:

```
using com.myorg.langs.usesPipe

result = usesPipe||something \| something ||//escape the |
```

Language extensions are executed as par normal Concurnas code, i.e as isolates.

36.2 Creating language extensions

Concurnas offers a powerful and concise API (exposed under: `com.concurnas.lang.langExt`) for implementing language extensions, of either existing programming languages or totally new ones! Let us first familiarise ourselves with the typical phases of language compilation and execution implemented in some manor by most language compilers/runtimes, from the perspective of Concurnas:

1. **Lexical analysis**. Splitting the input code into parable segments. (`val a = 10 -> 'val', 'a', '=', '10'`)

2. **Parsing**. Turning our stream of tokens from the previous stage into a verifiably syntactically correct structure.
3. **Semantic analysis**. Attributes meaning to our previously defined structure and verifies consistency of that meaning (e.g. variable types, scopes).
4. **Intermediate code generation (optional)**. Transcompiling our previous data structure to another format for easier working in downstream phases.
5. **Bytecode generation**. The final stage of the Concurnas compiler outputs *executable* bytecode.
6. **Code optimization**. The Concurnas runtime (operating on the JVM) verifies the integrity of the previously output bytecode whilst performing some runtime transformations and optimizations.
7. **Machine code generation**. The JVM creates optimized machine code from the previous bytecode and executes this machine code upon the CPU and GPUs available.

Aside: We see above that there are a large number of phases to the compilation process, in fact, within the Concurnas compiler (and most other compilers) these steps are often themselves broken down into further sub phases. Execution of these phases takes a non epsilon amount of time. It is for this reason (amongst others) that, in the interests of the high performance computing which Concurnas offers, steps 1-5 of 7 are performed at compilation time, once only, and the steps 6 and 7 are performed only infrequently and in a highly optimized manner, at runtime. This approach of course gives a natural performance boost over a runtime based scripting language which if naively implemented would implement all 7 of these stages (and perhaps not even implementing the final stage if it were a fully interpreted language) at runtime.

The good news is that with Concurnas language extensions we need focus our effort upon only steps 1, 2 and step 3 to some extent (depending on the complexity of our language), before finally outputting a String of Concurnas code as our intermediate code for step 4. Concurnas will then handle steps 5 onwards (whilst also implicitly performing steps 1 - 3 on the provided output Concurnas code). The first 3 phases tend to be where the most interesting and value added work regarding programming languages reside and so this puts us in a good position.

36.2.1 Defining language extensions

Language extension classes can be defined using idiomatic Concurnas code, affording us all the features of normal Concurnas code. Furthermore, Language extension classes are executed like normal Concurnas code, within an isolate - which greatly eases state management.

Language extension classes are required to output Concurnas code in the form of a String. Within that String of valid Concurnas code any aspect of Concurnas may be used. For instance, we can create new variables, classes, methods, functions etc as well as use those defined outside the definition of any language extension expressions.

Language extension classes must implement the `com.concurnas.lang.langExt.LanguageExtension` trait and expose a publicly accessible zero argument constructor. This trait stipulates the following two methods which must be implemented:

- `def initialize(line int, col int, loc SourceLocation, src String)Result`
- `def iterate(ctx Context)IterationResult`

When a language extension expression is encountered Concurnas will create one instance of the appropriate imported class (imported via the aforementioned `using` keyword) via its publicly accessible zero argument constructor. After this, the `initialize` method is called once. This provides the following parameters:

- `line int` - The line number at the start of the `||` block.
- `col int` - The column number at the start of the `||` block.
- `loc com.concurnas.lang.langExt.SourceLocation` - The location of the language exten-

sion expression - either:
  - At top level
  - Within a function
  - Within an extension function/method
  - At field level of a class
  - Within a method
- `src String` - The code of the language extension expression defined within the bounds of the `||` block.

A `com.concurnas.lang.langExt.Result` object must be returned from the `initialize` method. This includes a list of any errors, warnings which may have been encountered during initial compilation - for instance if the code is syntactically invalid (if analysed at this point) or defined at an unsupportable source location.

Next, Concurnas will call the `iterate` method. Concurnas is an iterative compilation compiler, as such multiple calls to the `iterate` method may be made in the process of compiling the source file in which the language extension expression is defined as more information regarding the context of the defined code becomes known. A language extension implementation may need to account for this.

A `com.concurnas.lang.langExt.Context` object is provided to the `iterate` method. This allows the language extension implementation to interrogate the host Concurnas compiler at the point of usage of the language extension via a language extension expression. The following methods are exposed on the `Context` object and can be of particular use in this regard:

- `def getVariable(name String)Variable?` - Returns the variable (if any) in scope matching the provided name.
- `def getclass(name String)Class?` - Returns the class (if any) in scope matching the provided name.
- `def getMethods(name String)List<Method>` - Returns all callable methods in scope matching the provided name.
- `def getNesting()List<Location>` - Returns a list of the nesting hierarchy in terms of methods and classes in which the language extension expression code resides.

Where relevant the aforementioned `Variable` and `Method` classes contain type information. This type information is expressed as a java bytecode field descriptor String, details of this can be found here:
JVM Spec.

The `iterate` method is required to output a String of Concurnas code [1] each time it is called. This String is then passed to the core Concurnas compiler for subsequent compilation.

The `iterate` method is expected to return a `com.concurnas.lang.langExt.IterationResult` object. This includes a list of any errors, warnings which may have been encountered during iterative compilation (for instance scope/type errors, usage context warnings etc). Additionally this object is to include a reference to the aforementioned output String of Concurnas code.

36.2.2 Errors and warnings

One of the most important jobs of any language compiler is the verification and validation of code. This can take place in either the initialization or iterative compilation stages of a Concurnas language extension, i.e. within the the `initialize` and `iterate` methods respectfully.

Problems in the verification and validation of code may take the form of errors or warnings, multiple of which may be outputted by the compilation phases, the `Result` class (of which `IterationResult` inherits) contains two list fields for holding these errors and warnings.

[1]this technically makes language extensions transpilers

These errors and warnings are then propagated back via the instances of the `Result` class returned from the `initialize` and `iterate` methods back to the main Concurnas compiler and prefixed with the name of the language extension referenced within the language extension expression.

If either the `initialize` or `iterate` methods throw an exception, this will be caught by the core Concurnas compiler and wrapped up as an error originating from the language extension (and prefixed as such). If the `initialize` throws an exception then subsequent compilation (i.e. invoking of the `iterate`) will not take place.

36.2.3 Practical tips

There are some practical tips which can be considered when we're building language extensions:

Abstract syntax trees

Abstract syntax trees or ATS's are an excellent intermediate data format representation. Many compilers make use of these. More information regarding this may be found here: AST

The visitor pattern

The visitor pattern crops up time and time again as a really useful method for navigating the aforementioned ATS. More information regarding this may be found here: Visitor pattern

Iterative compilation

Since Concurnas is an iterative compiler it is to be expected that multiple calls to our aforementioned `iterate` method will be made during the course of compilation. It is possible that a type of a dependant class, variable etc which was previously unknown will become known during this iterative compilation. Our language extension compiler may need to take this into account - i.e. by avoiding repeating work which it has already completed or by say caching a previous compilation result if it is agnostic to these iterative calls. It may also be appropriate to output a known to be incorrect variant of the expected output code (with errors flagged as appropriate) such that dependant code may be allowed to continue iterative compilation in some manner.

Focus of effort

Given that our language extension will be outputting Concurnas code, and that this Concurnas code will be passed to the core Concurnas compiler for further compilation as par normal Concurnas code, it is recommended that attention be focused on generating syntactically valid Concurnas code over semantically correct. In effect the core Concurnas compiler acts like a safety net for any problems which may have been missed during processing of the language extension code. The language extension doesn't have to be perfect, the core Concurnas compiler will do this hard work for us.

36.3 Example language extension

Here is a full example of a very simple lisp like language `com.mycompany.myproduct.langExts.miniLisp`.

In the interests of brevity this example focuses upon only a small subsection of syntax and takes a few shortcuts (imperfect parsing, operates upon numerical types only, no variable assignment/function definitions) etc, but it nevertheless serves nicely to illustrate the typical phases of compilation as well as the sorts of verification and validation we must perform within a language extension.

```
from com.concurnas.lang.LangExt import LanguageExtension, Context, SourceLocation
from com.concurnas.lang.LangExt import ErrorOrWarning, Result, IterationResult
from com.concurnas.lang.LangExt import Variable, Method, MethodParam
from java.util import Stack, Scanner, ArrayList, List

```

```
/*
* sample inputs:
*  (+ 1 2 3 )
*  (+ 1 2 n )
*  (+ 1 2 ( * 3 5) )
*  (+ 1 2 ( methodCall 3 5) )
*/

///////////////// AST /////////////////
enum MathOps(code String){PLUS("+"), MINUS("-"), MUL("*"), DIV("/"), POW("**");
   override toString() => code
}

open class ASTNode(-line int, -col int){
   nodes = new ArrayList<ASTNode>()
   def add(toAdd ASTNode){
      nodes.add(toAdd)
   }

   def accept(visitor Visitor) Object
}

class LongNode(line int, col int, along Long) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}

class MathNode(line int, col int, what MathOps) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}

class MethodCallNode(line int, col int, methodName String) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}

class NamedNode(line int, col int, name String) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}


//////////////// Parser ////////////////

def parse(line int, col int, source String) (ASTNode, Result) {
   rootNode ASTNode?=null

   warnings = new ArrayList<ErrorOrWarning>()
   errors = new ArrayList<ErrorOrWarning>()

   sc = new Scanner(source)
   nodes = Stack<ASTNode>()

   while(sc.hasNext()){
      if(sc.hasNextInt()){
         llong = sc.nextLong()

         if(nodes.isEmpty()){
            errors.add(new ErrorOrWarning(line, col, "unexpected token: {llong}"))
         }
         else{
            nodes.peek().add(LongNode(line, col, llong))
         }
      }else{
         str = sc.next()
         match(str){
            ")" => {

               if(nodes.isEmpty()){
                  errors.add(new ErrorOrWarning(line, col, "unexpected token:
                      {str}"))
               }else{
                  rootNode = nodes.pop()
                  if(rootNode <> null and not nodes.isEmpty()){
                     nodes.peek().add(rootNode)
                  }
               }
            }
            else => match(str){
               "( +" => nodes.push(MathNode(line, col, MathOps.PLUS))
               "( -" => nodes.push(MathNode(line, col, MathOps.MINUS))
               "( *" => nodes.push(MathNode(line, col, MathOps.MUL))
               "( /" => nodes.push(MathNode(line, col, MathOps.DIV))
               "( **" => nodes.push(MathNode(line, col, MathOps.POW))
               else =>{
                  if(str.startsWith("(")){
                     str = str.substring(1, str.length());
                     nodes.push(MethodCallNode(line, col, str))
                  }else{
                     if(nodes.isEmpty()){
                        errors.add(new ErrorOrWarning(line, col, "unexpected
                           token: {str}"))
                     }else{
                        nodes.peek().add(NamedNode(line, col, str))
                     }
                  }
               }
            }
         }
      }
   }

   rootNode?:LongNode(0, 0, 0), new Result(errors, warnings)
}

////////////////Visitors////////////////

trait Visitor{
   def visit(longNode LongNode) Object
   def visit(mathNode MathNode) Object
   def visit(namedNode NamedNode) Object
   def visit(methodNode MethodCallNode) Object
}

class CodeGennerator ~ Visitor{
   def visit(longNode LongNode) Object{
      "{longNode.along}"
   }

   def visit(mathNode MathNode) Object{
      String.join("{mathNode.what}", ("" + x.accept(this)) for x in
          mathNode.nodes)
   }

   def visit(methodCall MethodCallNode) Object{
      "{methodCall.methodName}(" + String.join(", ", ("" + x.accept(this)) for x
          in methodCall.nodes ) + ")"
   }

   def visit(namedNode NamedNode) Object{
      "{namedNode.name}"
   }
}
cg = CodeGennerator()

private numericalTypes = new set<String>()
{
   numericalTypes.add("I")
   numericalTypes.add("J")
   numericalTypes.add("Ljava/lang/Long;")
   numericalTypes.add("Ljava/lang/Integer;")
}

class NodeChecker(ctx Context) ~ Visitor{
   errors = ArrayList<ErrorOrWarning>()
   warnings = ArrayList<ErrorOrWarning>()

   def visit(longNode LongNode) Object => "I"
   def visit(mathNode MathNode) Object{
      for( x in mathNode.nodes){
         x.accept(this)
      }

      "I"
   }

   private def raiseError(line int, col int, str String){
      errors.add(new ErrorOrWarning(line, col, str))
   }

   def visit(namedNode NamedNode) Object{
      vari Variable? = ctx.getVariable(namedNode.name)
      type = vari?.type

      if(type){
         if(type not in numericalTypes){
            raiseError(namedNode.line, namedNode.col, "Variable: {namedNode.name}
                is expected to be of numerical type")
         }
      }else{
         raiseError(namedNode.line, namedNode.col, "Unknown variable:
            {namedNode.name}")
      }

      return type
   }


   def visit(methodCall MethodCallNode) Object{
      meths List<Method> = ctx.getMethods(methodCall.methodName)

      found = false

      argsWanted = methodCall.nodes
      wantedSize = argsWanted.size()

      for(method in meths){
         ret = method.returnType
         if(ret=="V" or ret in numericalTypes){
            margs ArrayList<String> = method.arguments
            if(wantedSize == margs.size()){
               found=true
            }else{
               //check to see if an arg is an array - then can attempt to vararg
                   in
               found = margs.stream().anyMatch(a => a <> null and
                   a.startsWith("["))
            }

            if(found){
               break
            }
         }
      }

      if(not found){
         raiseError(methodCall.line, methodCall.col, "Cannot find method:
             {methodCall.methodName}")
      }

      for(arg in argsWanted; idx){
         atype = ""+arg.accept(this)
         if(atype not in numericalTypes){
            raiseError(methodCall.line, methodCall.col, "method argument {idx+1}
                is expected to be of numerical type not: {atype}")
         }
      }

      return "I"
   }
}


///////////////// Lang /////////////////

class SimpleLisp ~ LanguageExtension{
   line int
   col int
   rootNode ASTNode?=null

   def initialize(line int, col int, location SourceLocation, source String)
       Result {
      this.line = line
      this.col = col

      //build abstract syntax tree...
      (this.rootNode, result) = parse(line, col, source)

      result
   }

   def iterate(ctx Context) IterationResult {
      rootn = rootNode
      if(rootn){
         nc = NodeChecker(ctx)
         rootn.accept(nc)

         code = ""
         if(nc.errors.isEmpty() and nc.warnings.isEmpty()){
            code = ""+rootn.accept(cg)
         }

         new IterationResult(nc.errors, nc.warnings, code)

      }else{
         errors = ArrayList<ErrorOrWarning>()
         warnings = ArrayList<ErrorOrWarning>()
         new IterationResult(errors, warnings, "")
      }
   }
}
```

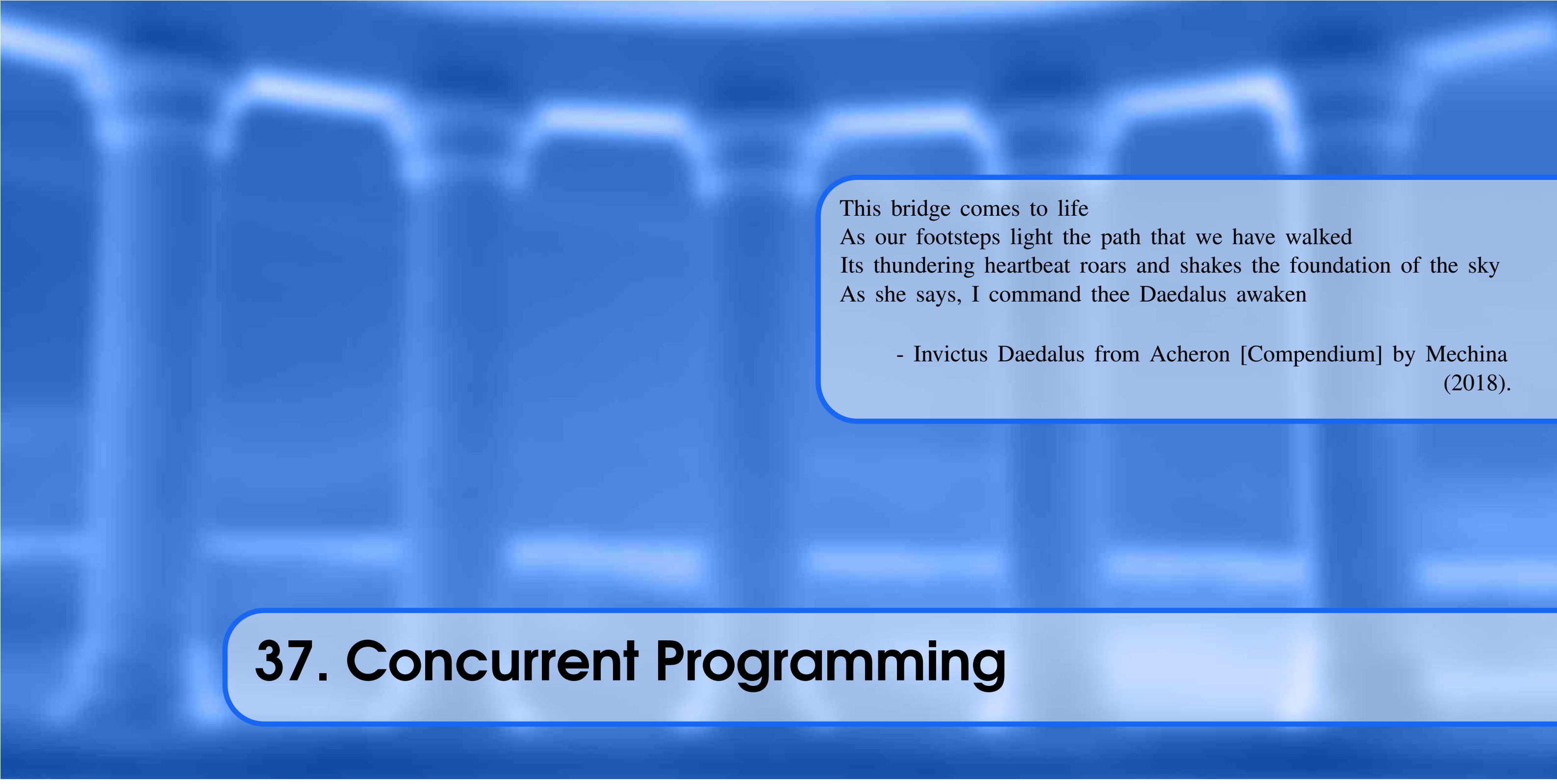
36.3.1 Using our example language extension

We can now use or previously defined language extension:

```
from com.mycompany.myproduct.langExts.miniLisp using SimpleLisp

result = SimpleLisp||(+ 1 2  3 3 ) )||//result == 9
```

IV

Concurrent, Distributed and GPU



This bridge comes to life
As our footsteps light the path that we have walked
Its thundering heartbeat roars and shakes the foundation of the sky
As she says, I command thee Daedalus awaken

- Invictus Daedalus from Acheron [Compendium] by Mechina (2018).

37. Concurrent Programming

Just as object providers allow us to separate the more interesting algorithmic work from the plumbing of an overall system, so does the concurrency model of Concurnas. With the Concurnas concurrency model we aim to eliminate the hard work and risk from building concurrent solutions, permitting the developer to focus on the more algorithmic and business relevant parts of their work and enabling that to scale.

This is achieved via six principal areas: isolates, actors, refs, reactive programming, temporal computing, transactions and parfor.

The solutions in this section are typically best aligned to scaling problems which are task based in nature. For solutions to more data oriented problems, taking advantage of Concurnas' support for GPUs is advisable, for more details on this see the GPU/Parallel programming chapter.

Solutions created in Concurnas using the concurrency primitives described here will naturally scale in line with the maximum physical hardware provided to them. However, they can only do so within the bounds of that physical hardware. In time, scaling beyond the confounds of a single machine is necessary, and here we enter the realm of distributed computing, for more details on this see the Distributed computing chapter.

37.1 Isolates

Isolates are like threads in conventional programming languages. Execution is concurrent and non deterministic. They are best suited for solving task based concurrent problems. In Concurnas they are automatically managed and mapped on to underlying hardware threads, the number of which are spawned being contingent on the underlying machine specification (usually the number of logical processor cores available). The upper bound for the number of isolates, and therefore concurrent tasks is constrained only by the amount of heap memory one has access to, as opposed to the much more restrictive limit in terms of hardware threads which can be created in conventional programming languages.

One of the most beautiful aspects of the isolate model, as we shall see, is that whether you have access to one processor core, or 100, your isolates will behave in a deterministic manner. This means that you don't have to re-write all your software when upgrading from a single core machine to one with 100 cores, and in fact, with idiomatic Concurnas code with many spawned isolates, your software will normally automatically take advantage of that added n core count and operate in a $\frac{1}{n}$th of the time.

The syntax to spawn an isolate is:

```
{/*code to execute*/}!.
```

Lets create some isolates now:

```
def gcd(x int, y int){//greatest common divisor of two integers
   while(y){
      (x, y) = (y, x mod y)
   }
   x
}

calc1 = {gcd(8, 20)}!//run this calculation in a isolate
calc2 = {gcd(6, 45)}!//run this calculation in a separate isolate, concurrently!

calc3 = {gcd(calc1, calc2)}!//wait for the results of calc1 and calc2 before
    calculating calc3, also in an isolate
```

Above, we are initially creating two isolates which will execute concurrently to calculate some greatest common divisor (`gcd`) values. When the values of these executions are known then these will be passed into a third invocation of `gcd`, again for concurrent execution.

If we have a single method to which we wish to run within an isolate, one need not use the curly brace notation: `{/*code to execute*/}!`, but may simply append the function call with the bang operator: `!`:

```
gcd(8, 20)!
//is equivilent to:
{gcd(8, 20)}!
```

Isolates are scheduled in a fair (currently non pre-emptive) manner and are able to pause execution at certain blocking points, such as in accessing a ref which does not already have a value assigned. This allows the underlying hardware processor to execute other isolates whilst whatever is blocking execution is resolved, maximising throughput!

Care should be taken to avoid calling blocking io code in an isolate as this will have the effect of locking up the underlying execution thread for the duration of the blocking operation. Instead, consider using an actor dedicated for i/o or using a reactive computing pattern (made easy with the support provided by Concurnas see the Reactive programming section below). Care should also be taken with actively infinitely looping code, which is generally considered poor practice in any programming language - luckily Concurnas provides us with lots of alternatives to this.

37.1.1 Isolate dependencies

Isolates operate within their own dedicated memory spaces. It is not possible to directly share memory between isolates, rather, isolate share state via communication: either with refs, actors or the variable tagged with the shared keyword. This makes reasoning and implementation of concurrent algorithms with Concurnas much easier than conventional programming languages which allow all state to be shared and where the developer must explicitly apply concurrency control on the subsection of shared state which is actually intentionally shared within the program.

This isolation of state is achieved by Concurnas explicitly copying all the dependencies of an isolate lazily upon execution. For example:

```
n = 10

nplusone = { n += 1; n }!//perform this calculation in an isolate
nminusone = { n -= 1; n }!//and this

assert nplusone == 11
assert nminusone == 9
assert n==10//n always remains unchanged
```

The above code will always provide a consistent output, despite the isolates non deterministic nature, since the `n` variable dependency is copied into each of the isolates and is not shared between them. Thus changes made to the variable in one isolate do not affect the other. Notice how we've not had to define anything in the way of a critical section, synchronization, lock management etc.

The dependency copy itself is a default copy in Concurnas, i.e. a deep copy of the isolates dependency, so for object dependencies which are very large, either making use of an actor (see the Actors section) or marking the dependant variable as shared (see the Shared variables and classes section) may be a more appropriate option.

Module level state

Variables defined outside of a function/class/actor (termed module level state) are copied in a special manner when it comes to usage within isolates. The rule is that state defined within the module spawning the isolate will be copied, but module level state defined in modules other than that spawning the isolate will be reinitialized (which means running all of the associated top level code to initialize them) within the isolate - i.e. the current state not copied.

For example:

```
//code defined in com.mycompany.library:
def initFromLit(){
   System out println 'initlaize fromLib'
   100
}
fromLib = initFromLit()

//code in seperate module defined in: com.mycompany.myApplication:
m = 100

com.mycompany.library.fromLib = 101
m = 101

res = { "fromLib={com.mycompany.library.fromLib}, m={m}" } !

//com.mycompany.library.fromLib == 101
//res == "fromLib=100, m=101":
```

Above we see that `fromLib` has 'reverted' to it's initial state of `100`, whereas `m` is captured as `101`. Furthermore, `initFromLit()` will be executed twice (once on the initial use of fromLib and secondly on the copying into the spawned isolate) - so `'initlaize fromLib'` will be output to the console twice.

37.1.2 Values returned from isolates

Isolates return refs (discussed in detail in the Refs section) which hold values of the type returned from the code within the isolate. For example:

```
res = {"I am an Isolate"}!
deref String = res
```

Above `res` is of type `String:`. We later extract the result contained within `res` when we assign it to the `deref` variable which itself is not a ref.

In the case where our isolate does not return a value we can still assign its result to a variable as follows:

```
def hi() void{
   System out println "hello world"
}

res = hi()!
await(res)
//when we reach this point we know that hi has been executed fully
```

Above, `res` will be assigned an object of type `Object`. We then use the `await` keyword in order to wait for the aforementioned value to be set to `res` before continuing with execution. This is of course an optional step, we could just fire and forget about our `hi() call` scheduled for concurrent execution within an isolate.

37.1.3 Exceptions in isolates

If an isolate throws an exception which it doesn't catch within its call chain/body then this will be assigned to the returning ref if there is one. Parts of the code which rely upon that returned value will then be required to handle that exception. This approach is advantageous since it ensures that if an exception has occurred within the ref it is not lost within the program. For example:

```
def divOp(a int, b int) => a/b//can throw a div by zero, ArithmeticException

res = divOp(12, 0)!

deref int = res//ArithmeticException is thrown on the access of 'res' here!
```

Default exception handler

In cases where an isolate does not return a value from a ref the default exception handler shall handle it. This simply prints a stack trace to the console via `System.err` and permits execution of the program to otherwise continue uninterrupted.

```
def divOp(a int, b int) => a/b//can throw a div by zero, ArithmeticException

divOp(12, 0)! //no return value assignment!
//any thrown ArithmeticException will be printted to the console via System.err
```

37.1.4 Isolate Executors

In some circumstances it is appropriate to modify the way in which isolates are executed. To this end there are a number of special isolate executors available as part of the Concurnas Standard Library which can be used.

The general syntax for spawning isolates within an executor is as follows:

```
isolate ! ( executor )
```

The custom isolate executors as part of the Concurnas standard library are:

Dedicated Thread

The `concurrent.DedicatedThread()` executor will force execution of the isolate to take place within a dedicated worker with its own thread. This is particularly useful in cases where one is calling non-Concurnas JVM code (for instance, written in Scala or Java) which blocks on io (e.g. networking) or otherwise (e.g. infinite loops). Recall that blocking code would otherwise prevent other isolates from being executed in a timely manner via the usual mechanism, so this executor is a good choice in this instance.

If the called code within the isolate either directly or indirectly spawns new isolates, then these will be multiplexed as normal onto the root set of workers.

This executor should be used judiciously since creating dedicated threads doesn't scale as well as isolates multiplexed onto the usual set of shared workers. Though if one is calling code which is known to block, then this is usually the only, and best, option available.

Here is an example:

```
added = {10+10}!(concurrent.DedicatedThread())
```

It is not necessary to explicitly terminate a `concurrent.DedicatedThread()` executor since it is meant for once time use, it is automatically terminated after completion of an isolate executed via it. Note that this means that `concurrent.DedicatedThread()` executors cannot be reused.

Dedicated Thread Worker Pool

The `concurrent.DedicatedThreadWorkerPool()` executor creates a pool of workers on to which isolates may be executed via reference to the executor instance. It is similar to the aforementioned `concurrent.DedicatedThread()` executor in that it will force execution of the isolate to take place within a dedicated worker with its own thread. The difference here is that if the called code within the isolate either directly or indirectly spawns new isolates, then these will be multiplexed onto the new pool of dedicated workers.

The number of workers created is definable by specifying the `workerCount` default parameter of the `concurrent.DedicatedThreadWorkerPool()` executor constructor. If unspecified this value defaults to the number of core Concurnas workers spawned by the scheduler associated with the current running isolate in which the `concurrent.DedicatedThread()` executor is created.

Unlike the `concurrent.DedicatedThread()` executor, it is best practice to explicitly terminate the executor after one has finished using it as this will shut down the spawned worker threads which are no longer useful. This can be achieved by calling the `terminate` method on the executor. Note that, upon garbage collection, or termination of an implicit 'parent' executor (including the root executor), the worker pool will be implicitly terminated.

An example of usage:

```
pool = new concurrent.DedicatedThreadWorkerPool()
added = {10+10}!(pool)
pool.terminate()
```

37.1.5 Sync blocks

A `sync` block will ensure that all isolates created by code executed either directly or indirectly bt its body within the context of its executing isolate, have completed execution before permitting further execution by the aforementioned isolate. For example:

```
def gcd(x int, y int){//greatest common divisor of two integers
   while(y){
      (x, y) = (y, x mod y)
   }
   x
}

calc1 int:
calc2 int:

sync{
   calc1 = {gcd(8, 20)}!
   calc2 = {gcd(6, 45)}!
}

//calc1 and calc2 have now been set
```

Returning values from `sync`

Sync blocks may return a value:

```
def gcd(x int, y int){//greatest common divisor of two integers
   while(y){
      (x, y) = (y, x mod y)
   }
   x
}

calc1 int:
calc2 int:

complete = sync{
   calc1 = {gcd(8, 20)}!
   calc2 = {gcd(6, 45)}!
   "all done"
}

//complete == "all done" and calc1 and calc2 have now been set
```

37.2 Shared variables and classes

Isolates will make a copy of all of their non ref and non actor dependencies in order to ensure that no state is accidentally shared between instances. This default behaviour is always safe but sometimes is inappropriate, for instance when making use of Java objects which have their own concurrency control, and for read only data structures (especially when they are large since the copy operation performs a deep copy). This can be overridden by making use of the `shared` keyword.

We can declare a new variable assignment as being shared and use it as normal within an isolate as follows:

```
shared numbers = [1, 2, 3, 4, 5, 6]

complete = {numbers += 1; true}! //vectorize add one to each element
```

```
await(complete)//wait for iso to complete

// numbers == [2, 3, 4, 5, 6, 7]
```

Method variables and class fields may also be tagged as shared.

37.2.1 Caveats

Only the value of the variable is shared, the variable itself is not. As a result, re-assigning a shared variable to a different value will not result in the new value being shared. Consequently non array primitive types may not be tagged as shared.

Top level global variables, at module level, may be declared with an initial value as shared, but care should be taken when assigning them values at module level (both directly or indirectly via a function/method etc). Since, as we have previously seen, top level module code is run on import by an isolate, this has the effect of wiping out whatever was previously stored within the shared variable every time an isolate which uses any aspect of the module is executed... (thus defeating the point of the variable being shared). Here is an example of what to watch out for:

```
//in module com.myorg.code.py
public shared sharedvar = new Integer(0)

sharedvar = 26 //top level module core assigning a value to sharedvar - dangerous

//in module: com.myorg.othercode
from com.myorg.code import sharedvar

defmymethod(){
   sharedvar = 50
   [sharedvar {sharedvar}!]//when the iso is executed sharedvar will be 'reset'
       to 26 within the iso
}

defmymethod()// [50 26:]
```

Removing the `sharedvar = 26` line will have the effect of allowing us to preserve the assigned value of `50` within the `defmymethod` method when the isolate: sharedvar! is run. i.e.

```
//in module com.myorg.code.conc
public shared sharedvar = new Integer(0)

//in module: com.myorg.othercode
from com.myorg.code import sharedvar

defmymethod(){
   sharedvar = 50
   [sharedvar {sharedvar}!]
}

defmymethod()// == [50 50:]
```

The shared tag is ignored when shared variables are assigned to, or read from refs... they are still copied, though refs themselves are naturally shareable between isolates.

37.2.2 Shared Classes

Classes may be tagged as shared, this has the same effect as tagging a variable as shared but applies to all instances of objects of the shared class (including subclasses). See the Shared Classes section for more information.

37.3 Actors

Concurnas supports concurrency through Actors. The actor model of concurrency itself has existed since the 1970's and is seeing a resurgence in recent years due to the proliferation of multicore computer architectures and the fact that, compared to the shared state model of computation, offers a simpler and more intuitive model

Actors in Concurnas behave like classes in that instances of actors can be created with constructors and methods may be called upon those instances. The difference compared to classes however is that actors can be shared between isolates, they are not copied. They are able to do this because they provide concurrency control on all of their method invocations to the effect of turning execution on an actor into a single threaded operation. This makes them ideal for implementing the likes of i/o operations (especially those such as writing to disk where contention from concurrency would actually reduce throughput).

Under the hood actors run within their own isolate and like isolates, actors (which run in their own dedicated isolate), will make a copy of all input arguments to any of their constructors or methods that are invoked.

One of the nice things about the way in which Actors have been implemented in Concurnas is that one does not have to give up the advantages of static typing in order to use them.

Actors are defined in a very similar way to classes and have two variants, untyped actors and typed actors which we explore here.

37.3.1 Untyped Actors

Let's create an untyped actor:

```
open actor MyActor{
  -count = 0
  def increment(){
    count++
  }
}
```

We say this actor is 'untyped' since it does not explicitly operate upon another type, we will examine the concept of a 'typed' actor which does operate on another type in detail in the Typed Actors section. When defining actors we may use the same syntax as classes - including generics, constructors, class definition level fields, fields, inheritance, traits etc.

The actor manages its own concurrency to the extent of sequentially executing concurrent requests made of it. Method invocations seem to behave as normal from the perspective of authoring code, i.e. the caller invokes the method and a value is returned if appropriate - as normal. However, at runtime behind the scenes, execution is being requested of the actor in its own isolate, the caller isolate pauses execution until it receives a response from said actor. When the method has been executed by the actor it will notify the caller which can then can then carry on with execution as normal. This is different from normal method invocation where there is transference of stack frame control into the method and where the invoking isolate is responsible for code execution.

We can create an instance of the aforedefined actor, and call some methods upon it as follows:

```
ma = new MyActor()
```

```
ma.increment()//just like normal method call will block until exeuction is
    complete

sync{
   {ma.increment()}!//called concurrently
   {ma.increment()}!
}

count = ma.count//count == 3
```

The above would not be possible if were to an ordinary object in place of our actor since they do not have any concurrency control and as such must be copied into isolates in order to prevent the state of the object being *accidentally* shared. Never the less as we can see above, one of the nice things about actors in Concurnas is that they can be used seamlessly like ordinary objects.

Like classes, typed actors may inherit from other actors:

```
actor Decount < MyActor{
   def decrement(){
      count--
   }
}
```

One restriction placed upon actors is that non private fields of actors may not be accessed. Instead, getters and setters should be used (which can be easily automatically generated in Concurnas see the Setters and Getters section).

37.3.2 Typed Actors

Typed actors enable us to create an actor which wraps around an instance of an ordinary object. This is ideal for use in cases where we're using pre existing code, perhaps from user defined libraries or the JDK, and wish to create an actor instance of them.

For example let's say we have a predefined class we wish to use as an actor:

```
class MyCounter(-count int){
   def increment(){
      count++
   }
}
```

We can now define and use our typed actor through the aid of the *of* keyword as follows:

```
actor MyActor of MyCounter(0)

//let's use our new typed actor:
ma = new MyActor()

ma.increment()

sync{
   ma.increment()!
   ma.increment()!
}

count = ma.count//count == 3
```

Our typed actor defined above, `MyActor`, has created a stub method instance of all of the methods exposed within our ordinary class: `MyCounter` such that they can be seamlessly called as if `MyActor` were an instance of `MyCounter`. As and when we come to create an instance of `MyActor`, the typed actor will, behind the scenes, create a `MyCounter` instance (accessible as the variable `of` - see the The `of` keyword section) to which it will direct method invocations in a serial manner. It's for this reason that all the constructors defined on the wrapped `MyCounter` instance are also callable on the `MyActor` instance.

Instances of generic classes may be created as above with generic qualification or that qualification differed to the creator of the actor:

```
actor StringListActor of java.util.ArrayList<String>//qualified

open actor ListActor<X> of java.util.ArrayList<X>//differed
```

As with untyped actors, actors may inherit from other actors or even be defined as abstract:

```
actor ChildStringListActor < ListActor<String>
abstract actor AbstractListActor<X Number> < ListActor<X>
```

Typed actors implement all traits defined by the class of which it's an actor of. Hence the following holds:

```
trait MyTrait1
trait MyTrait2
class ChildClass ~ MyTrait1, MyTrait2

actor MyActor of ChildClass

//
ma = new MyActor()

assert ma is MyTrait1 //true!
assert ma is MyTrait2 //true!
```

Typed actors may not be created of actors. So the following is not valid:

```
actor InvalidActor of ChildStringListActor//not valid!
```

The `of` keyword

Like untyped actors, typed actors can define methods:

```
actor MyActor of MyCounter(0){
   def decrement(){
      of.count--
   }
}
```

In the example above we can see that the `of` keyword is used in order to refer to the object which our defined actor is referring to. Also, just like the `this` keyword the `of` keyword can usually be inferred. Thus it would be acceptable to write the above code as:

```
actor MyActor of MyCounter(0){
   def decrement(){
      count--
   }
}
```

Calling actor methods

Sometimes, we are obliged to create a method on a typed actor having the same signature as that of the class to which it is acting upon. In these circumstances when a normal method invocation is observed on the actor instance, Concurnas will infer that the actor method upon the class being acted upon should be invoked, instead of that defined on the actor itself. In order to force execution of the actor version of said method, one can use the `:` operator as follows:

```
actor MyActor of MyCounter(0){
   def increment(){
      of.increment()
      of.increment()
   }

}

def doings(){
   ma = new MyActor()
   ma:increment() //call version of method defined on MyActor

   "ok " + ma.count
}
```

37.3.3 Default Actors

Another way to create a typed actor is to simply use the actor keyword in place of or in addition to the new keyword when creating an object. For example:

```
class MyCounter(-count int){
   def increment(){
      count++
   }
}

inst1 = actor MyCounter(0)   //create a new actor on MyCounter
inst2 = new actor MyCounter(0) //create a new actor on MyCounter
```

We can now use `inst1` and `inst2` as actor instances.

37.3.4 Instances of and casting actors

Typed actors may only be compared with other actors via the `is` and `as` keywords, but they are reified types. Thus we're able to write code such as the following:

```
actor MyUntyped {
   def something() => "hi"
}

xxx = new MyUntyped()
o Object = xxx

assert o is actor //true
assert o is MyUntyped //true
res = (o as MyUntyped).something() //res == "hi"
```

And when using generics this can be very helpful:

```
class MyClass<X>(~x X)
actor MyActor<X> of MyClass<X>

xxx = new MyActor<String>('hi')
o Object = xxx
assert o is MyActor<String> //true
```

37.3.5 Shared variables and classes

Actors will ignore the shared nature any variables which have been marked as shared when they are passed to an actor method. That is to say that they will be copied as par normal irregardless of the fact that they have been marked as shared. This is not the case for constructor invocation however, when a actor constructor is invoked and shared variables passed as arguments are treated as being shared and are not copied.

The behaviour of shared classes remains unchanged, i.e. if an instance object of a class marked as shared is passed to either a actor constructor or method, it will not be copied as par usual.

Shared Method Parameters for Actors

Recall that actors operate within their own dedicated Isolate. As such, mutable state is copied into them on execution. In order to suppress this behaviour, e.g. for classes having transient fields, for a method parameter (as part of a method of an actor), mark the input parameter as being shared:

```
actor MyActor(){
   def callme(shared aparam WithTransientFields){
      //... work as normal
   }
}
```

37.3.6 Actor gotcha's

There are a few gotcha's which one must factor into one's design of software when using actors.

Spawning isolates directly within actors

Isolates spawned directly by **Untyped actors** run within the context of the spawning actor. This has the effect of forcing the isolate, for all intents and purposes, to run non-concurrently. For example:

```
actor MyActor{
   -an int = 99

   def spawndoer(){
      done := {an++; true}!//this will be run in the context of the actor
      done:
   }
}


anActor = new MyActor()
await(anActor.spawndoer())
got = anActor.an //returns 100 as expected
```

Isolates spawned directly by normal classes, which have been captured by typed actors are run upon the actor as normal, they are not run within the context of the spawning actor, but rather a copy of the captured class. Here is an illustration of this behaviour:

```
class Myclass{//normal class
   -an int = 99

   def spawndoer(){
      done := {an++}!
      await(done)
   }
}

anActor = actor Myclass()//normal class captured by typed actor
anActor.spawndoer()//spawns an isolate to perform the operation
got = anActor.an //an is still 99, not 100!
```

37.4 Refs

Standard thread based models of communication in most programming languages require state to be shared and for access to it to be explicitly controlled in critical sections. Generally this is implemented on a pessimistic basis, in assuming that shared data access will be contentious, by requiring the use of explicit read-write locks or synchronized blocks on our critical sections. Engineering concurrent applications wish shared memory is one of the most challenging aspects of modern software engineering.

Concurnas is different. Concurnas introduces the concept of a ref. This is an entity which can be freely shared between isolates which refers to and safely manages another value. refs provide an optimistic approach towards concurrency in that they provide communication of state via message passing. Message passing is widely accepted as an easier though equally capable model of concurrency to work with as we do not need to spend time coding read-write locks or synchronized blocks, rather than time can be spent working on our core objectives.

As the value of a ref can change over time this allows us to build reactive components upon them (such as `await`, `every` and `onchange` explored below) which again opens up a far more natural way of programming than what most developers are used to. Concurnas provides transactions for the times when we need to change one or more refs on an atomic basis.

37.4.1 Creating refs

There are a number of ways in which refs can be created:

```
aref1 int: //defined but unassigned local ref of type int
aref2 = new int://defined but unassigned local ref of type int
aref3 int := 21 //defined and assigned local ref of type int
aref4 := 21 //assigned local ref of inferred type int
aref5 = 21: //assigned variable of inferred type int: with the right hand side
    being a ref creation expression
aref6 = {21}! //assigned variable of inferred type int: - right hand side being
    a ref returned from an isolate - this is the most computationally expensive
    variety of ref creation - if you find yourself doing this to create simple
    refs from expressions that don't require concurrent execution, consider
    using the preceding form
```

Refs can be created of any type - even refs themselves (see below refs of refs)! Refs may not be of nullable types.

All the refs above are of type `int:`. As can be seen above the key when creating refs is the use of `:` postfixed to a type or prefixed to the assignment operator `=` - this tells the compiler that we wish to create a ref type.

37.4.2 Using refs

Just like normal variables, we can assign a value to the ref at any point during or post declaration. For example:

```
aref1 int: //defined but unassigned local ref of type int
aref1 = 100 //assigned just like normal!
```

When we have a method or other location where a ref is expected as an input type, and the input we have provided is of that refs component type, Concurnas will automatically create a ref to hold that value and set it as appropriate. For example:

```
def expectsRef(a int:) int => a:get()

expectsRef(12)//12 is automatically converted into a ref, i.e. expectsRef(12:)
```

Ref variable dereferencing

When we wish to perform an operation on a ref variable, or pass a ref to a function which is not expecting a ref - the ref will be automatically unassigned. If no value has yet been assigned to the ref further execution will be blocked until a value has been assigned.

Type wise, when unassigning a ref with type `X:`, the type of the value returned will be `X`. For `int:` this is `int`. For example:

```
a int:

{a = 99}!//execute our ref assignment in a separate isolate

b int = a//a is unassigned when our isolate spawned above complete's execution
```

refs are dereferenced not just on variable assignment, but at any place where they are used. For example on function invocation:

```
def takesInt(a int) => a

res = {a = 99}!

takesInt(res)//res is dereferenced
```

If we wish to prevent this unassignment we can use the `:` operator when referencing the variable. For example:

```
a int: = 12
b = a:
c := a

//the &== operator tests for equality by object reference and not by object
    value as the == operator does
assert b: &== a: //a and b are references to the same ref
assert c: &== a: //a and c are references to the same ref
```

The : operator may also be used in order to access methods on the ref itself. For instance the `hasException` method will return true if an exception has been set on the ref:

```
aref = 12:
aref:setException(new Exception("uh oh"))

hase = aref:hasException() //returns true, an exception has beenset
```

37.4.3 Notable methods on refs

All refs are subtypes of the class: `com.concurnas.bootstrap.runtime.ref.Ref`. This class exposes a number of occasionally useful methods of note that are accessible via the : operator:

- `def isSet()boolean` - returns true if a ref has been closed, an initial value or an exception set (non blocking method).
- `def waitUntilSet()void` - returns when a ref has been closed, an initial value or an exception set (blocking method).
- `def close()void` - close a ref, no further values may be assigned.
- `def isClosed()boolean` - returns true if a ref has been closed (non blocking method).
- `def setException(e Throwable)boolean` - set an exception on a ref.
- `def hasException()boolean` - returns true if a ref has been closed (non blocking method).

When creating a ref via the aforementioned methods one is creating a ref of type: `com.concurnas.runtime.ref.Local<X>` where X is the reified type of the ref. In other words:

```
aref1 int:
//is equivilent to:
bref1 com.concurnas.runtime.ref.Local<int>
```

Instances of `Local<X>` provide the following additional methods:

- `set(x X)void` - set the latest value of the ref. (equivalent to assignment)
- `get() X` - get the latest value of the ref. (blocking method).
- `getNoWait()X?` - get the latest value of the ref. Value will be returned as `null` if no initial value has been set (non blocking method).
- `get(withNoWait boolean)X?` - get the latest value of the ref. May return `null`. (optionally blocking method).

37.4.4 Arrays of refs

Arrays of refs may be created in the usual manner:

```
arrayofrefs1 = [1: 2: 3:]
arrayofrefs2 = new Integer:[10]
arrayofrefs3 = new Integer:[2, 2](1)//with array initalizer
```

Arrays of refs are of type: `com.concurnas.runtime.ref.LocalArray<X>` where `X` is the ref type (`Integer:` above).

37.4.5 Ref variable reassignment

The ref variables themselves may be reassigned by ensuring that the assignment operator is prefixed with :, for example:

```
aref int: = 100
```

```
bref int: = 200
assert aref <> bref //values not equal
assert aref: <> bref://refs not equal

aref = bref //assign the value of bref to aref
assert aref == bref //values equal!
assert aref: <> bref://refs not equal


aref := bref //assign the ref itself: bref to aref
assert aref == bref //values equal!
assert aref: == bref://refs equal!
```

37.4.6 Refs are reified types

refs are reified types hence Concurnas knows what type a ref is holding at runtime. This allows us to safely write code such as the following:

```
def checkIfIntRef(something Object) boolean => something is int:
```

37.4.7 Closing refs

refs can be put into a closed state, after which no further values may be assigned to them, and any reactive components which use them will no longer consider them to be something to watch for changes to react to after notification.

```
aref = 10:
aref:close()
assert aref:isClosed()//aref is now closed
```

Any reactive components that are registered to watch for changes to a ref will be invoked upon said ref being closed, for example:

```
aref := 10

isclosed boolean:
onchange(aref){
   if(aref:isClosed()){
      isclosed = true
   }
}
aref:close()

await(isclosed)
```

Above, we create a ref `isclosed` which is only set by our `onchange` block after `aref` has been closed.

37.4.8 Custom refs

In 99% of situations the built in ref types are all we need to solve problems. But sometimes it can be useful to define our own custom refs (e.g the `gpus.GPURef` ref type). To do this all we need to is subclass the `com.concurnas.runtime.ref.Local` class. However, there are a few caveats to be remembered when building custom refs which we shall explore here.

Refs, being reified types, require an additional initial argument, of type `Class<?>[]`, be added to all defined constructors to hold the reified type information generated seamlessly by Concurnas at compilation time. When constructing refs this extra argument does not need to be populated. For example:

```
class CustomRef<X>(type Class<?>[], ~extraArg int) <
    com.concurnas.runtime.ref.Local<X>(type){
   this(type Class<?>[]) => this(type, 99)
}
```

No argument constructor

Additionally, in order for custom refs to be implicitly creatable, they should specify a no argument constructor. Failure to do so will result in a runtime exception if an implicit creation is attempted.

```
class CustomRef<X>(type Class<?>[], ~event int) <
    com.concurnas.runtime.ref.Local<X>(type)
class CustomRefWithNoArg<X>(type Class<?>[], ~event int) <
    com.concurnas.runtime.ref.Local<X>(type){
   this(type Class<?>[]) => this(type, 12)
}

def fail(a Object) Object:CustomRef{
   return null//implicitly create a CustomRef to hold null - which will fail
}
def ok(a Object) Object:CustomRefWithNoArg {
   return null//implicitly create a CustomRefWithNoArg
}

ok("123")//this will return a Object:CustomRefWithNoArg object
fail("123")//this will throw a runtime exception because there is no no argument
    constructor defined for CustomRef
```

Using custom refs

Custom refs may be created by appending the desired custom ref constructor (without the initial, additional, reified type array) after the `:` operator. For example:

```
class CustomRef<X>(type Class<?>[], ~extraArg int) <
    com.concurnas.runtime.ref.Local<X>(type){
   this(type Class<?>[]) => this(type, 99)
}

inst1 = int:CustomRef(12)//insatnce of our CustomRef
inst2 = int:CustomRef()//insatnce of our CustomRef
```

Alternatively, when we are using the declaration with assignment form of variable declaration, a custom ref may be implicitly constructed and the appropriate set method matching the assignment value called. For example:

```
inst1 int:CustomRef = 12//insatnce of our CustomRef with implicit contruction

//above is equivilent to:
inst2 int:CustomRef; inst2 = 12
```

The above is equivalent to:

```
inst1 int:CustomRef;
inst1 = 12
```

RefArrays

Ref arrays: `com.concurnas.runtime.ref.RefArray<X>` are a handy custom ref type included in the Concurnas runtime. RefArrays provide an efficient means of providing ref like behaviour on arrays via a fixed length array like structure.

Notable methods on `RefArray<X>` are:

- `get() X[]` - get the latest value of the ref. (blocking method).
- `get(withNoWait boolean)X[]` - get the latest value of the ref. May return `null`. (optionally blocking method).
- `getNoWait()X[]?` - get the latest value of the ref. `null` if unset. (non blocking method).
- `set(x X[])void` - set the latest value of the ref.
- `get(i int)X` - get the latest value of the ref at index i. (blocking method).
- `get(i int, withNoWait boolean)X?` - get the latest value of the ref at index i. May return `null`. (optionally blocking method).
- `getNoWait(i int)X?` - get the latest value of the ref at index i. `null` if unset. (non blocking method).
- `put(i int, x X)void` - set the latest value of the ref at index i.
- `getSize()int` - Returns the size of the `RefArray`.
- `modified()List<Integer>` - Returns a non empty list of indexes corresponding to the most recently changed elements of the `RefArray` in a transaction only if called within the context of a reactive component: `every`, `onchange`, `await`, `async`.

Example usage:

```
import com.concurnas.runtime.ref.RefArray

xx int:RefArray = [Integer(1) 2 3 4]
xx[0] := 99
elm = xx[3]

//xx == [99 2 3 4]
//elm == 4
```

An alternative to RefArrays are arrays of refs.

37.4.9 Nested refs

The Concurnas ref syntax supports nesting, i.e. refs of refs. Though this feature is of limited usability there are occasions where it can be incredibly useful. The principles are straightforward. Let's look at some examples:

```
aref1 int:: //declare a ref of a ref
aref2 int:: = 53 //declare and assign a value to the ref of a ref
aref3 = 53:: //infer type as ref of ref: int::

anint1 int = aref2 //extract value helf by ref of ref
anint2 int = aref2::get() //extract value helf by ref of ref
nestedref int: = aref2:get()//extract ref held by ref
```

Above we can see that when we declare nested refs we need only append the ref operator to the element we are nesting. In the case of a type this is: `int::` and when calling methods on nested refs:

aref2:get(). Concurnas will automatically ref/unref a value to the appropriate level of nesting in cases where possible:

```
def expectsRefOfRef(arefofref int::) => arefofref

aref1 = 12:
expectsRefOfRef(aref1)//equvilent to: aref1: (which is of type int::)
```

Refs of custom refs, and vice verse may be created and used in a similar manner:

```
class CustomRef<X>(type Class<?>[], ~extraArg int) <
    com.concurnas.runtime.ref.Local<X>(type)

custRefOfRef int::CustomRef = 16//CustomRef holding a ref of type int
```

37.5 Reactive programming

Concurnas provides four major components for supporting reactive programming. The reactive elements are defined as: `await`, `every`, `onchange` and `async`. These elements allow us to write code in a natural reactive style and allows our programs to perform operations contingent upon a change to one or more monitored refs.

37.5.1 `await`

The `await` keyword takes a series of expressions evaluating to one or more references and will block execution until at least one of those references has been set. For example:

```
complete boolean:
{
   //some complex calculation..
   complete = true
}!//spawn an isolate to execute the above concurrently.

await(complete)//pause further execution until complete is set
```

`await` guards

An optional guard condition may be included within the `await` statement. If so the `await` statement behaves as before but will only return true when the guard condition resolves to true:

can be declared within the `every` or `onchange` block

```
execount int: = 0
for(a in 1 to 10){
   //some complex calculation..
   execount++
}!//spawn an isolate to execute the above concurrently.

await(execount; execount==10)//pause further execution until execount == 10
```

The execution of the `await` statement (including guard expression) takes place within its own isolate.

37.5.2 every and onchange

Concurnas provides two reactive blocks which are central to most reactive programs. These are the every and onchange blocks. They both take a ref (or group of refs) and will monitor this ref(s) for changes. Upon a notification of a change occuring the body of the block shall be executed.

Like the await statement, every and onchange blocks persist within their own isolate, when they react to changes to refs they execute their code body within that isolate. Hence upon their successful initialization they immediately return the flow control to the invoking isolate.

The difference between the every and onchange blocks is that the every block will be initially triggered upon initialization of the block if any of the monitored refs has an initial value set. The onchange block on the other and will only be triggered if a change is made to a ref post initialization.

Here is an example of a every and onchange block:

```
lhs int:
rhs int:
result1 int:
result2 int:
every(lhs, rhs){}
   result = lhs + rhs
}

onchange(lhs, rhs){}
   result2 = lhs + rhs
}
```

every and onchange, just like all blocks in Concurnas, supports the single line block syntax, i.e.:

```
lhs int:
rhs int:
result1 int:
every(lhs, rhs){}
   result1 = lhs + rhs
}
//can be written as:
plusop2 = every(lhs, rhs) {lhs + rhs}
```

every and onchange blocks will be triggered every time a change is made to a monitored ref. Code executed within their body which references the ref(s) that were changed in the transaction which caused that trigger with have their values locked at the value which was written in the transaction, other refs (which may or may not be monitored) will reflect the latest state of the ref in question as normal. An example:

```

aref int:
onchange(aref){
   System.out.println("latest value of aref: {aref:get()}")
}

aref=1
aref=2
aref=3

//will output to System.out:
//1
```

```
//2
//3
```

every and onchange blocks will cease to monitor refs or be executable (and hence become eligible for garbage collection) when all of their monitored refs are in a closed state or are otherwise out of scope. Additionally if the block is explicitly returned from via the return keyword, then it will also cease to be executable - see the Returning from every and onchange section for more details.

Declarations within every and onchange blocks

It is possible to assign and declare a ref within a every or onchange block as follows:

```
def retRef() int: => 12
itmchanged = onchange(watch = retRef()) { watch }
```

Returning from every and onchange

every and onchange blocks may return values, for example:

```
lhs int:
rhs int:
plusop2 = every(lhs, rhs){lhs + rhs}
```

Since every and onchange blocks operate within their own isolate if an exception is thrown and left uncaught within that block, the exception will be set on the return value. In the above example plusop2. After this exception has occurred no further execution of the every and onchange block will take place. The effect is the same in cases where there is no return value, the difference being that the exception will be handled by the isolate default exception handler - which will simple output a stack trace to the System.err console output stream.

If the return keyword is explicitly used in order to return from a every or onchange block then this has the additional side effect of terminating future execution of the every or onchange block.

every and onchange parameters

every and onchange blocks may have the following optional parameters attached to them by specifying one post ; after the refs to be monitored:

- onlyclose - every and onchange blocks tagged with this parameter will only react to refs being closed.

Example:

```
lhs int:
rhs int:
howmanyclosed = every(lhs, rhs; onlyclose){changed.getChanged().length}
```

Implicit ref monitoring

Concurnas will automatically infer which refs to monitor for an every or onchange block if none are provided in the definition:

```
x := 10
y := 5

result = onchange { x + y } //x and y will automatically be monitored for changes
```

Concurnas will monitor only those refs defined outside the scope of the onchange/every block. Refs referred to indirectly within the body of methods/functions will not be monitored. For example:

```
x := 10
def xSquared() => x ** 2

result = onchange{ xSquared()}//x will NOT be automatically monitored
```

In fact, in the above example, this will not compile because no refs to monitor have been specified (nor can be inferred) in the onchange definition.

37.5.3 Compact `every` and `onchange`

Concurnas provides a more compact way to define `onchange` and `every` blocks when they are used on the right hand side of an assignment. In essence, one can replace `onchange` with <- and `every` with <=.

```
x := 10
y := 5

res1 <-(x, y) x + y //onchange with dependencies explicitly defined
res2 <=(x, y) x + y //every with dependencies explicitly defined

res3 <- x + y //onchange with dependencies implicitly defined
res4 <= x + y //every with dependencies implicitly defined
```

This compact syntax in particular makes performing reactive computing with refs very convenient.

37.5.4 Async

The `async` block enables us to define a collection of related `every` and `onchange` blocks with an optional `pre` and `post` block. `async` blocks perform execution within one dedicated isolate for all `every` and `onchange` blocks. Like `every` and `onchange` blocks, `async` blocks will permit the initializing isolate to continue with execution post initialization.

The optional `pre` block enables us to define and initialize state which is accessible only to the `every` and `onchange` blocks of the `async` block. The optional `post` block enables us to execute code when the monitored refs of all the `every` and `onchange` blocks are in a closed state and the `async` block terminated.

Example:

```
finalCount int:
tally1 = 2:
tally2 = 2:

async{
   pre{
      count = 0
   }

   every(tally1){
      count += tally1
   }

   onchange(tally2){
      count += tally2
   }

   post{
      finalCount = count
   }

}

tally1 = 9; tally1 = 10; tally1 = 10
tally2 = 45; tally2 = 3; tally2 = 53

tally1:close(); tally2:close()

await(finalCount)

//finalCount == 132
```

Async returning values

Like single every and onchange, async blocks may return values. Either all of the every and onchange blocks must return a value or one value must be returned from the optional post block.

Example of all the every and onchange blocks returning values:

```
finalCount int:
tally1 = 2:
tally2 = 2:

lastproc = async{
pre{
count = 0
}

every(tally1){
count += tally1
count//return this value
}

onchange(tally2){
count += tally2
count//return this value
}

post{
finalCount = count
}

}

tally1 = 9; tally1 = 10; tally1 = 10
tally2 = 45; tally2 = 3; tally2 = 53

tally1:close(); tally2:close()

await(finalCount)
```

```
//finalCount == 132
```

Example of the `post` block returning a value:

```
tally1 = 2:
tally2 = 2:

finalCount = async{
pre{
count = 0
}

every(tally1){
count += tally1
}

onchange(tally2){
count += tally2
}

post{
count//return this value
}

}

tally1 = 9; tally1 = 10; tally1 = 10
tally2 = 45; tally2 = 3; tally2 = 53

tally1:close(); tally2:close()

await(finalCount)
//finalCount == 132
```

37.5.5 Reacting to multiple refs

In the previously examined examples of reactive programming we have only considered the case of monitoring individual refs for changes. However, Concurnas supports monitoring of multiple refs at the same time. The following groupings of refs may be monitored in any of the reactive elements: `await`, `every`, `onchange` and `async`:

- **Individual refs** - Multiple individual refs may be referenced in a comma separated list:

  ```
  lhs int:
  rhs int:
  plusop = every(lhs, rhs){ lhs + rhs }
  ```

  Multiple refs may be declared and assigned within the reactive element provided that the are in the comma seperated list, for example:

  ```
  lhs int:
  rhs int:
  plusop = every(a = lhs, b = rhs) { a + b}
  ```

- **Arrays of refs** - Multiple individual refs may be referenced in a array of refs:

```
lhs int:
rhs int:
watchAr = [lhs rhs]
plusop = every(watchAr){ lhs + rhs}
```

The refs monitored are those present within the array upon creation of the reactive element in question. If the contents of the array changes post element creation these changes will not be included in the set of refs to be monitored. For monitoring of a dynamic set of refs, a `ReferenceSet` object is more appropriate.

- **Lists, maps or sets of refs** - Multiple individual refs may be referenced in a list or set of refs or the key set of a map:

```
lhs int:
rhs int:
watchAr list<int:> = [lhs, rhs]
plusop = every(watchAr) { lhs + rhs}
```

As with likes of arrays of refs, only the refs present within lists, maps or sets of refs at the point of element creation will be included for monitoring. For monitoring of a dynamic set of refs, a `ReferenceSet` object is more appropriate.

- **ReferenceSets** - Instances of `com.concurnas.runtime.cps.ReferenceSet` can be used in order to monitor a dynamically changeable set of refs:

```
from com.concurnas.runtime.cps import ReferenceSet
lhs int:
rhs int:

liveSet = new ReferenceSet()
liveSet.add(lhs)
liveSet.add(rhs)

numChanged = every(liveSet) { changed.getChanged().length }

liveSet.remove(lhs)
//our every expression will continue to monitor only 'rhs' for changes...
```

Either closing or removing a ref from the monitored `ReferenceSet` will result in it being no longer being monitored by the reactive element.

- **A mixture of the above** - Multiple instances of the aforementioned groupings of refs may be used in a reactive element provided that they are presented as a comma separated list:

```
lhs int:
rhs int:
another1 int:
another2 int:
another3 int:
watchAr = [another1 another2 another3]
numChanged = onchange(lhs, rhs, watchAr) { changed.getChanged().length }
```

A reactive element listening to more than one ref will be awoken for execution if any of refs it monitors is changed.

There are some special considerations and tools to bear in mind when working with reactive elements monitoring more than one ref:

changed

The `changed` keyword can be used in order to obtain the set of refs which have been changed as part of the transaction which has caused the reactive element to be activated. All changes to refs, even to a single ref outside of a transaction will result in a transaction being created. The changed keyword itself will return the transaction object (of type `com.concurnas.bootstrap.runtime.transactions.Transaction`) holding the changed set of refs, accessible by calling the `getChanged()com.concurnas.runtime.ref.LocalArray<com` method.

```
lhs int:
rhs int:
numChanged = onchange(lhs, rhs) { changed.getChanged().length }
//numChanged is a ref containing the number of refs changed in a transaction
    causing onchange to be triggered
```

Reactive Element termination

When a reactive element is monitoring more than one ref for changes it will be terminated when all of those refs are closed. This can be problematic when one is using a `com.concurnas.runtime.cps.ReferenceSet` in order to dynamically react to changes in refs since when they happen to be all closed and/or removed from the monitored `ReferenceSet` instance the reactive element will be closed for future execution. We must be mindful of this behaviour and if our use case dictates that there are cases where we must remove or close all refs being monitored in a `ReferenceSet` then we must be careful to recreate a new reactive element if we later come to have more refs which need to be dynamically monitored.

37.6 Transactions

Concurnas supports software transactional memory via the `trans` block keyword. This allows us to make changes to one or more refs and have those changes visible outside of our transaction on an atomic basis. `trans` blocks behave like normal blocks in so much as they operate within the same isolate as the invoker.

Recall that a reactive element listening to more than one ref will be awoken for execution if any of refs it monitors is changed in a transaction. For this reason the following code will result in our `onchange` block being invoked twice:

```
lhs int: = 100
rhs int: = 100
onchange(lhs, rhs) {//onchange will always act on the latest known value
   System.out.println("sum: {lhs + rhs}")
}

lhs -= 10
rhs += 10

//will output:
//sum: 200 or 190
//sum: 200
```

Further more, there is some non determinism here in that the first value output may or may not include the change which is made to the `rhs` ref since this may not have have occurred yet when the `onchange` is triggered

We can change this behaviour by combining our changes to `lhs` and `rhs` in one transaction, in a `trans` block:

```
lhs int: = 100
rhs int: = 100
onchange(lhs, rhs) {
   System.out.println("sum: {lhs + rhs}")
}

trans{
   lhs -= 10
   rhs += 10
}

//will output:
//sum: 200
```

The above will always provide a consistent result of a single output value from our `onchange` block.

37.6.1 Transaction execution

Transactions are guaranteed to complete execution atomically or throw an exception. Changes occurring within an exception are not visible outside of that transaction until the entire transaction has completed execution.

Transactions operate on an optimistic basis in assuming that ref contention, where a ref is changed by a different entity other than the transaction, is an unlikely occurrence. Should it occur however, the transaction will internally 'roll back' all the refs which it has changed and try execution again and again until it succeeds.

It's for this reason that non mutable non-ref changing operations such as i/o, calling methods on mutable objects or changing variables from outside the transaction should be avoided, since these are not rolled back should ref contention occur (and are sometimes, such as in the case of i/o, impossible to roll back anyway). If we do need to perform non-ref related operations within a transaction we need to ensure that they are idempotent operations.

37.6.2 Returning from transactions

Transactions, like any other block in Concurnas, may return a value:

```
account1 int: = 1000
account2 int: = 120

takeoff = 10

prevBal = trans{
   prev = account1
   account1 -= takeoff
   account2 += takeoff
   prev
}
//prevBal == 1000
```

37.6.3 Nested Transactions

Transactions may be nested. Any changes to refs taking place within an inner nested transactions will be visible to any outer nesting transactions upon completion, and those changes, along with those made in any nesting transactions will also be visible outside the transaction once the outermost layer of transaction has been completed. For example:

```
acc1 = 10000:
acc2 = 10000:
acc3 = 999:

trans{
   trans{
      acc3++
   }
//change to acc3 is visible below but at this point not outside the trans
   acc1 -= acc3
   acc2 += acc3
}
//now all changes to acc1, acc2 and acc3 are visible
//acc3 == 1000, acc1 == 9000, acc2 == 11000
```

37.6.4 Change sets

As we have already seen when it comes to reactive elements, they will be triggered upon any of their monitored refs being changed. However, as a result of a transaction, from the perspective of the reactive element, it is possible for the change set to include rfs which are not monitored as part of the reactive elements group of interest. This should be borne in mind when building algorithms which inspect the `changed` set of a reactive element. For example:

```
acc1 = 10000:
acc2 = 10000:

whatchanged = onchange(acc1) { changed.getChanged().length }

trans{
   acc1 -= 10
   acc2 += 10
}
//whatchanged == 2
```

Above, we see that `whatchanged` will be set to `2`. This may seen like a bug at first because the `onchange` block is only monitoring one element, but this is expected as the changed set includes changes to two refs within the transaction changing them.

37.7 Temporal Computing

A common pattern in concurrent systems engineering is to want to have some for of time based trigger, "wait 10 seconds then do x" or "do y every 3 seconds etc" or "perform this action at this certain time". At Concurnas we refer to this as temporal computing.

Temporal computing is supported via the Pulsar library found at: `com.concurnas.lang.pulsar`. This library allows us to schedule activities to take place in the future after a certain amount of time has elapsed. It also allows us to schedule tasks for repetition.

The pulsar provides two implementations of the `com.concurnas.lang.pulsar.Pulsar` trait which presents five methods. These all return a ref which will be updated with the current time at the point of event trigger:

- `after(after java.time.Duration)java.time.LocalDateTime:` - Fire off after the specified duration.
- `schedule(when LocalDateTime)OffsetDateTime:` - Fire off at a certain date time in the future.
- `repeat(after Duration)LocalDateTime:` - Trigger with a certain degree of frequency.
- `repeat(after Duration, until LocalDateTime)LocalDateTime:` - Trigger with a certain degree of frequency until a point in time in the future.
- `repeat(after Duration, times long)LocalDateTime:` - Trigger with a certain degree of frequency for a fixed number of times.
- `getCurrentTime()LocalDateTime` - the current time as far as the Pulsar is concerned. This may not correspond to real time for the purposes of testing etc.

It is intended that the returned ref be listened to for triggers via an `every` block. We'd recommend using an `every` block instead of `onchange` to cater for the chance that event has triggered before the triggered block has yet been created for execution.

Once the stream of events returned has reached a state where by it can no longer be updated the ref will be closed. This will occur after the first trigger for the `after` and `schedule` cases, and after the repetition conditions have been met for the `repeat` cases. For the `repeat` forever instance, the ref will have to be closed manually by calling `:close()` or by going out of scope.

Repetition intervals may not be negative. Events scheduled for future execution, via the `after` and `schedule` methods, may be in the past, in which case they will be triggered immediately.

The two provided implementations of the `com.concurnas.lang.pulsar.Pulsar` trait are as follows:

- `actor com.concurnas.lang.pulsar.RealtimePulsar` - Uses the realtime system clock upon which our program is executing for event scheduling.
- `actor com.concurnas.lang.pulsar.FrozenPulsar` - Supports injecting of time as a controllable variable. This is aimed at testing of Pulsar based solutions.

Both implementations above are actors, this is handy because they can be shared between isolates.

37.7.1 Developing temporal applications

Lets look at an example of a temporal 'hello world' application:

```
from com.concurnas.lang.pulsar import Pulsar, RealtimePulsar
from java.time import Duration

trait TaskToDo{
   def doTask() void
}

inject class EventScheduler(pulsar Pulsar, task TaskToDo){
   def scheduleEvent(){
      tigger := pulsar.after(Duration.ofSeconds(10))//schedule event for 10
          seconds time...
      every(tigger){
         task.doTask()
      }
   }
}

class HelloWorldTask ~ TaskToDo{
   def doTask(){
      System out println "Hello world!"
   }
}

provider Temporal{
   provide EventScheduler
   single Pulsar => new RealtimePulsar()//single as we wish use the same
       instance for all provided EventScheduler's
   TaskToDo => new HelloWorldTask()
}


schduler = new Temporal().EventScheduler()
schduler.scheduleEvent()
```

Above we are using the ref returned from a call to the `after` method of our injected `Pulsar` instance to call the `doTask` method of our injected `HelloWorldTask` instance. Our `tigger` ref will have a value set to it after 10 seconds and then it will be closed. The other scheduling methods exposed by the `Pulsar` instance may be used in a similar fashion to the above.

Testing temporal applications

Like concurrent computing, applications making use of temporal computing can be difficult to test, especially for tasks which are scheduled to occur infrequently, irregularly or a long way in the future. The naive way to test this sort of application this is to do so in real-time, and have to wait for however long is required for a scheduled event to occur in the future. With Concurnas however, we can use the `FrozenPulsar` implementation in order to speed up this process. For example:

```
from com.concurnas.lang.pulsar import Pulsar, FrozenPulsar
from java.time import Duration

trait TaskToDo{
   def doTask() void
}

inject class EventScheduler(pulsar Pulsar, task TaskToDo){
   def scheduleEvent(){
      tigger := pulsar.after(Duration.ofSeconds(10))//schedule event for 10
          seconds time...
      every(tigger){
         task.doTask()
      }
   }
}

class TestTask() ~ TaskToDo{
   -taskRun boolean:
   def doTask(){
      taskRun = true
   }
}

provider TemporalTest{
   provide EventScheduler
   single provide FrozenPulsar => new FrozenPulsar()//single as we wish use the
       same instance for all provided EventScheduler's
   single provide TaskToDo => new TestTask()//single for same reasons as above
}


tempoTest = new TemporalTest()
scheduler = tempoTest.EventScheduler()
scheduler.scheduleEvent()
task = tempoTest.TaskToDo() as TestTask

pulsar = tempoTest.Pulsar() as FrozenPulsar

//now we inject the current to our pulsar...
pulsar.currentTime = pulsar.currentTime + Duration.ofSeconds(10)
await(task.taskRun)//wait for task to be run and for this ref to be set with a
    value
```

Above we are progressing time by 10 seconds[1] and injecting it into our `FrozenPulsar` instance. We then wait for our taskRun to be set in our `TestTask` instance as expected.

A point to bear in mind when using the `FrozenPulsar` implementation, say when testing an application making use of a repeatable event, is that progressing time to an infinite period in the future will not result in an infinite number of events being fired off, rather only one event will be triggered as a consequence of the injecting of current time once.

37.8 Parfor

Concurnas has support for parallel for loops in the form of a `parfor` loop. These are a convenient and intuitive mechanism for performing task based operations in the context of a loop, in parallel. The syntax of the `parfor` loop is the same as a regular C style for or Iterator style for except that the `parfor` keyword is used. `parfor` loop's may use index variable's but may not use may use else block's. `parfor` loop's may return values, the returned value shall be of type `java.util.List<X:>` where `X` is the ordinary type returned from the `parfor` block.

For example:

```
def gcd(x int, y int){//greatest common divisor of two integers
   while(y){
      (x, y) = (y, x mod y)
   }
   x
}

res1 = parfor(b in 0 to 10){
   gcd(b, 10-b)
}

res2 = parfor(b =0; b < 10; b++){
   gcd(b, 10-b)
}
//res1 == res2 == [10:, 1:, 2:, 1:, 2:, 5:, 2:, 1:, 2:, 1:, 10:]
```

The `parfor` loop operates by creating an isolate for each iteration upon which it's operating and adding it to a returned list of refs (if a return value is expected).

[1]We are able to use the + operator as it is overloaded in the `LocalDateTime` class which defines a method plus taking a `Duration` object as a parameter

37.8.1 Parforsync

In addition to `parfor`, Concurnas provides `parforsync`. This is functionally the same as `parfor` except it ensures that all spawned isolates have completed execution before returning control to the caller and permitting execution past the `parforsync` block.

```
res1 = parforsync(b in 0 to 10){
   gcd(b, 10-b)
}
//exeuction of code below this point contingent on all isolates completed

//res1 == [10:, 1:, 2:, 1:, 2:, 5:, 2:, 1:, 2:, 1:, 10:]
```

37.8.2 Parfor list comprehension

Both `parfor` and `parforsync` may be used in order to perform list comprehension:

```
res1 = gcd(b, 10-b) parfor b in 0 to 10
res2 = gcd(b, 10-b) parforsync b in 0 to 10
```

Filter expressions may not be applied to the list comprehension expression where `parfor` and `parforsync` are used.

More details of list comprehensions can be found in the List Comprehensions chapter.

37.9 Which solution to use

Concurnas, being a language designed from the beginning for building reliable, scalable, high performance concurrent, distributed and parallel systems presents a wealth of different options regarding computation. Here we present a brief summary of the different options available in Concurnas and what sorts of problems they are best and least (where appropriate) well suited to solving.

One important consideration to bear in mind with any Isolate based concurrency solution in Concurnas (parfor, isolates, actors and reactive programming) is that the cost(both in terms of programming effort and actual computational resources) of creating an isolate is non epsilon. Sometimes problems are of a degree of complexity that, even though they can be solved in a concurrent manner, they would be best suited to a single threaded solution for the cost of solving them in a concurrent manner would be greater, this is particularly acute for simple operations and/or small amounts of data. The effect is magnified when one introduces distributed computing - where we also need to consider a wider array of failure cases.

- **For comprehensions**
  For comprehensions are great in instances where we need a quick one line solution for iterating over elements of an iterable object with the potential for filtering. However, for comprehensions are not concurrent operations and so do not scale with processor cores even if our input data size to iterate over is large.
- **Vectorization**
  Vectorization is a nice convenient way of working with array like data in a very concise manner. Vectorization does not take advantage of the multi-core nature of modern CPUs and so on its own has the same disadvantages as for comprehensions. GPU computing is often a better alternative to numerical computing with vectorization, though the programming model can be more work up front.
- **Java [parallel] streams**

Java streams present a large and very capable API for working with data. They even provide a parallel computation implementation which is based on traditional Thread based execution and hence does not take advantage of the Concurnas model of concurrent execution. Again, for CPU based problems GPU computing is often a better solution.

- **Parfor**
  Parfor is a great solution for when we need to implement an operation on each element of an iterable data structure and those instances do not need to interact with one another. They are a good solution for task based parallel problems - where lot of i/o, interaction with main memory, etc much take place. But for CPU based problems where much calculation is required, GPU programming is usually a better solution.
- **Refs**
  Refs are an integral part of, and help us utilize, other concurrent programming solutions in Concurnas. Refs are built around the concept of optimistic shared state. All writes and reads to refs are atomic, but their state is changeable and often nondeterministic. The optimistic model of computation breaks down somewhat where there is a large degree of contention on a shared ref. For this reason actors are often a better alternative where a ref is shared and contended between a number of isolates.
- **Isolates**
  Isolates, like refs form the basis of many of the concurrency primitives in Concurnas. On their own they are ideally suited to implementing task based solutions to problems.
- **Actors**
  Actors are ideal for cases where we need to implement controlled complex shared mutable state wrapped up within an object. If ever we find ourselves needing to share an object between isolates, then an actor is our answer. Since they effectively turn all requests of them into a single threaded execution chain, if we are not careful they can become bottlenecks in our applications. They are best suited to providing task based, discrete services for instance to i/o and/or controlled access to state.
- **Reactive programming: await, onchange, every, async**
  The reactive programming constructs offered by Concurnas are an excellent way of working with refs in order to solve reactive and temporal logic based problems in an intuitive and natural manner. Like most of the concurrency solutions in Concurnas they can be used for both task based and CPU based computation, though are better suited to solving task based problems.
- **Transactions**
  Transactions enable us to modify more than one ref in an atomic fashion. Care should be taken to make transactions as simple as possible and to avoid non idempotent side effects in their execution for they can be repeated as many times as necessary in order to complete a transaction. Transactions on high contention refs should be avoided, here actors are often a better choice.
- **GPU computing**
  GPU computing is ideal for CPU based execution where we must perform lots of the same operations upon a large data set. Compared to single core execution (e.g. using vectorization or for comprehensions) a speed up of 100x (two orders of magnitude) is often achievable when switching execution to the GPU! Though some extra engineering work is required in order to unlock this. Considerations to bear in mind when using GPU computing are that data transference to and from the GPU can be the bottleneck of many GPU based algorithms. The setup and clean up work required in order to utilize the GPU does require some attention. GPU computing is not appropriate for task based execution.
- **Distributed computing**

Distributed computing is often the final step on the scalability path for our algorithms. Distributed computing allows our programs to run across hundreds of even thousands of computers. Concurnas makes distributed computing easy by virtue of its first class citizen support for this form of computing. Distributed computing is especially useful when accessing remote or shared resources (e.g. databases, high powered GPUS etc). Some additional engineering work is required over localized computing in that the failure landscape for distributed solutions to problems is larger.

Time to wake up
Refill new destiny
Through this

- The Whitest of Lies from Prototype by NeuroWulf (2005).

38. GPU/Parallel programming

Concurnas has built in first class citizen support for programming GPUs and associated parallel computing constructs. GPUs are massively data parallel computation devices which are perfect for solving SIMD (single instruction, multiple data) and normally CPU bound problems. Compared to a single CPU core algorithm implementation, in solving a computation bound problem, it is common to be able to obtain up to a 100x speed improvement (two orders of magnitude) when using a GPU! And this is with a far reduced energy and hardware cost per gigaflop relative to CPU based computation. All modern graphics cards have a built in GPU and there exist dedicated GPU computation devices. In fact some of the world's leading supercomputers achieve their parallelization through the use of dedicated GPU hardware. With Concurnas this power is your as well.

Support is provided via interfacing to OpenCL - an excellent multiplatform API for GPU computing. OpenCL is supported by the big three GPU manufacturers - NVidia, AMD and Intel. However, even with conventional, raw OpenCL (or any GPU computing platform for that matter) one must write a lot of boilerplate code and perform a lot of non work related management in order to get computation done on the GPU. A such, authoring large applications can be an intimidating prospect. With Concurnas you will see that this boilerplate code is minimized, allowing the developer to focus on solving real business problems. Additionally there is no need to learn (and have to paradigm shift into) C or C++ which is the native language used by OpenCL for expressing parallel functions on the GPU as functions marked as parallel in Concurnas are automatically translated into a form understandable by the GPU/OpenCL. GPU computing is for everyone.

The helper classes associated with GPU computing are a part of the core Concurnas API. This can be found here.

What follows is a practical guide covering the key components of GPU computing with Concurnas.

38.1 Prerequisites

In order to make use of the Concurnas GPU parallel programming solution the following prerequisites must be satisfied:

- Access to at least one OpenCL ready GPU on one's computation machine (almost all modern graphics cards support this)
- OpenCL compatible driver compliant to at least version 1.2 of OpenCL for said GPU(s). Consult your graphics card manufacturer for support for details on this.

Concurnas enables parallel computation on GPUs, FPGA accelerators and conventional CPUs. Although OpenCL natively supports execution on the CPU, we do not encourage use of this in Concurnas for solving anything but strictly data based parallel problems.

To see if one has access to any GPU devices one can use the `getGPUDevices` command as part of the GPU API:

```
from com.concurnas.lang import gpus

gps = gpus.GPU()
gpudevices = gps.getGPUDevices()
firstDevice = gpudevices[0]
deviceDetails = firstDevice.toString()

//deviceDetails == NVIDIA CUDA: [GeForce GTX 570]
```

Note that `com.concurnas.lang.gpus` is an auto import, therefore it is not necessary to explicitly import `gpus`, and we shall cease to do so in the subsequent examples.

Note that a machine may have more than one GPU (in fact this is common with machines having integrated graphics processors either on the motherboard or CPU - often being Intel HD graphics). But, if no GPU devices are available it can be useful for some problems to 'fall back' on to CPU based computation. The interface/compute model is the same and one will still obtain the advantages of the SIMD instruction set whilst processing on the CPU which is quicker for data parallel problems than the alternatives provided in Concurnas (which are more oriented towards task parallelism). One can select the CPUs available on one's machine via the `getCPUDevices`, example:

```
def chooseGPUDeviceGroup(gpuDev gpus.DeviceGroup[], cpuDev gpus.DeviceGroup[])
    gpus.Device {
   if(not gpuDev){//no gpu return first cpu
      cpuDev[0]
   }else{
      for(grp in gpuDev){
         if('Intel' not in grp.platformName){
            return grp
         }
      }
      gpuDev[0]//intel device then...
   }
}

gps = gpus.GPU()
deviceGrp gpus.DeviceGroup= chooseGPUDeviceGroup(gps.getGPUDevices(),
    gps.getCPUDevices())
device gpus.Device = deviceGrp.devices[0]
```

In the above example we choose the first non Intel based GPU, failing that we fall back to returning the first CPU device available.

We can examine the capabilities of both the device group and the individual devices of which it composes by calling the get methods of interest on the objects. Here we just look at an example summary for both:

For the DeviceGroup:

```
gps = gpus.GPU()
deviceGrp = gps.getGPUDevices()[0]

summary = deviceGrp getSummary()
/*
summary =>
Device Group:  NVIDIA CUDA: [GeForce GTX 570]
OpenCL Version: OpenCL 1.2 CUDA 9.1.84
Vendor:        NVIDIA Corporation
Extensions:    cl_khr_global_int32_base_atomics ...
*/
```

For an individual device:

```
gps = gpus.GPU()
deviceGrp = gps.getGPUDevices()[0]
device = deviceGrp.devices[0]

summary = device getSummary
/*
summary =>
Device:                   GeForce GTX 570
Vendor:                   NVIDIA Corporation
Address Bits:             64
Available:                true
Compiler Available:       true
Little Endian:            true
Error Correction Support: false
Global Memory Size:       1.3 GB
Local Memory Size:        48.0 kB
Compute Units:            15
Clock Frequency:          1560
Constant Args:            9
Constant Buffer Size:     64.0 kB
Max Memory Allocation Size: 320.0 MB
Param Size:               4352
Max Work Group Size:      1024
Max Work Item Sizes:      [1024 1024 64]
Base Address Align:       4096
Data Type Align Size:     128
Profiling Timer Resolution: 1000
Type:                     CL_DEVICE_TYPE_GPU
Vendor ID:                4318
OpenCL Version:           OpenCL 1.1 CUDA
Driver Version:           391.35
Extensions:               cl_khr_global_int32_base_atomics...
*/
```

38.1.1 Code Portability

When working with ordinary Concurnas code operating on general purpose CPUs, the computing environment and how one codes for it is assumed to be highly homogeneous. I.e. it's rare that one need to consider the clocks speed, amount of L1/L2 cache one's cpu has, or even the amount of RAM available for operation, generally speaking these things are automatically optimized/can be assumed as being adequate for ones software to run. Additionally, one does not need to adapt ones code to optimize for different CPU architectures/RAM configuration etc, this is something which the compiler/virtual machine takes care of.

However, when one is working with GPUs, although there are many optimizations in place as part of the compilation process (e.g. optimal opcode generation, register allocation etc) the decisions required in order to optimally solve a problem on a GPU are generally left for the developer to optimize for, and if one is not diligent, one can write code which is not very portable/optimal for the range of GPUs one's code is run on. There are some properties referenced above which are of particular interest insofar as code portability is concerned, which generally need to be factored in in ones software:

- **Address Bits** - This indicates whether the GPU is operating in a 32 or 64 bit environment. Determines the size of the `size_t` primitive type. In order to assist with portability between 32 and 64 bit environments the `size_t` primitive type - 32 bits (4 bytes being an `int`) or 64 bits (8 bytes being a long) is provided. This is the type returned from functions such as `get_global_id` and `get_local_id`, `sizeof` when used within gpu kernels and functions, is the datatype of pointers and is the type used to address arrays. This permits one to write code that is agnostic of whether one is operating in a 32 or 64 bit environment (though practically one can assume a 64 bit environment in most cases).
- **Local Memory Size** - This is the amount of memory per compute unit which is accessible to a work group (collection of parallel work items) running on it, this memory can be used as a limited shared cache between the work items in a work group.
- **Global Memory Size** - Note that GPUs usually have far less available to them on an individual basis than our host.
- **Max Memory Allocation Size** - Not only are we limited to the Global Memory Size, but chunks of memory may be no larger than this value. This is often a value one must consider when designing an algorithm for portability, and different GPUS often have different max allocation sizes.
- **Max Work Item Sizes** - When designing an algorithm which uses local memory, one must be cognizant of these values.

38.2 Events

It is advantageous for computation on the GPU to take place on an concurrent basis, both for performance reasons - in allowing the GPU to reorganise execution, and for practical reasons - as we can move data to, from and between GPUs (a relatively slow operation) at the same time as they are performing execution, thus enabling us to create mini pipelines of work for our GPU, always keeping it busy, maximising throughput.

In Concurnas we make use of OpenCL events to support control of asynchronous computing on the GPU. These are wrapped within refs and exposed as GPURef's. Many of the core GPU operations operate on an asynchronous, non blocking basis: reading, writing and copying data between the GPU and program execution.

GPURefs have their status set upon completion of the task which created them.

The typical workflow of execution involves first copying data to the gpu, performing execution, and then copying back the result to heap managed memory. We can wait, via the /await/ keyword, on the GPURefs returned from the data writing call to ensure that execution only begins after data copying is complete, similarly, we can wait on the GPURef returned from execution before continuing to copy the result to main memory. Following on from our previous example:

```
inoutGPU = device.makeOffHeapArrayMixed<int[]>(int[].class, 10)
copyComplete boolean:gpus.GPURef = inoutGPU.writeToBuffer([1 2 3 4 5 6 7 8 9 10])
await(copyComplete)
//rest of program will execute only when copyComplete has been set..
got = inoutGPU.readFromBuffer()
```

In the above example, had we not waited for the `copyComplete` ref to complete, the `readFromBuffer` call may have returned an unexpected result. So by explicitly waiting we are ensuring consistency.

There is a way to perform the above synchronization which is generally preferred from a performance and elegance of code perspective. All the non blocking GPU operations take a vararg of GPURef's to wait for before continuing execution:

```
inoutGPU = device.makeOffHeapArrayMixed<int[]>(int[].class, 10)
copyComplete boolean:gpus.GPURef = inoutGPU.writeToBuffer([1 2 3 4 5 6 7 8 9 10])
got = inoutGPU.readFromBuffer(copyComplete)//readFromBuffer will execute when
    coyComplete has been set
```

In the above example the blocking of the readFromBuffer call will take place within the OpenCL framework. Generally speaking this is the preferred method of synchronization as the control takes place closer to the hardware and the code is easier to write (at least in omitting the explicit call to await).

If one forgets to capture the resultant GPURef from an asynchronous, non blocking GPU operation, then the operation will turn into a blocking one as the calling code will wait for the resulting GPURef to be set. This is because all the asynchronous non blocking GPU operations are marked with the `@com.concurnas.lang.DeleteOnUnusedReturn` annotation.

38.3 Data transference

38.3.1 Buffers

Before we can perform calculations on the GPU we must first copy data to our GPU device. We can work with both scalar values (single values) and n dimensional arrays. Data must be of type either: primitive or boxed primitive (Integer, Character etc). We must first create an appropriate buffer to work with the data on the GPU, this is a blocking operation:

```
singl gpus.GPUBufferInputSingle = device.makeOffHeapArrayInSingle<int>(int.class)
```

Here, single has been created as an In buffer, allowing us to copy data into it only. Single and n dimensional array buffers can be created as either:

- **In**: permitting only writing of data to them, most often used for arguments to kernels (makeOffHeapArrayInSingle/makeOffHeapArrayIn)
- **Out**: Permitting only reading of data from them, useful for results of kernels (makeOffHeapArrayOutSingle/makeOffHeapArrayOut)
- **Mixed**: Permitting both reading and writing of data. (makeOffHeapArrayMixedSingle/makeOffHeapArrayMixed)

It is advised that one chooses from the above types of buffer carefully, as it allows the GPU to optimize memory access.

The reified method local generic type argument is used in order to determine the size of the underlying data allocation space on the GPU.

Let's now create some n dimensional arrays on the GPU:

```
ar gpus.GPUBufferInput = device.makeOffHeapArrayIn<int[]>(int[].class, 10)
mat gpus.GPUBufferInput = device.makeOffHeapArrayIn<int[2]>(int[2].class, 2,
    4)//2 by 4 matrix
```

Here we are creating both an array (ar) and a matrix (mat). Note how we must specify the dimensionality of the structure by populating the second argument of the method (which is a vararg) with its dimensions.

Normally, for n dimensional arrays, Concurnas uses Iliffe vectors (wikipedia link), but when working on the GPU, memory is organized in a serial manner. In practice this means is that n dimensional (where n> 1) arrays cannot be irregular in shape if you wish to correctly work with them on the gpu.

38.3.2 Writing, Reading Data

All of the commands used to read, write and copy data to GPU buffers are non blocking and will return a GPURef for synchronization. They also take a list of GPURefs to wait for completion for as a final vararg.

Let us write some data to an In Buffer:

```
ar gpus.GPUBufferInput = device.makeOffHeapArrayMixed<int[]>(int[].class, 10)
copyComplete boolean:gps.GPURef = ar.writeToBuffer([1 2 3 4 5 6 7 8 9 10])
```

We can pass any n dimensional array of the buffer type to an In or Mixed buffer using the `writeToBuffer` method. However, since the data is flattened for purposes of execution on the gpu, but we must ensure that the data passed is of equal size to that of the buffer otherwise a runtime exception will be thrown. For instance, it would be acceptable to for a 1 dimensional int array of 10 elements to be passed to a 2 dimensional array (matrix) buffer with 2 x 5 dimensionality as the data size is the same (`10 == 2 * 5 => 10` entries).

We can read from a buffer as follows:

```
result int[] = ar.readFromBuffer(copyComplete)
```

Note above by passing in the `copyComplete` ref we are waiting for the write operation to first complete.

Single buffers are even easier to work with:

```
ar gpus.GPUBufferInput = device.makeOffHeapArrayMixed<int>(int.class)
copyComplete boolean:gps.GPURef = ar.writeToBuffer(99)
result int = ar.readFromBuffer(copyComplete)
//result = 99
```

38.3.3 Copying Data

We can copy data from an Out or Mixed buffer to a In or Mixed buffer using `copyToBuffer`:

```
intoutGPU1 = device.makeOffHeapArrayMixed<int[]>(int[].class, 10)
intoutGPU2 = device.makeOffHeapArrayMixed<int[]>(int[].class, 10)
c1 := intoutGPU1.writeToBuffer([1 2 3 4 5 6 7 8 9 10])

g1 := intoutGPU1.copyToBuffer(intoutGPU2, c1)
//intoutGPU2 now contains a copy of the data asigned to intoutGPU1
```

The above copy mechanism can be used in order to copy non overlapping regions of data inside the same buffer.

38.3.4 Writing, Reading and Copying subregions

We can copy subregions of data to and from as well as between buffers using the `[a ... b]`, `[a .. ]` and `[ ... b]` syntax and via additional arguments to the `copyToBuffer` method. When using the `[ ... ]` syntax since we have no way of capturing the resultant GPURef of the write or read operation, the caller will block implicitly until this is returned, and will call delete on the GPURef object freeing the memory used for it on the GPU.

```
intoutGPU1 = device.makeOffHeapArrayMixed<int[]>(int[].class, 20)
//intoutGPU1 is inialized with all 0's

intoutGPU1[5 ... 8] = [66 55 44]//blocking operation
subWrite := intoutGPU1.subAssign(15, 18, [66 55 44])//non blocking operation

await(subWrite)

result1 = intoutGPU1[5 ... 8]//blocking subread
result2 := intoutGPU1.sub(15 18)//non blocking subread

await(result2)
```

The other variants of subregion are intuitive.

When copying between regions we can add additional arguments to the `copyToBuffer` method:

```
intoutGPU1 = device.makeOffHeapArrayMixed<int[]>(int[].class, 10)
intoutGPU2 = device.makeOffHeapArrayMixed<int[]>(int[].class, 10)
c1 := intoutGPU1.writeToBuffer([1 2 3 4 5 6 7 8 9 10])

g1 := intoutGPU1.copyToBuffer(intoutGPU2, 2, 4, 2, c1)
```

Here, we are copying two items from `intoutGPU1` into the 2nd index of `intoutGPU2`.

38.4 sizeof

It can be useful to know how many bytes of memory a structure will take up on the GPU. To this end a qualifier can be specified to the `sizeof` keyword as follows:

```
myArray = [0 1 2 3 4 5 6 7 8 9]
size = sizeof<gpus.gpusizeof> myArray
//myArray == 40 bytes
```

38.5 Kernels and functions

Kernels are the core which drive our execution on the GPU. The compute model for kernels does have a number of differences relative to non gpu Concurnas code which one needs to be aware of in order to derive maximum value from them.

Here is an example kernel for simple matrix multiplication[1] with its conventional, non GPU counterpart on for comparison:

[1]Those readers familiar with the intricacies of GPU optimization will recognize this as a naive, unoptimized, implementation. This is intentional and we shall cover some details of Kernel optimization later.

Listing 38.1: GPU kernel

```
gpukernel 2 matMult(wA int, wB int,
    global in matA float[2], global
    in matB float[2], global out
    result float[2]) {
   globalRow = get_global_id(0) //
      Row ID
   globalCol = get_global_id(1) //
      Col ID

   value = 0f;
   for (k = 0; k < wA; ++k) {
      value += matA[globalCol * wA +
          k] * matB[k * wB +
          globalRow];
   }

   // Write element to output matrix
   result[globalCol * wA +
      globalRow] = value;
}
```

Listing 38.2: CPU equivilent

```
def matMult(M int, N int, K int, A
    float[2], B float[2], C
    float[2]) {
   for (m=0; m<M; m++) {
      for (n=0; n<N; n++) {
         acc = 0.f
         for (k=0; k<K; k++) {
            acc += A[k][m] * B[n][k]
         }
         C[n][m] = acc
      }
   }
}
```

The GPU kernel is only executable on a device as par below. Kernels cannot be executed via normal function invocation. GPUs operate on a single instruction, multiple data basis, thus unlike the non gpu code on the right, the two outermost for loops are not required as the kernel is executed in parallel on the cores of the GPU device executing it.

We can use something like the following in order to execute the kernel on the GPU (assuming we have already selected a device):

```
inGPU1 = device.makeOffHeapArrayIn(float[2].class, 2, 2)
inGPU2 = device.makeOffHeapArrayIn(float[2].class, 2, 2)
result = device.makeOffHeapArrayOut(float[2].class, 2, 2)

c1 := inGPU1.writeToBuffer([1.f 2.f ; 3.f 4.f])
c2 := inGPU2.writeToBuffer([1.f 2.f ; 3.f 4.f])

inst = matMult(2, 2, 2, inGPU1, inGPU2, result)//create a 'kernel' object from
    our gpukernel
compute := device.exe(inst, [2 2], c1, c2)//execute kernel on device, with 2 x 2
    (4) work items, whilst waiting for c1 and c2 (operations to write to gpu) to
    complete
 ret = result.readFromBuffer(compute)//read result from output buffer, whilst
     waiting for gpu to finish execution of our kernel
```

We shall now look in detail at what is taking place above...

38.5.1 Work Items

The memory/execution model for OpenCL exposed within Concurnas is as follows:

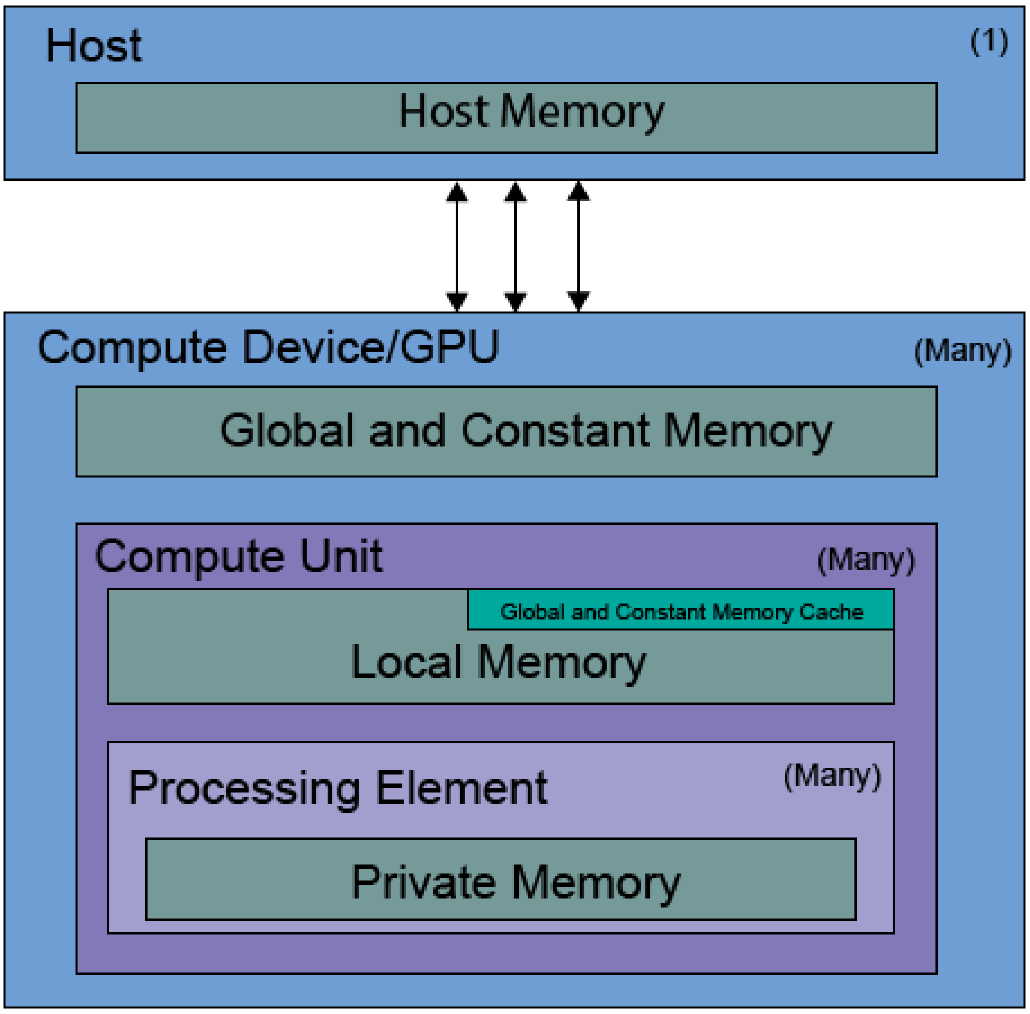


Figure 38.1: Memory/execution model for OpenCL exposed within Concurnas

As shown in 38.1, each host machine may have many Compute devices, each made up of multiple compute units, themselves holding multiple processing elements.

When we execute Kernels, they are executed as work items in parallel across processing elements of the compute units (as work groups) on the GPU device(s) we invoke them on. Processing elements have their own private memory space where they store registers etc, and they have access to the local memory of the compute unit they are a part of as well as global memory. Access to these memory spaces for processing elements executing work items is exponentially slower the further one moves away from the processing element - hence, private memory access is the quickest, then local memory and then global memory, however, the amount of memory at each level available is inversely proportional to the locality. We have some respite when accessing repeated elements of global memory however as a portion of local memory (normally 16Kb of the 64Kb total) is dedicated to caching global and constant memory.

As we have already seen, interaction between host and compute device memory is governed via the use of buffers. Moving data from the host to the GPU is a relatively slow operation, in fact, generally speaking reading data from a GPU is approximately 10x slower than writing data. This must be factored into the design of most systems which take advantage of the GPU.

Each kernel instance is able to determine which unique work item it is by invoking the `get_global_id` auto imported gpu function.The most common way to operate on data in parallel on the gpu is then to take this identifier and use it to select a section of data to operate on. For example, here is a very simple kernel adding two 1 dimensional arrays together, interacting with both global and constant memory:

```
gpukernel 1 addtwo(constant A int[], constant B int[], global out C int[]){
   id = get_global_id(0)
   C[id] =A[id] + B[id]
}
```

Each work item here is operating on one item of data (one from A, one from B and one written to C), but, as par our matrix multiplication example above, work items may address more than one item of data.

We generally do not need to concern ourselves with the way in which Kernels are mapped to compute units and then individual work items (as this is automatically handled) unless we are looking at optimizing code - which we shall examine later.

38.5.2 Kernel dimensions

Most GPU parallelism is performed on n-dimensional array data. N-dimensional arrays in Concurnas kernels are always flattened to 1 dimensional arrays. This is performed implicitly when data is copied into a GPU Buffer. Likewise, when data is copied out of a GPU buffer, it is converted back into its n dimensional form. Given this flattening of n dimensional array data, navigation around the array space is thus performed via arithmetic, with consideration for the dimensionality of the arrays.

In order to assist with this array navigation, kernels must specify a dimension parameter between 1 and 3 (set to `2` in the above example). One can consider that for the case of a 2 dimensional kernel, the execution is performed as a matrix. Thus a call to `get_global_id(0)` provide an x axis, and `get_global_id(1)` will a y axis reference (note if the kernel dimensionality was set to 3 then we would be able to call, `getGlobalId(3)` to return a z axis reference). If the kernel were specified as a 1 dimensional kernel then only a `getGlobalId(0)` call would be valid.

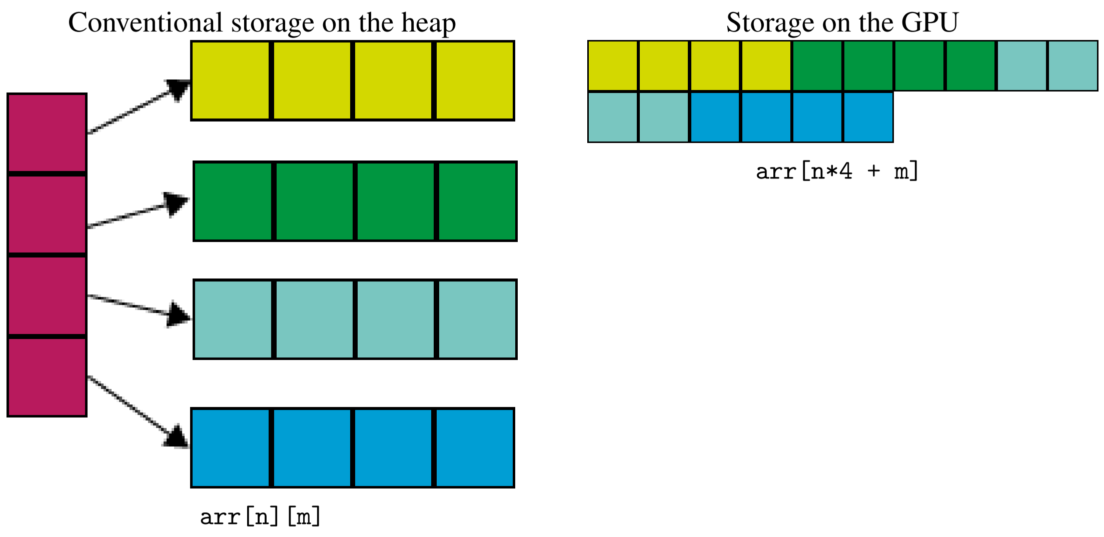


Figure 38.2: A 4 x 4 matrix allocated on the heap and the same matrix on the GPU.

In 38.2 we see an example of this effect on a 4x4 matrix and how on these are addressable conventionally and on the GPU.

38.5.3 Kernel arguments

The syntax for kernel arguments is slightly more involved than for normal function definitions in Concurnas:

```
annotations? (global|local|constant)? (in|out)? (val|var)? name primitive_type
```

All kernel parameters must be of primitive time or boxed equivalent (Integer, Float etc). Any kernel parameters may be marked as either: global, local or constant or without a qualifier - those that are are treated as pointers (See Pointers) within the body of the function:

- `global` - The parameter value is held within the global memory space, it must be a pointer type (function parameters are automatically converted to this form)
- `local` - The parameter value is held within the local memory space specific to the compute unit the processing element operating the work item instance of the current kernel resides within. Note that it is limited in size to generally no larger than 64 Kb.
- `constant` - The parameter is a constant. Changes cannot be made to the value. The amount of constant space on a gpu is limited and implementation specific (see the device parameter: Constant Args and Constant Buffer Size above), though usually it's 64Kb.
- BLANK - Indicates that the parameter is private to the kernel instance and may not be shared amongst work group items. Changes can be made but they are not visible outside of the work item.

Kernel parameters marked as global may optionally be marked with the type of buffer capability they expect. This further enhances the type safety provided by Concurnas when working with GPUs:

- `in` - indicating that only an in or mixed buffer may be used as an invocation argument.
- `out` - indicating that only an out or mixed buffer may be used as an invocation argument.
- BLANK - indicating that any type of buffer (in, out or mixed) is permissible as an invocation argument.

Kernel arguments may be specified as arrays of dimensionality greater than 1, however, within the kernel itself these are flattened to one dimension, as discussed in the Kernel dimensions section above, are only addressable as such. Additionally they are useable only as pointers.

Type erasure. Due to the fact that generic types are erased at runtime it is not possible for Concurnas to differentiate between gpukernel function signatures differing by global/local/constant parameter qualification or in kernel dimensions. Hence the following three definitions are ambiguous:

```
gppukernel 1 kfunc(global A int[]){... }
gppukernel 1 kfunc(local A int[]){... }//ambiguous
gppukernel 2 kfunc(global A int[]){... }//ambiguous
```

But Concurnas is able to differentiate between signatures differing in terms of in/out parameter qualification. Hence the following is not ambiguous:

```
gppukernel 1 kfunc(global in A int[]){... }
gppukernel 1 kfunc(global out A int[]){... }//NOT ambiguous!
```

38.5.4 Calling functions from kernels

One is not limited just to executing code within kernels. Kernels may invoke other kernels (with matching kernel dimensions), built in utility functions as well as defined callable functions declared as gpudef:

```
gpudef powerPlus(x int, raiseto int, plus int) => (x ** raiseto ) + plus
```

As with kernels, input parameters may be marked as either global, local or constant or without a qualifier, they cannot be invoked like normal Concurnas functions, are type erased and they are subject to the same restrictions in their definition, as described below. `gpudef` functions cannot be called by ordinary Concurnas code.

There are a number of essential and useful built in utility functions which are auto included for all gpu kernels and gpu functions. Two common and essential ones are:

- `get_global_id(dim int)int` - returns the global id for the dimension specified
- `get_local_id(dim int)int` - returns the local id for the dimension specified.

The full list of utility functions can be found at: `com.concurnas.lang.gpubuiltin.conc`

38.5.5 Stub functions

Sometimes, particularly in the cases where one has existing opencl C99 compliant OpenCL C code to use in Concurnas, it can be useful to define functions as stubs and reference in this code. We can achieving this in two ways via the use of the `com.concurnas.lang.GPUStubFunction` annotation as follows:

Explicit source. The source parameter of the annotation consumes a String holding the C99 compliant OpenCL c code to load in place of the function definition when called by the GPU. This is a useful technique for expressing smallish blocks of code:

```
@GPUStubFunction(source="float plus(float a, float b){ return a + b;}")
gpudef plus(a float, b float) float
```

Source files. The sourcefiles parameter of the annotation consumes an array of Strings referencing the files holding the C99 compliant OpenCL c code to use in place of the function definition when called by the GPU. The file referenced may be absolute or relative to the working directory:

```
@GPUStubFunction(sources=['./cpucode.cl', './gpuutilCode.cl'])
gpudef plus(a float, b float) float
```

The two methods above can be used together, i.e. it's possible to define both explicit source code and files to import. The Annotation may be attached to regular functions (which may optionally be declared abstract), gpudef functions and gpukernels (which must be declared abstract).

It's also possible to use neither explicitly defined source code or reference any source files. In this case Concurnas will assume that the dependant code has been included in a explicit source or source file definition specified in a different GPUStubFunction reference.

Note that Concurnas assumes that the source code provided is correct, i.e. the enhanced type safety which Concurnas provides at compile time for GPUs cannot be provided with stub functions. However, runtime safety is still provided, and invalid code within a kernel chain will be flagged in a GPUException upon invocation.

38.5.6 Recursion

We are not permitted to make use of recursion when defining gpu kernels and functions, either directly or indirectly. However, it's often rare that an algorithm genuinely requires recursion, beyond satisfying code aesthetics, in order to solve a problem. Often recursive solutions to problems can be rewritten in a non recursive, though less graceful, matter. Let us look at a classic recursive solution to the fibonacci series generation problem, in classical Concurnas on the left and in Parallel gpu Concurnas code on the right:

Listing 38.3: Classical Concurnas code

```
def fib(n int) long{
   if(n == 0){
      return 0
   }elif(n <== 2){
      return 1
   }else{
      return fib(n-2) + fib(n-1)
   }
}






.
```

Listing 38.4: Parallel GPU Concurnas code

```
gpudef fib(n int) long {
   if(n == 0){
      return 0
   }elif( n <== 2){
      return 1
   }else{
      r = 0L; n1 = 1L; n2 = 1L
      for(i = 2; i < n; i++){
         r = n1 + n2
         n1 = n2
         n2 = r
      }
      return r
   }
}
```

38.5.7 Function overloading

Although Concurnas will permit us to define more than one version of a function in an overload manner (same name, different signature), only one version of that overloaded function may be referenced in an execution of a kernel. This restriction is checked for via a compile time and runtime check.

38.5.8 Kernel and function variables

Similar to kernel and function parameters, variables may be qualified with `local` or `global`. E.g.:

```
local xyz3 int[] = [1 2 3] //an array defined in local memory
```

Variables qualified as `constant` may only be created at the top level of the code outside of any gpu kernels or functions. They may be referenced across multiple gpu kernels or functions. For example:

```
constant fixedInc = 100

gpukernel 1 incrementor(global in A int[], global out B int[]) {
ida = get_global_id(0)
B[ida] = A[ida] + fixedInc
}
```

There are a some additional caveats concerning qualified variables one must note:

- Constant variables cannot be re-assigned to once they have been declared.
- Global variables may be of pointer type only.
- By qualifying variables in the above manner, we are indicating to our GPU that we wish the value of the variable to persist within the const, local or global memory space of the GPU. As such, once a variable has been qualified as say, global, it cannot be reassigned to a non global (practically just local) variable, as they are different physical memory spaces on the GPU.

38.5.9 Kernel/function restrictions

When it comes to writing code to run on the gpu, whether it be kernels or functions, there are a number of restrictions in terms of what components of Concurnas can be used both when defining and using them. Some of these have already been elaborated. The full list is as follows:

- GPU Kernels and functions must be declared as functions. They may not be class methods or be nested in any way.
- Only GPU Kernels and functions may only invoke one another, they cannot invoke normal Concurnas functions. Non GPU Concurnas code may not directly invoke GPU Kernels or functions. (invocation from non GPU Concurnas code is described in the section below Invoking Kernels).
- The parameters of GPU Kernels and GPU functions must all:
  - Be of primitive type (or boxed equivalent) and be of any array dimensionality (though inside the kernel this is flattened to one level if marked global, local or const)
  - May not be varargs
  - May not have default values
  - May be marked as either global, local or constant or without a qualifier. If the parameter is qualified then it will be converted to a pointer within the body of the function. On invocation, will require a corresponding buffer object to be passed to it.
  - May be marked with in or out to denote readwriteability.
- GPU Kernels must return void.
- Variables declared outside the scope of a gpu kernel or gpu function must be declared val for them to be used inside.
- Recursion of GPU Kernels or GPU functions, either directly or indirectly is not supported.
- Overloading of GPU Kernels or GPU functions is not supported.
- The new operator may not be used - except for creating arrays.
- Arrays can only be created with dimensions expressed as constants[2].
- Kernels must have a dimensionality expressed of between 1 and 3. Note that Kernel parameters themselves may be natively of more than one dimension, when passed to a GPU buffer they will be implicitly flattened to one dimension as previously described.

Additionally, the following elements of Concurnas syntax/functionality may not be utilized:

- Objects:
  - Enumerations
  - Direct or self returning dot operator
  - Lambdas
  - init blocks
  - Method references
  - `&==` and `&<>`
  - Annotations
  - The super keyword
  - Local classes
  - Nested functions
  - Tuples
- Object Providers
- Generics
- Pattern matching
- Exceptions:
  - Try catch
  - Throwing of exceptions
- Compound statements:
  - Break and continue returning a value
  - Only `for( ; ; )` is valid, other variants of `for` are not
  - For and while blocks with else

[2]These are created on the stack and not the heap, unlike conventional arrays in Concurnas

  - For and while blocks an index
  - Compound statements which return a value (`if`, `while`, `for` etc)
- Concurrency/reactive computing:
  - The await keyword (if you want to use this then barriers are probably the solution)
  - Refs
  - The changed keyword
  - `onchange`, `every`, `async`
  - `async`, `sync`
  - Actors
  - `parfor`, `parforsync`
  - Transactions
- Arrays/Maps:
  - Arrays of non constant size (e.g. `new int[n]` where `n` is a variable)
  - Array default values
  - The length parameter of an array is not exposed.
  - List instantiation
  - Empty arrays
  - Maps
  - Array pre/post and sublist range references
  - Instantiated arrays outside of assignment statements
- References to non constant module level, global, variables
- Other:
  - The assert keyword
  - Named parameters on function invocations
  - `@` copy
  - The `del` keyword
  - offHeap data
  - The `in` or `is` keywords
  - the >>> operator (<< and >> are ok)
  - `sizeof` with qualification. Normal `sizeof` is ok
  - string and regex declarations
  - The `with` keyword
  - Language extensions
  - Multi assign using an assignor other than = or with `global`/`local`/`constant` variables.
  - The null safety related operators: safe call: `?.`, elvis: `?:` or not-null assertion: `??`

Concurnas handles the above restrictions at multiple levels in the programming and execution process by checking for them as part of the type system, during compilation time and finally at runtime. Unfortunately, for the C99 compatible C code expressed via stub functions, the restriction checking and error reporting, although comprehensive, is performed at runtime - which is of course not as convenient or safe as if it were to be performed exclusively at compile time.

38.5.10 Invoking kernels

Returning to our complete matrix multiplication example above:

```
inGPU1 = device.makeOffHeapArrayIn(float[2].class, 2, 2)
inGPU2 = device.makeOffHeapArrayIn(float[2].class, 2, 2)
result = device.makeOffHeapArrayOut(float[2].class, 2, 2)

c1 := inGPU1.writeToBuffer([1.f 2.f ; 3.f 4.f])
c2 := inGPU2.writeToBuffer([1.f 2.f ; 3.f 4.f])
```

```
inst = matMult(2, 2, 2, inGPU1, inGPU2, result)//create a 'kernel' object from
    our gpukernel
compute := device.exe(inst, [2 2], c1, c2)//execute kernel on device, with 2 x 2
    (4) work items, whilst waiting for c1 and c2 (operations to write to gpu) to
    complete
ret = result.readFromBuffer(compute)//read result from output buffer, whilst
    waiting for gpu to finish execution of our kernel
```

We first obtain the kernel by calling the gpukernel function like a normal function invocation, this will return a GPUKernel object which can be executed by a device. Global, local and constant kernel parameters must be satisfied with arguments that are buffers (inGPU1 and inGPU2 above). Unqualified arguments may point to non buffered variables, but note however that these are copied at invocation time to the GPU so it is advisable to keep these data structures small since as previously discussed, data transference is often the bottleneck in GPU work

Next we call exe on our chosen device. If this is the first time we have called the kernel on the device's parent device group, then the kernel will be compiled which takes a non epsilon amount of time. We must pass in an array, the product of which corresponds to the number of work items we wish to spawn in the number of dimensions (between 1 and 3 inclusive), corresponding to those required of our kernel. We may also optionally specify any local dimensions in the same way. Finally we can optionally pass in references to events to wait for completion of, in the above example we wait for the buffer write operations in events c1 and c2 to complete.

An GPURef object holding the execution completion event is returned from the exe method. Execution of a kernel is said to have completed one all of the work items have completed execution. As with buffer operations, kernel execution is a non blocking asynchronous operation, so we must wait on this object is appropriate.

38.6 Profiling

The Concurnas GPU implementation provides detailed profiling options. This is useful for both development, in enabling one to debug one's GPU code so as to determine how much time is spent performing certain operations, and for monitoring purposes. GPURef objects created from asynchronous non blocking gpu operations have a getProfilingInfo method, which returns an object of type GPUProfilingInfo class holding all the profiling information available:

```
c1 := inGPU1.writeToBuffer([ 1 2 3 4 ])
copyduration = c1.getProfilingInfo().getWorkDuration()
summary = c1.getProfilingInfo().toString()
```

In the above example, the value returned from the getWorkDuration method call is a delta in nanoseconds (divide this by 1e6 to obtain a millisecond value).

38.7 Pointers

Since when working with GPUs we are working closer to the metal than ordinary Concurnas code, when writing gpu kernels and gpu functions we are afforded access to the use of pointers. These operate the same as pointers in low level languages such as C and C++ and can be very useful for defining programs to operates on the GPU or porting over existing code to operate within the Concurnas framework and not having to rely on GPUFunctionStubs (and the reduced amount of type safety stubs expose). Pointers are especially useful when working with subregions of global memory variables.

A pointer is a variable which contains the memory address of a different variable. We can have a pointer to any variable type. Pointer types are defined as follows:

`(*+)ordinaryType`

Example pointer types:

```
*int //a pointer to an integer
*float//a pointer to a float
**int//a pointer to a pointer to an integer
***int//a pointer to a pointer to a pointer to an integer
```

The ~ operator provides the address of a variable:

`~variable`

We can use the ~ operator in order to create pointers like this:

```
normalVar int = 12
pnt *int = ~normalVar //pointer holding the address of normalVar
pnt2pnt **int = ~pnt //pointer holding the address of the pointer to normalVar
```

The type upon which we apply the address `&` operator must match that to which the pointer type refers. For example:

```
aint = 12
afloat = 12.f

pnt1 *int = ~aint //ok int == int
pnt2 *int = ~afloat //error, type mismatch, int <> float
```

If we want to obtain the value pointed to by a pointer, we prefix the variable with the dereference operator /*/. We can prefix it for as many times as we require dereferencing:

```
normalVar = 12
pnt  = ~normalVar
pnt2pnt  = ~pnt

//now for the dereferencing:

normalVarAgain int = *pnt
aPointer *int = *pnt2pnt
normalVarOnceMore int = **pnt2pnt
```

In order to disambiguate pointer dereferencing from the multiplication operator, parentheses must be used:

```
normalVar = 12
pnt  = ~normalVar
pow2 = pnt) *(*pnt)
```

The dereferencing operation is similar for when we want to set the value pointed to by a of a pointer:

```
normalVar = 12
pnt *int = ~normalVar
*pnt = 99 //the variable normalVar now equals 99
```

38.7.1 Pointers and Arrays

Pointers and arrays in gpu kernels and gpu functions are two closely related concepts. We can set the value of a pointer to be the address of an array with element offset (remember, we count from 0) as follows:

```
normalArray = [1 2 3 4 5]
pnt1 *int = ~normalArray[0] //pnt1 is set to the address of the first (start)
    element of the array
pnt2 *int = ~normalArray[1] //pnt1 is set to the address of the second element
    of the array
```

An especially useful feature of pointers to arrays (and strictly speaking, pointers in general) is the ability to perform pointer arithmetic and use them, syntactically, as one dimensional arrays:

```
normalArray = [1 2 3 4 5]
pnt1 *int = ~normalArray[0]
pnt2 *int = ~normalArray[1]

//arithmetic...

pnt1++//pnt1 now points to the second entry in normalArray
avalue = *(pnt1+1)//avalue now equals the 3rd value from normalArray, 3
*(pnt2+1) = 100//set the 3rd value of normalArray to 100
extracted = pnt2[3]//obtain the 3rd value of normalArray, which is now 100
```

Multi dimensional arrays can be operated on in a similar way:

```
myAr = new int[3, 4] //a matrix
myAr[0, 0] = 12
myAr[0, 1] = 14

pnt *int = ~myAr[0][0]//set pnt to point to the first value of the matrix
pnt += 1//pnt now points to the second value of the first row of the matrix

value = *pnt//value is now 14
```

38.7.2 Array of pointers

Sometimes it can be useful to make use of an array of pointers. This can be defined as follow:

```
a = 44
b = 88
c = 23

pntc = ~c
arOfPnt *int[] = [~a ~b pntc]//array of pointers
*arOfPnt[1] = 99 //b now equals 88
```

38.8 Local memory

It is expensive to read and write to global memory, thus for algorithms involving a high degree of spatial locality, using local memory can greatly improve processing time performance. This is achieved by essentially manually caching the data of interest to a work group (composed of a number of work items operating on processing units) within the local memory region specific to the compute unit work group. Here we will examine how this can be achieved.

Data that is in local memory is shared between all work items in a work group. Recall that compute units on a gpu execute the work items as part of a work groups on their processing elements. This interaction usually needs to be synchronised by using barriers. Work items in different work groups (resident therefore on different compute units) cannot share information via local memory - this has to be achieved via using global memory.

38.8.1 Local Buffers

We can specify local kernel parameters by qualifying them with `local`. When doing so they must be satisfied with `gpus.Local` buffers on kernel invocation as follows:

```
localBuffer = gpus.Local(long[].class, localItemSize)
```

Essentially we are just providing the type and buffer size information. Local buffers provide no means to write into/out of them outside the GPU. This of course means that local kernel parameters may not be further qualified as being `in out` parameters.

The amount of local memory available to a work group executing on a compute unit is limited - generally it is no more than 64Kb, to be shared across multiple buffers. To see how much local memory is available the `device.getLocalMemSize()` function can be called.

38.8.2 Barriers

Since we are working very close to the metal with GPUs, synchronization of parallel work items running on a GPU compute unit is slightly more involved than what we are used to elsewhere in Concurnas. We use barriers in order to ensure that all work items executing on a processing element have finished writing to local (or global) memory. This is essential for algorithms which use local memory on a reductive/iterative basis.

We achieve this by using the barrier gpudef function:

- **barrier(true)** - The barrier function will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory.
- **barrier(false)** - The barrier function will queue a memory fence to ensure correct ordering of memory operations to global memory. This can be useful when work-items, for example, write to buffer and then want to read the updated data.

38.8.3 Local Work Size

When invoking gpukernels using local memory we must specify the dimensions used for our local identifiers accessible by the work items running as part of a work group. We do this in much the same way as we defined global dimensions on invocation; we pass in an array of dimensions. There are however some caveats we must be aware of. The specified local dimensions must:

- Evenly divide into the global dimensions specified.
- Be no larger than the device specific Max Work Item Sizes per dimension (up to 3). This can be determined at runtime by calling `device.getMaxWorkItemSizes()`.

Due to the requirement that local work size dimensions evenly divide into global dimensions sometimes it means that data will have to be padded or the algorithm used otherwise adjusted.

A work item can determine its local id by using the `get_local_id` gpudef function. It is also often useful to know the size of the local work size, the `get_local_size` gpudef function provides this.

Concurnas uses OpenCL to achieve GPU parallelism, and this has been designed to be agonistic of the number of compute units available for performing execution, nevertheless it is useful to know what this is and calling `device.getComputeUnits()` will return this.

38.8.4 Example Kernel using local Parameters

Reduction is an important algorithmic concept which is often encountered when building parallel algorithms for running on GPUs. Often it is used in order to derive a singular or reduced summary value from previous calculations. Here we examine a reduction algorithm with calculates the sum of long values in an array.

This diagram illustrates the general algorithmic approach and how local memory fits into this:

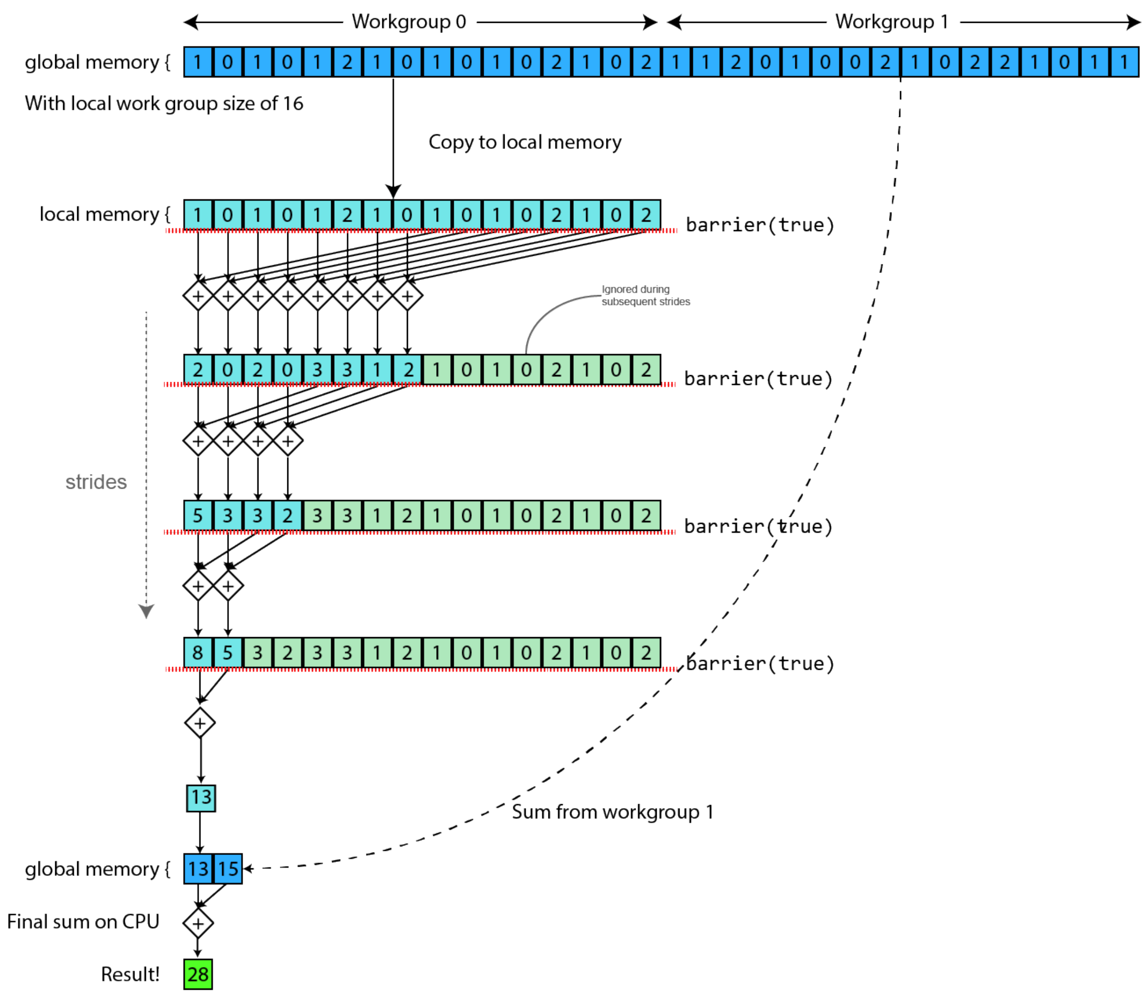


Figure 38.3: Reduction algorithm with multiple local memory strides

In 38.3 we see a visual representation of the reduction algorithm. First we copy our data into global memory, then we pass over to the algorithm above. Once there each of the items in our work group copy the segment of data they are working on into local memory, before iteratively reducing the data in that buffer by half on each round of summation until only one value in local buffer array slot zero remains. This is then written into global memory and the final summation of these values (one for each work group) calculated on the CPU. Note that we set the local work group to size 16 in order to make it easier to show on the diagram, but in practice this value will be much larger.

Given that we are reducing by half on every iteration it is important for this algorithm that the number of work items in a work group be a power of two. This is of course not a requirement for all algorithms taking advantage of local memory.

Now let us look at the kernel, invocation and supporting function code:

```
gpukernel 1 reduce(global val input long[], global partialSums long[], local
    localSums long[]){
   local_id = get_local_id(0)
   group_size = get_local_size(0)

   // Copy from global memory to local memory
   localSums[local_id] = input[get_global_id(0)]

   // Loop for computing localSums
   stride = group_size/2
   while(stride>0) {
      // Waiting for each 2x2 addition into given workgroup
      barrier(true)
      // Divide WorkGroup into 2 parts and add elements 2 by 2
      // between local_id and local_id + stride
      if (local_id < stride){
         localSums[local_id] += localSums[local_id + stride]
      }
      stride/=2
   }

   if(local_id == 0){ // Write result into global memory
      partialSums[get_group_id(0)] = localSums[0]
   }
}

def chooseGPUDeviceGroup(gpuDev gpus.DeviceGroup[], cpuDev gpus.DeviceGroup[]) {
   if(not gpuDev){//no gpu return first cpu
      cpuDev[0]
   }else{//intel hd graphics not as good as nvidea dedicated gpu or ati so
       deprioritize
      for(grp in gpuDev){
         if('Intel' not in grp.platformName){
            return grp
         }
      }
      gpuDev[0]//intel device then...
   }
}

def makeData(itemcount int, range = 3){
   data = new long[itemcount]

   random = new java.util.Random(654L)
   for(i = 0; i < itemcount; i++){
      data[i] = random.nextInt(range)//output: 0,1 or 2 with default range = 3
   }
   data
}

def sum(inputs long...){
   ret = 0
   for(i in inputs){
      ret += i
   }
   ret
}


def main(){
   gps = gpus.GPU()
   deviceGrp = chooseGPUDeviceGroup(gps.getGPUDevices(), gps.getCPUDevices())
   device = deviceGrp.devices[0]

   localItemSize = device.getMaxWorkItemSizes()[0] as int //algo requires local
       memory to be power of two size,
   //the max first dimention gpu size is a factor of two so we use this

   InputDataSize = 1000*localItemSize as int
   System.\out.println("InputDataSize: {InputDataSize} elements")

   inputdata = device.makeOffHeapArrayIn(long[].class, InputDataSize)
   partialSums = device.makeOffHeapArrayMixed(long[].class,
       InputDataSize/localItemSize)
   localSums = gpus.Local(long[].class, localItemSize)
   data = makeData(InputDataSize)
   c1 := inputdata.writeToBuffer(data)//create some psudo random data

   inst = reduce(inputdata, partialSums, localSums)
   compute := device.exe(inst, [InputDataSize], [localItemSize], c1)//we must
       pass in the local item size for each work group
   ret = partialSums.readFromBuffer(compute)

   ctime = c1.getProfilingInfo().toString()

   //cleanup
   del inputdata, partialSums
   del c1, compute
   del deviceGrp, device
   del inst

   restum = sum(ret)//sum of the partialSums in the array slots to get our result
   verify = sum(data)//verify result on CPU

   System.\out.println('PASS! result: {restum}' if restum == verify else 'FAIL!
       result {restum} vs {verify}')
   ""
}

//////////
//Output:
//InputDataSize: 1024000 elements
//PASS! result: 1024399

//Assuming execution is taking place on a gpu with the first MaxWorkItemSize
    elemnt of 1024 .
```

The only significant change we make here in terms of execution vs a kernel with only global memory interaction is in having to provide a local work size dimension array, in the same way as we do global dimensions - `device.exe(inst, [ItemCount], [localItemSize], c1)`.

In the final phase of the algorithm above we are obtaining the resultant value by summing the values within the array read from our GPU. It is possible to perform this final step on the GPU but for the purposes of clarity in this example this has been excluded.

38.8.5 Example Kernel using local Variables

Let us now revisit our matrix multiplication example examined previously and see if we can improve performance by using local memory, specifically local variables...

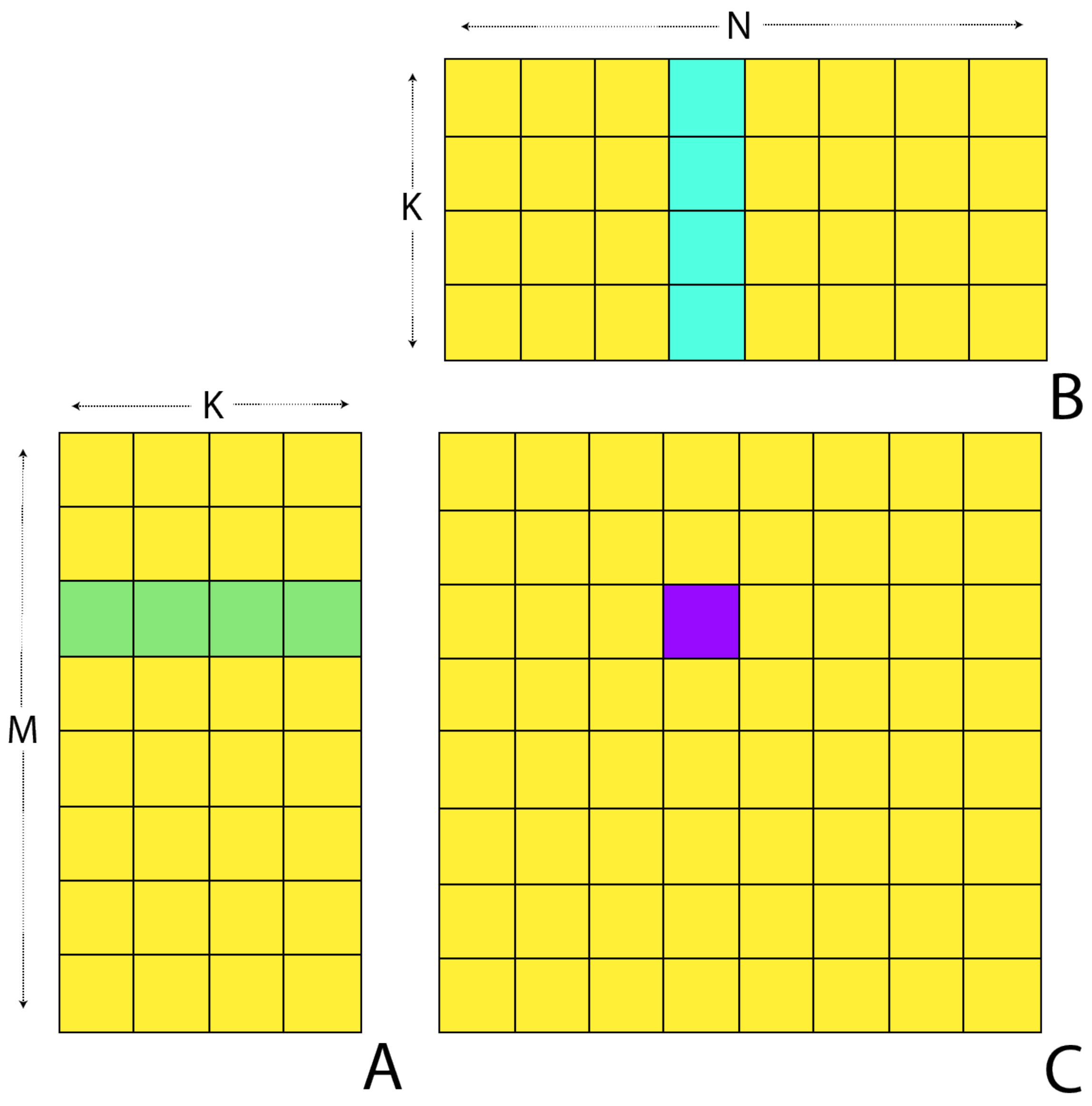


Figure 38.4: A naive algorithm for matrix multiplication

We see in 38.4 that each resulting cell is the product of each row of A and column of B summed together. We can iterate in a couple loop through each row and column of A and B respectfully. The code implementing this algorithm on the GPU and CPU is very similar and can be seen in functions `matMultNaive` and `matMultOnCPU`. It turns out that this algorithm expresses a high degree of spatial locality since a row of C is dependant on every value in B and a row of A, hence if we dedicate calculation of an element in C to a work item in our GPU we end up loading the values in A and B many multiples of times. In fact, relative to the amount of work done simply loading data (a relatively slow operation) we spend very little time doing actual computation.

We can improve on this algorithm by making use of a local cache as follows:

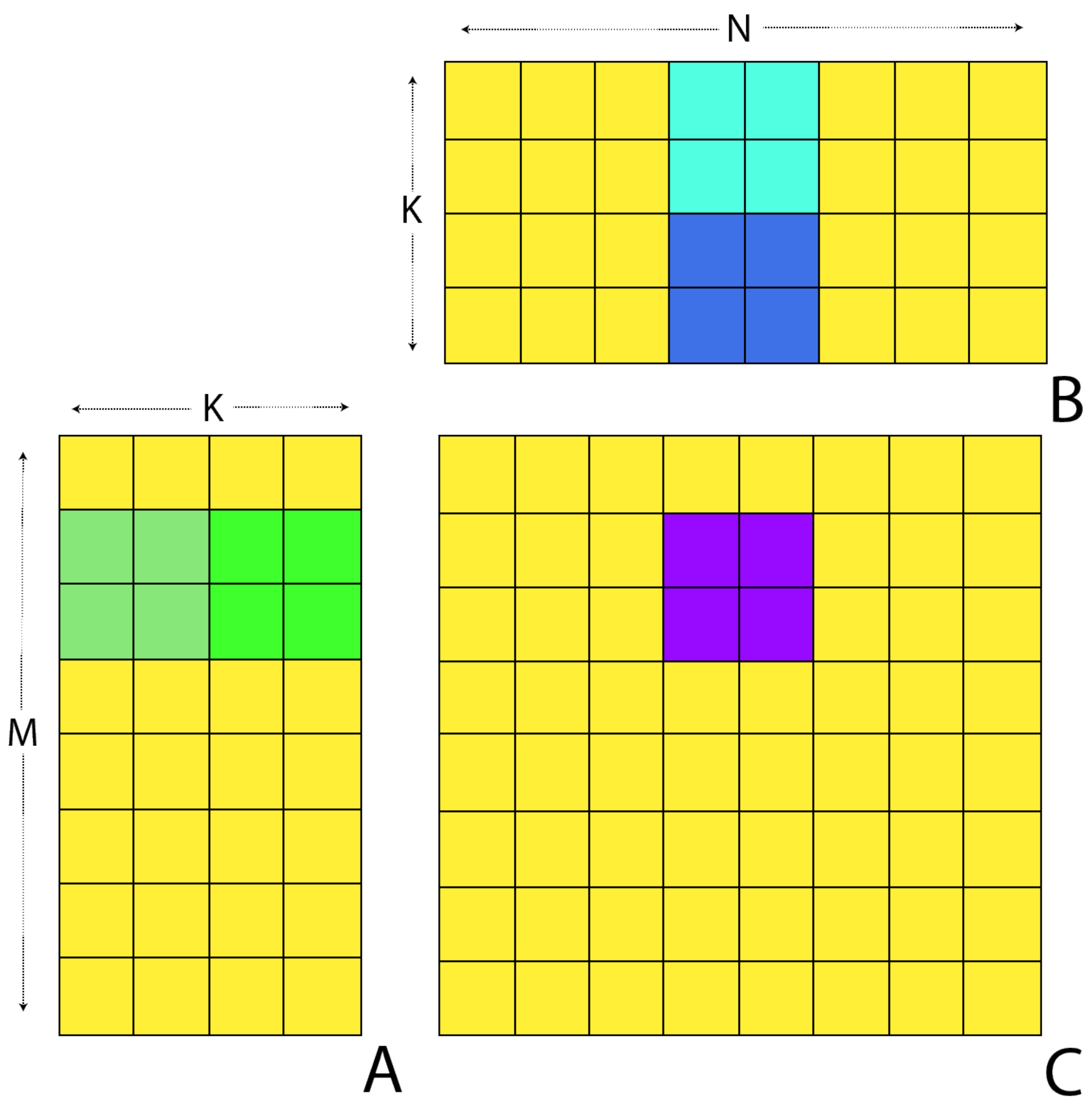


Figure 38.5: An algorithm for matrix multiplication utilizing local memory cache.

In 38.5 we see that we can calculate a block of values of C by caching in local memory two equal sized segments from A and B. The overall calculation performed by our work group is shown in 38.6:

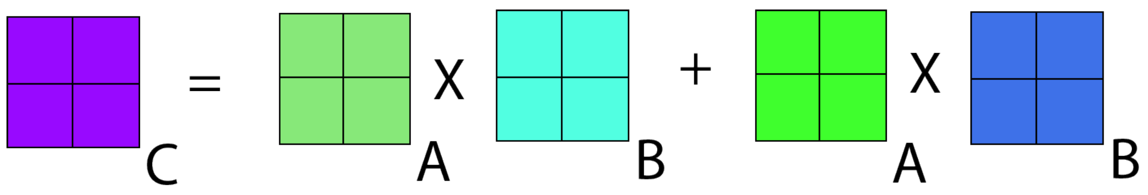


Figure 38.6: A calculation of a region of our output array C

We can see how an individual value of C is from the local cache in 38.7.

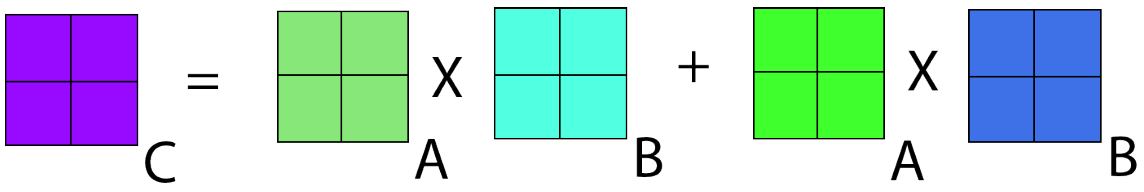


Figure 38.7: Calculation of a single element of the range in C

Let us now look at the code which implements this, in function: `matMultLocal`. Below the naive and cpu implementation of matrix multiplication are included for sake of comparison. We square a 128 by 128 float element matrix populated with pseudo random data:

```
def matMultOnCPU(M int, N int, K int, A float[2], B float[2], C float[2]){
   for (m=0; m<M; m++) {
      for (n=0; n<N; n++) {
         acc = 0.f
         for (k=0; k<K; k++) {
            acc += A[k][m] * B[n][k]
         }
         C[n][m] = acc
      }
   }
}

gpukernel 2 matMultNaive(M int, N int, K int, constant A float[2], constant B
    float[2], global out C float[2]) {
   globalRow = get_global_id(0)
   globalCol = get_global_id(1)

   acc = 0.0f;
   for (k=0; k<K; k++) {
      acc += A[k*M + globalRow] * B[globalCol*K + k]
   }

   C[globalCol*M + globalRow] = acc;
}

val CacheSize = 16

gpukernel 2 matMultLocal(M int, N int, K int, constant A float[2], constant B
    float[2], global out C float[2]) {
   row = get_local_id(0)
   col = get_local_id(1)
   globalRow = CacheSize*get_group_id(0) + row //row of C (0..M)
   globalCol = CacheSize*get_group_id(1) + col //col of C (0..N)

   //local memory holding cache of CacheSize*CacheSize elements from A and B
   local cacheA = float[CacheSize, CacheSize]
   local cacheb = float[CacheSize, CacheSize]

   acc = 0.0f

   //loop over all tiles
   cacheSize int = K/CacheSize
   for (t=0; t<cacheSize; t++) {
      //cache a section of A and B from global memory into local memory
      tiledRow = CacheSize*t + row
      tiledCol = CacheSize*t + col
      cacheA[col][row] = A[tiledCol*M + globalRow]
      cacheb[col][row] = B[globalCol*K + tiledRow]

      barrier(true)//ensure all work items finished caching

      for (k=0; k<CacheSize; k++) {//accumulate result for matrix subsections
         acc += cacheA[k][row] * cacheb[col][k]
      }

      barrier(true)//ensure all work items finished before moving on to next
          cache section
   }

   C[globalCol*M + globalRow] = acc
}

def chooseGPUDeviceGroup(gpuDev gpus.DeviceGroup[], cpuDev gpus.DeviceGroup[]) {
   if(not gpuDev){//no gpu return first cpu
      cpuDev[0]
   }else{//intel hd graphics not as good as nvidea dedicated gpu or ati so
       deprioritize
      for(grp in gpuDev){
         if('Intel' not in grp.platformName){
            return grp
         }
      }
      gpuDev[0]//intel device then...
   }
}

def createData(xy int, range = 10){
   ret = float[xy, xy]
   random = new java.util.Random(654L)
   for(n = 0; n < xy; n++){
      for(m = 0; m < xy; m++){
         ret[n,m] = random.nextInt(range+1)//1. ... 10. inclusive
      }
   }
   ret
}

def main(){
   gps = gpus.GPU()
   cpus = gps.getCPUDevices()
   deviceGrp = chooseGPUDeviceGroup(gps.getGPUDevices(), cpus)
   device = deviceGrp.devices[0]

   xy = 128//assume square matrix with 128 * 128 elements

   matrixA = device.makeOffHeapArrayIn(float[2].class, xy, xy)
   resultNaive = device.makeOffHeapArrayOut(float[2].class, xy, xy)
   resultLocal = device.makeOffHeapArrayOut(float[2].class, xy, xy)

   data = createData(xy)//blocking operation

   matrixA.writeToBuffer(data)

   inst1 = matMultNaive(xy, xy, xy, matrixA, matrixA, resultNaive)
   inst2 = matMultLocal(xy, xy, xy, matrixA, matrixA, resultLocal)
   comp1 := device.exe(inst1, [xy xy], null)
   comp2 := device.exe(inst2, [xy xy], [CacheSize CacheSize])

   resNative = resultNaive.readFromBuffer()
   resLocal = resultLocal.readFromBuffer()

   conventional = float[xy, xy]
   tick = System.currentTimeMillis()
   matMultOnCPU(xy, xy, xy, data, data, conventional)
   toc = System.currentTimeMillis()

   System.\out.println('Time to compute on CPU: ' + (toc - tick) + "ms")
   System.\out.println('Naive ' + ('PASS ' if resNative == conventional else
       'FAIL ') + comp1.getProfilingInfo())
   System.\out.println('Local ' + ('PASS ' if resLocal == conventional else
       'FAIL ') + comp2.getProfilingInfo())

   //cleanup
   del matrixA, resultNaive, resultLocal
   del deviceGrp, device
   del inst1, inst2
   del comp1, comp2
}
```

Output when running on a single core of a Intel Core i7-3770K (quad core) CPU and 480 parallel work items of one NVIDIA GeForce GTX 570 (i.e. of a comparable generation):

```
Time to compute on CPU: 20ms
Naive PASS Work Duration: 0.293ms
Local PASS Work Duration: 0.051ms
```

Observe that we create the local cache, local variable, within the `matMultLocal` kernel itself instead of passing it in.

What is interesting here is the effect of our caching optimization. Armed with our profiling information (see above), we can measure the impact of this. Execution on a single core CPU takes 20 milliseconds, our naive implementation on the other hand takes 0.293 of a millisecond which is 68x quicker. We achieve a further improvement of almost 6x on this when we use local memory implementation. This brings our overall speed improvement to almost 400x over a single core CPU! Certainly worth the extra work.

Of course, there are further optimizations we can add to further increase this, and the algorithm needs to be adjusted to deal with inputs which don't divide nicely into our cache size, but for the sake of brevity we shall leave this here.

38.9 Explicit memory management

Since we are working with non heap managed memory on the GPU, and because there generally is very little of it (relative to general purpose RAM where our heap resides, and other persistent storage such as disk drives) it is impractical to offer garbage collection of it. Thus we must explicitly delete/free GPU resources either by calling the delete method on the a relevant object, or by using the del keyword. Concurnas' Garbage collection mechanism will not automatically free these resources.

The types of object which we must explicitly free one we have finished with them are:

- **Buffers**: `gpus.GPUBufferManagedInput`, `gpus.GPUBufferManagedOutput`, `gpus.GPUBufferManagedMixedSingle`, `gpus.GPUBufferManagedInSingle`, `gpus.GPUBufferManagedOutSingle`, `gpus.GPUBufferManagedMixedSingle`. Note that Local buffers do not need deletion as they are not allocated nor allocatable outside of the GPU.
- **DeviceGroups**: `gpus.DeviceGroup`
- **Devices**: `gpus.Device`
- **Kernels**: `gpus.Kernel`
- **GPURefs**: Returned from asynchronous non blocking calls from:
  - All Buffer memory interaction functions.
  - Device kernel execution `Device.exe(...)`

In the case of GPURefs. If one deliberately negates or forgets to capture the resultant GPURef from an asynchronous non blocking GPU call then in addition to the calling code waiting for the call to complete (thus turning the call into a blocking call), the delete method will be called on the GPURef upon completion, thus freeing the memory allocated to manage the ref. This is a deliberate and very convenient mechanism for avoiding memory leaks.

This can be achieved either by explicitly calling the delete method on these objects, or by using the del keyword.

```
device.delete()
del deviceGroup, kernel
```

38.10 Finishing

It can be useful to explicitly wait for a device to finish all current queued execution (buffer reading, writing, kernel execution), the finish method can be invoked for this purpose which will block until all work has been completed:

```
device.finish()
```

Calling del on a device will also implicitly call finish, thus ensuring that all work on the device has been completed before being freed.

38.11 Double Precision

Concurnas permits the use of single (float) and double (double) precision floating point types in gpu kernels and gpu functions[3]. It should be noted however that operations performed on double precision type are considerably slower than the equivalent with float type. Thus when possible one should strive to use floating point type over double.

[3]Normally, OpenCL implementations require an explicit declaration to perform operations on data of double precision type. But in Concurnas this is provided automatically, no explicit declaration is needed.

When creating arrays care should be taken to explicitly declare floating point values as float as the default instantiation type for a floating point literal values is double.

38.12 Concurrent use of GPUs

The GPU interaction related classes under `com.concurnas.lang.gpus` (including, buffers, device groups, devices, kernels and gpurefs) are marked as transient.This is deliberate as it prevents accidental sharing of gpu related objects between differing iso's, which, if permitted would make state management challenging in most systems. It also avoids problems with attempting to persist gpu objects off heap, which doesn't make sense for gpu interaction objects.

In cases where multiple iso's require access to the GPUs at the same time, using an actor is advised, this permits one a fine degree of control over the GPUs available both from the perspective of pipelining of requests to them, and from the perspective of easily managing memory.

38.13 Notes

The reader who is aware of the inner workings OpenCL will no doubt find many aspects of the structure of the Concurnas implementation of GPU computing very familiar. This is deliberate. By not deviating to far away from "the way things are done" currently with the API exposed in raw OpenCL we hope to make GPU computing something which is natural to do for both OpenCL veterans, whilst simplifying the implementation details enough to make working with GPUs easy for newcomers.

Additionally, in the interests portability, we have omitted the tools and techniques exposed in OpenCL from version 2.0 onwards due, mainly to lack of compatibility with NVidia drivers, which for most of the older GPUs are only OpenCL 1.2 compliant.

The tiger said, "Excuse me, but I'm very hungry. Do you think I could have tea with you?"

- Judith Kerr. (1968). The Tiger Who Came to Tea. Harper Collins.

39. Distributed Computing

So far we have been looking at concurrent computing on a single physical machine. A natural extension of concurrent computing is distributed computing, where by computation is performed at a physical location which differs from that of the origin of the request to do so.

In time, as the demands placed upon one's software increase, it is often necessary to scale one's solution beyond more than one machine. Additionally, from an architectural/cost perspective, it is often most sensible to centralize resources, whether it be data (in the form of databases) or high performance hardware (such as gpu cluster machines, multi physical cpu machines etc), and have clients access that resource in a controlled manner on a service basis (with associated monitoring, SLA's etc). With Concurnas this is made easy since distributed computing is treated like a first class citizen in the language.

Since distributed computing is by nature non local to one's physical machine originating the need to perform distributed computing, a communication network must be utilized in order to initiate, coordinate and resolve that computation. This of course introduces a whole set of engineering problems which need to be overcome in order to create reliable software. Whereas with localized concurrent computing we need only concern ourselves with application level errors (errors in our logic or local parts of our computer we make use of), with distributed computing we must factor in inevitable errors of the communication network we are utilizing. Luckily Concurnas abstracts away most of the tedious boilerplate hard work allowing one to focus on what matters most.

Whereas with a localized machine, we are unlikely to receive notification (and therefore be able to react to) a catastrophic unexpected error since it would most likely manifest in the form of our entire machine failing or it being in such a state so as to render it unusable thereafter. But, as with distributed computing, when we are utilizing a communication network we must factor in this source of error, as it is at least an order of magnitude or three (10x - 1,000x) more probable to occur. For example network failure or the physical machines we're trying to make use of becoming unavailable (scheduled maintenance/downtime etc). To this effect we will see that Concurnas has special first class citizen syntactical support requiring capture and response to these forms of expected, unpredictable failure.

39.1 Creating a remote request

We can connect to a remote host and perform remote computation via a remote isolate in much the same way as we can create a new isolate. The syntax is both elegant and concise. Let's look at an example:

```
rm = Remote('localhost')//establish a connection to a remote Concurnas server
rm.connect()//optionally called
result int: = {1+1}!(rm.onfail(e=>e.retry()))//result => 2
rm.close()//close wait for result of our isolate above to be set
```

Alternatively we can define the isolate to execute as follows:

```
rm = Remote('localhost')
rm.connect()//optionally called
def func() => 1+1
result int: = func()!(rm.onfail(e=>e.retry()))//result => 2
rm.close()//close wait for result of our isolate above to be set
```

In the above cases, first we establish a connection to a remote Concurnas server by creating a `Remote` connector with a hostname/ip address and optional port (which defaults to `42000` if unspecified): `Remote('localhost')`. Note that Remote is automatically imported.

We next call `connect()` to establish a network connection to our remote server. This is an optional step, it will be called upon first submission of an isolate if it a connection has not already been established.

Next we attempt to submit the isolate `{1+1}!` for execution remotely. This occurs as normal but except for us passing an executor on the right hand side of the bang `!` operator. This executor is returned from the remote objects `onfail` method, which obliges the remote request initiator to handle the error case of a isolate task submission failing at inception by passing an object of type `com.concurnas.dist.RemoteFailureHandler` to handle this occurrence. `RemoteFailureHandler` is a SAM type thus we are able to use the compact lambda syntax to pass in a handler above. Our error handling strategy above is to simply re-attempt to submit the isolate task if it cannot be initially submit or a server side execution error occurs.

An object of type `com.concurnas.dist.RemoteFailureContext` is passed to the provided `RemoteFailureHandler`. This contains the following noteworthy utility methods:

- `retry()` - Attempt to re run the isolate.
- `result()` - Returns the ref to which the return value of the remotely spawned isolate would have its return value assigned. Returns null if the isolate does not return a value.
- `reason()` - Returns the Exception indicating the cause of failure.

The error handler can even be stateful, for example:

```
from com.concurnas.dist import RemoteFailureContext, RemoteFailureContext

rem = Remote('localhost')
result = {1+1}!(rem.onfail( new RemoteFailureHandler{
   attempts = 1

   def handle(e RemoteFailureContext){
      if(attempts++ > 10){//retry 10 times then give up
         e.result():setException(e.reason())
      }else{
         e.retry()
      }
   }
} ))
//result == 2
```

If the error handler itself throws an exception, then this exception will be set on the ref returned from the isolate. If no ref is returned from the isolate, then this exception will be passed to the default isolate error handler (this is generally not a desirable situation).

Remember that an exception occurring within the bounds of the isolate itself, does not constitute a distributed computing communication (or otherwise) failure, and is recorded on the exception value of the ref as normal - the error handler discussed here does not catch this.

Being an isolate, of which the resultant type is a ref, allows us to otherwise carry on with execution asynchronously until the point where the value of the ref is required. This is an incredibly powerful mechanism, it's feasible to envisage hundreds or even thousands of these remote requests being spawned across thousands of machines in order to solve a complex problem.

When we are finished with the remote connection we can call the `close()` method in order to wait for the result of any pending isolates and finally disconnect from the remote server once the results (positive or an exception) are known. By implementing the close method in this way, we can use remote connections within try-with-resources blocks. For example:

```
try(rm = Remote('localhost')){//close rm on completion of this block
   rm.connect()//optionally called
   result int: = {1+1}!(rem.onfail(e=>e.retry()))//result => 2
}
```

A second variant of the `close()` method exists for remote connections. The `close()` method takes a boolean parameter, `close(hard boolean)`. If this is set to `true` then the connection will be closed immediately, any outstanding remotely spawned isolates will have an exception set on their resultant ref:

```
rm = Remote('localhost')
rm.connect()//optionally called
result int: = {1+1}!(rem.onfail(e=>e.retry()))//result => 2
rm.close(hard = true)//do not wait for the result above to complete
```

In the above case of an explicit hard disconnection any outstanding isolates will have an `ClientPrematureDisconnection` exception set on their result refs.

Some care should be taken with hard disconnections, since we are not waiting for the result of our submit isolate before closing our connection. Closing a connection will prevent as yet unstarted remote isolates from starting.

It is essential to call the close method when one is finished using the remote connection because just like other forms of io (e.g. files), failing to do so will keep the connection open resulting in a resource leak.

39.1.1 Common error handling techniques

The `com.concurnas.dist.Remote` class comes with a number of pre packaged error handling mechanism. Here are two notable ones:

- `onFailRetry()` - retry a remote request a fixed number of times (by default, 10) before giving up and setting the last reason for failure as the exception on the ref returned by the isolate.

- `onFailFail()` - If the initial execution attempt fails, give up and set the provided exception on the ref returned by the isolate.

39.1.2 Network errors after submission

So far we have covered errors occurring up to an isolate being successfully submit to, and executed by, a remote executor. Once the isolate has been received at the remote server and acknowledged as such a client remote connection will automatically reconnect to the remote server in the case of any networking disconnections.

Submit isolates will be executed by the remote server even in the case of unexpected client disconnection, the result of said computation will be cached until the client reconnects or the remote server itself is closed. If a client explicitly hard closes a connection, then any outstanding isolates will be terminated.

39.2 Creating a remote server

A remote server can easily be created in Concurnas by using the following code:

```
from com.concurnas.lang.dist import RemoteServer
rmServer = RemoteServer()
remServer.startServer()//start remote server
//...use remote server...
rmServer.close()
```

A port may optionally be specified, if not then the server will be spawned on port: `42000`.

It is essential to call the close method when one is finished using the remote server because just like other forms of I/O (e.g. files), failing to do so will keep all its client connections open resulting in a resource leak.

39.2.1 Security Managers

Since when we are creating a remote server we are opening up our machine for external access. It's often beneficial to exert some additional control over this code which we may not fully trust. To this end one can pass a `java.security.PermissionCollection` to the `com.concurnas.lang.dist.RemoteServer` instance. For example:

```
from java.security import PermissionCollection, Permissions, AllPermission
perms = Permissions()
perms.add(new AllPermission())

remServer = RemoteServer(permissions = perms)
```

In the above example, in using a `AllPermission` object we are giving our remote code the privileges as the spawned server executing it. In practice this scope can of course be narrowed. For more details on security managers see here: Concurnas Security Managers and here: Java security managers

In the case where no security policy is defined above, remote code will be executed with the same privileges as the spawned server.

39.2.2 Request Dependencies

Distributed computing in Concurnas is agnostic to the origin of the source code being submit. Remote servers are treated almost like dumb terminals though usually with a tremendous amount of compute power and exposure of special resources (databases, custom hardware etc). In practice this means that a connecting client may submit whatever code they like, with whatever code dependencies they deem appropriate.

The server will interact with the requesting client so as to obtain a copy of the bytecode required in order to execute the service if it doesn't already have a copy locally for the connected client. This is part of the protocol implemented by the client and server components of the distributed computing framework in Concurnas and occurs automatically behind the scenes. Any specified code will be executed by the remote server provided that it complies with any security policies defined. The dependency provision protocol itself is quite sophisticated and in the interests of performance is able to perform static code dependency analysis in order to determine and preemptively provide upfront the code that is required in order to execute a request.

V

Tools

40. Compiling Concurnas code

Concurnas code can be compiled via the `concc` command line tool or via IDE which has support for Concurnas. Here we shall examine the `concc` command line tool. We can compile a `.conc` file as follows:

```
D:\work>concc hello.conc
```

The `concc` command line tool has the following syntax:

```
concc options sourcefiles/directories
```

There must be at least one sourcefile or directory provided as an argument. Source code files must have the `.conc` suffix to be compilable. If a directory reference is provided then this will be searched, including nested directories, for `.conc` files. Multiple entries can be delimited with a space.

If any `.conc` file has a dependency on a `.conc` file listed in the same directory, then the dependant `.conc` file will also be compiled.

By default the `.conc` files will be compiled in to one or more .class file per source file input, into the same directory as the original sourcefile. To override the output directory of the .class files, use the -d option described below. Where the output directory has been overridden, if it does not already exist, Concurnas will attempt to create it.

The directory hierarchy relative to the working directory of the `concc` call denotes the package name of the output .class files for `.conc` files specified in the sourcefiles. The root directory is the current working directory in which `concc` is run. The root working directory can be overridden using the -root option described below. All referenced source files are taken to be relative to this directory.

Where a directory is specified for compilation, the root of the directory is taken to be root for nested `.conc` files for the purposes of package name generation. The exception being where the specified directory path is fully qualified, in which case the base of the nested hierarchy will be taken as the root directory for package name generation purposes.

Sourcefiles and directories may also be prefixed with an element specific root directory. This will override the global root directory (if specified) or default working directory. The syntax for this is as follows:

```
root[sourcefiles or directories]
```

Using an element specific root is generally the easiest method by which one can perform compilation with correct package names for multiple source files existing across multiple directories.

40.1 Options

There can be zero to many options specified on a call to `concc`

- **-d directory**: Override the directory where .class files will be output. concc will try to create this directory if it does not already exist.
- **-a** or **-all**: Copy all non .conc files from source directories to provided output directory (output directory is overridden with the **-d** option).
- **-jar jarName[entryPoint]?**: Creates a jar file from all generated/copied files. Optional entryPoint class name may be specified which will be used to populate a META-INF/MANIFEST.MF file within the jar. Can only be used in conjunction with the **-d** option.
- **-c** or **-clean**. Remove all generated files from output directory. Useful in conjunction with -jar option. Any generated jar files will not be removed.
- **-classpath path** or **-cp path**: The classpath option enables one to override the CLASSPATH environment variable. Elements are delimited via ; and must be surrounded by `""` under Unix based operating systems. Elements may consist of directories, `.class` file references or `.jar` file references.
- **-verbose**: Provide additional compilation information including bytecode output.
- **-root directory**: Override the root directory of `javacc`. Package names for `.conc` files will be generated relative to this.
- **-werror**: Normally, warnings do not prevent class file generation, setting this tag will treat warnings as errors, thus preventing class file generation warnings are generated for the input `.conc` source file in question.
- **-J**: JVM argument. Prefix jvm arguments with -J in order to pass them to the underlying JVM. e.g. `-J-Xms2G -J-Xmx5G`.
- **-D**: System property. Prefix system properties with -D in order to pass them to the underlying JVM. e.g. `-Dcom.mycompany.mysetting=108`.
- **–help**: List documentation for options above.

It is recommended that the **-d directory** option be used for all but the most trivial of compilations.

40.2 Examples

Compile all code in `.src` relative to default working directory:

```
concc ./src
```

Compile all code in `.src` relative to default working directory but override the output directory, writing output classes to `/classes`:

```
concc -d ./classes ./src
```

As par previous example and (via `-a` option) copying all non .conc files to `/release` directory:

```
concc -a -d ./release ./src
```

As par previous example but additionally packaging all files in output directory into a jar: `release.jar` and removing all copied files from the output directory except the output jar, `release.jar`:

```
concc -a -clean -jar release.jar -d ./release ./src
```

As par previous example but additionally specifying an entry point (`com.mycompany.product.Start`) the main method of which will be executed in the event of execution via the `conc` command:

```
concc -a -clean -jar release.jar[com.mycompany.product.Start] -d ./release ./src
```

Compile all code in `.src` relative to default working directory and `Mycode.conc` in `extrasrc`.

```
concc ./src ./extrasrc/Mycode.conc
```

As above but override the root of `Mycode.conc` to be `extrasrc` (thus the package name of the code in `Mycode.conc` will not be prefixed with `extrasrc`):

```
concc ./src ./extrasrc [Mycode.conc]
```

As above but override the output directory, writing output classes to `/classes`:

```
concc -d ./classes ./src ./extrasrc [Mycode.conc]
```

As above but with custom libraries (jars and classes) set on the classpath:

```
concc -cp ./include/MyLib.jar;./include/UtilClass.class -d ./classes ./src
    ./extrasrc [Mycode.conc]
```

40.3 JVM arguments

Arguments (for example garbage collection and memory allocation tuning) may be applied to the underlying JVM executing `concc` by prefixing them with `-J`. For example:

```
concc -J-Xmx2g ./src ./extrasrc/Mycode.conc
```

40.4 System properties

System properties may be specified on the command line when running `concc` by prefixing them with `-D`. For example:

```
concc -Dcom.mycompany.mysetting=108 ./src ./extrasrc/Mycode.conc
```

40.5 Adding concc to the path

It is recommended that the `<installDir>/bin` be added to the system path in order to permit `concc` to be run from any path. Check your operating system documentation for details of how to do this. The Windows installer for Concurnas will perform this automatically, as will installation via the SDKMAN! platform.

40.6 Jar files

Jar's are an excellent way of packaging code for organization and distribution. As seen above the `concc` tool is able to create these as described previously. Additionally the jar tool provided as part of the JDK can be used to achieve this objective.

40.7 Notes

The first time the `concc` tool is executed, or when a new JVM environment upon which is its being executed is detected, an 'installation' will take place wherein Concurnas will set itself up to execute efficiently upon said JVM environment by caching a copy of the JDK. This can take up to a few minutes to complete.

41. Running Concurnas Programs

Compiled programs written in Concurnas may be executed via the `conc` tool. For example, in conjunction with the `concc` tool for compilation:

Using example code, `hello.conc`:

```
System.out.println("Hello World!")
```

We can compile and execute this as follows:

```
D:\work\bin>concc hello.conc
D:\work\bin>conc hello
Hello World!
D:\work\bin>
```

The syntax for the `conc` tool is as follows:

```
conc options* entryPoint? cmdLineArguments*
```

Let us now look in detail at what is taking place above:

41.1 Running code with Concurnas

Compiled Concurnas code can be run with Concurnas via the `conc` command which comes as standard as part of the Concurnas distribution. We can also run `conc` in interactive Read-Evaluate-Print Loop (REPL) mode, this is explored in the REPL chapter. Here we focus upon non-interactive execution of pre compiled code. If `conc` has not been added to the path (the default during installation), the tool may be started from within that directory. If Concurnas has not already completed its installation for the detected JDK which it is being run under, it will first complete that process (this may take a few minutes).

We use the `conc` command with an entry-point class or jar file reference in order to start `conc` in non-interactive execution mode:

```
D:\work>conc myClass
D:\work>conc myClass.class
D:\work>conc myJar
D:\work>conc myJar.jar
```

A full list of optional options for `conc` is found under the Command line options section.

41.2 Bootstrapping Code

In order to run a program with Concurnas via the `conc` tool we must pass an entry-point class or jar file reference. For a jar file to be used in this manner it requires a class within it to be nominated as its entry point. This entry point reference is then examined and one of the following is executed in order of precedence:

1. `def main(args String[])void`
2. `def main()void`
3. Top level code

We see above that specifying a `main` method within our entry-point class is not a requirement, and if one is not defined then this will result in any top level code of the class file being invoked upon execution instead. However, using a `main` method is recommended in cases where we wish to use a referenced piece of code as both a library and program. This is because, where the code within the .class file is used like a library, upon first execution of something imported from that library class, all the top level code will be executed, including our code we wish to execute as a program - something which is rarely desirable. An explicit `main` method on the other hand must be explicitly invoked, thus avoiding this problem.

The `main` method declaration will receive any command line parameters passed to the program upon execution if its signature is declared as: `def main(args String[])void`.

41.3 Command line options

There can be zero to many options specified on a call to `conc`:

- **-classpath path** or **-cp path**: The classpath option enables one to override the CLASSPATH environment variable. Elements are delimited via ; and must be surrounded by `""` under Unix based operating systems. Elements may consist of directories, `.class` file references or `.jar` file references.
- **-s**: Server mode. When running in non interactive mode (i.e an entry point is provided) Concurnas will not terminate the process upon the entry point main method (or alternative) completing execution.
- **-bc**: Print bytecode. This option is applicable when running in REPL mode. Print the bytecode generated from the provided input.
- **-J**: JVM argument. Prefix jvm arguments with -J in order to pass them to the underlying JVM. e.g. `-J-Xms2G -J-Xmx5G`.
- **-D**: System property. Prefix system properties with -D in order to pass them to the underlying JVM. e.g. `-Dcom.mycompany.mysetting=108`.
- **–help**: List documentation for options above.

41.4 Examples

41.4.1 Single file programs

We can compile and execute our simple hello world (named littlehello.conc) as follows:

```
D:\work>concc littlehello.conc
Finished compilation of: D:\work\littlehello.conc -> D:\work\littlehello.class
    [littlehello]

D:\work>conc littlehello
hello world

D:\work>
```

In practice it is very rare that we will write single file programs.

41.4.2 Multi file programs

Lets take a slightly more involved hello world example (named `hello.conc`) which compiles into multiple class files thus requiring us to make of the classpath argument:

```
class SayHello{
   def do(args String[]){
      System.err.println("hello world received args: "+ args)
   }
}

def main(args String[]){
   new SayHello().do(args)
}
```

We can compile and execute this code as follows:

```
D:\work>concc hello.conc
Finished compilation of: D:\work\hello.conc -> D:\work\hello$SayHello.class
    [hello$SayHello], D:\work\hello.class [hello]

D:\work>conc -cp ./ hello
hello world got args: null

D:\work>
```

Notice above that we must make use of the `-cp` argument in order to specify the current directory as holding both our entry point class: `hello.class` and supporting `SayHello` class: `hello$SayHello.class`.

Note that whilst it is best practice to specify a `-cp` argument, in the above case it may be omitted as Concurnas will automatically add the current directory to the classpath in instances where no `-cp` argument is provided and the entry class is within the current working directory[1]:

```
D:\work>conc hello
hello world got args: null

D:\work>
```

We can pass command line arguments to our hello world class since its main method takes a String array argument:

```
D:\work>conc -cp ./ hello there
```

[1]This feature has been added so as to aid newcomers to the programming language

```
hello world got args: [there]

D:\work>
```

41.4.3 Multi file programs in jar's

In practice it is often most convenient for the purposes of software distribution and organization, to package code up within jar's, these can be executed via the use of the `-jar` argument as illustrated here. Let's take our `hello.conc` we previously implemented:

```
D:\work>concc -jar helloWorld[hello] hello.conc
Finished compilation of: D:\work\hello.conc -> D:\work\hello$SayHello.class
    [hello$SayHello], D:\work\hello.class [hello]

D:\work>conc helloWorld.jar
hello world got args: null

D:\work>conc helloWorld.jar there
hello world got args: [there]

D:\work>
```

41.4.4 Running programs in server mode

Normally, when running in non-interactive mode (i.e an entry point is provided) Concurnas will terminate the process upon the entry point main method (or alternative) completing execution. To suppress this behaviour, for instance, for a server application, we use the `-s` argument as follows:

```
D:\work>conc -s helloWorld.jar there
hello world got args: [there]
```

The above program will not terminate until it receives as a kill signal from the operating system.

41.5 Terminating Code

If we are running a Concurnas application in server mode (as par above) it can often be useful to register shutdown handlers. These are invoked upon receiving a SIGTERM signal from the operating system (check your operating system documentation for details on how to do this). More details on shutdown handlers can be found in the shutdown handlers section.

41.6 Starting `conc` in interactive REPL mode

The `conc` tool has the ability to run compiled Concurnas code as we have seen in this chapter. It also has the ability to run in interactive REPL mode. More information about running in interactive REPL mode can be found in the The Concurnas REPL section.

41.7 JVM arguments

Arguments (for example garbage collection and memory allocation tuning) may be applied to the underlying JVM executing `conc` by prefixing them with `-J`. For example:

```
D:\work>conc -J-Xmx2g helloWorld.jar there
hello world got args: [there]
```

```
D:\work>
```

41.8 System properties

System properties may be specified on the command line when running `conc` by prefixing them with `-D`. For example:

```
D:\work>conc -Dcom.mycompany.mysetting=108 helloWorld.jar there
hello world got args: [there]

D:\work>
```

41.9 Adding conc to the path

It is recommended that the `<installDir>/bin` be added to the system path in order to permit `/conc` to be run from any directory. Check your operating system documentation for details of how to do this. The Windows installer for Concurnas will perform this automatically, as will installation via the SDKMAN! platform.

41.10 Notes

The first time the `conc` tool is executed, or when a new JVM environment upon which is its being executed is detected, an 'installation' will take place wherein Concurnas will set itself up to execute efficiently upon said JVM environment by caching a copy of the JDK. This can take up to a few minutes to complete.

42. The Concurnas REPL

Concurnas offers a lightweight tool for programming and executing with Concurnas code. The standard Concurnas runtime distribution can be run in either non-interactive or interactive Read-Evaluate-Print Loop (REPL) mode via the command line tool: `conc`. Here we shall examine this REPL mode. The Concurnas REPL is a handy tool for learning about Concurnas and trying out ideas in a scriptable manner. It evaluates declarations, statements, and expressions as they are entered and immediately shows the results in an interactive fashion.

42.1 Running the Concurnas REPL

The Concurnas REPL is included as standard as part of the Concurnas distribution. We can start it using the same command as we use for executing compiled code in a non-interactive manner: `conc`. If `conc` has not been added to the path, the tool may be started from within that directory. If Concurnas has not already completed its installation for the detected JDK which it is being run under, it will first complete that process (this may take a few minutes).

We use the `conc` command without an entry-point class or jar reference in order to start `conc` in REPL mode:

```
C:\concurnas-1.14.017>conc
```

Here is an example of the sort of output we expect to see when `conc` is up and running:

```
C:\concurnas-1.14.017>conc
Welcome to Concurnas 1.14.017 (Java HotSpot(TM) 64-Bit Server VM, Java 11.0.5).
Currently running in REPL mode. For help type: /help

conc>
```

A full list of command line options for `conc` is found under the Command line options section.

42.1.1 Closing the Concurnas REPL

The ordinary way to exit the Concurnas REPL, is by entering the `/exit` command or by pressing Ctrl+D:

```
conc> /exit
```

42.2 Syntactical elements

The Concurnas REPL recognises the all elements of Concurnas syntax including: variable declarations and assignments, functions, extension functions, class (including actors and traits) definitions, type providers, imports, typedefs, usings and expressions.

Elements of Concurnas code entered into `conc` are immediately compiled and executed. Feedback about that compilation process in terms of warnings and/or errors is shown and if there are no unrecoverable errors execution of the input code will take place and the results presented.

Let us now input some code to assign a value to a variable:

```
conc> myvar = 99
myvar ==> 99
```

We see above that the variable `myvar` has been assigned the value `99`. `conc` will output all top level variables assigned in a provided expression after compilation and execution.

If a provided expression returns a value but does not assign that result to a variable, a scratch variable will be created:

```
conc> 10*10
$0 ==> 100
```

The scratch variable may be referenced in subsequent expressions for evaluation:

```
conc> res = $0 + 1
res ==> 101
```

In order to suppress the printing and creation of scratch variables, the input expression need only be terminated with a semi colon: `;`. For example:

```
conc> def timesTwo(a int) => a*2
conc> timesTwo(2);
conc>
```

We can obtain extra feedback by toggling the verbose mode on using the `/verbose` command:

```
conc> /verbose
|  Set verbose mode: on

conc> def timesTwo(a int) => a*2
|  created function timesTwo(int)

conc>
```

Notice how above we now receive the extra line of feedback upon creation or modification of a function definition: `|  created function timesTwo(int)`.

42.2.1 The continuation prompt

The Concurnas REPL provides support for structures that require mutl-line input such as functions and classes. This is achieved by showing the continuation prompt: >:

```
conc> def foo(a int, b int){
   >   a + b
   > }
conc>
```

In creating a class the effect is the same:

```
conc> class Person(firstName String, sirname String){
   >   override toString(){
   >     "Person({firstName, sirname)"
   >   }
   > }
conc>
```

42.2.2 Changing definitions

We can overwrite functions, classes and other top level definitions which have already been defined as follows:

```
conc> /verbose
|  Set verbose mode: on

conc> def times3(a int) => a*2//oops!
|  created function times3(int)

conc> def times3(a int) => a*3//fixed!
|  redefined function times3(int)
conc>
```

42.2.3 Errors and Warnings

Any errors or warnings applicable for our code will be reported back to us after evaluation of the code submit to the REPL. For example:

```
conc> concat = "My best number: " - 4
|  ERROR 1:9 in concat - numerical operation cannot be performed on type
    java.lang.String. No overloaded 'minus' operator found for type
    java.lang.String with signature: '(int)'

conc> concat = "My best number: " + 4 //that's better!
concat ==> My best number: 4

conc>
```

42.2.4 Exceptions

If we write code which throws an exception, the exception stack trace provided by the JVM upon which Concurnas runs will be formatted by the REPL in order to help us identify where the exception was thrown. In the below example we try to throw a null pointer exception (some which is made deliberately difficult through Concurnas' inclusion of nullability in its type system):

```
conc> def npeThrower(input String?){
   >   input??.length()//dangerous
   > }

conc> npeThrower("hi")//normal usage
$0 ==> 2

conc> npeThrower(null)
|  java.lang.NullPointerException
|    at npeThrower(line:2)
|    at line:1

conc>
```

42.2.5 Forward references

The Concurnas REPL permits definitions of top level elements which reference functions, variables and classes that are not yet defined. Although in defining such elements an error will be reported, as soon as all the missing dependencies are satisfied the element will be usable:

```
conc> /verbose
|  Set verbose mode: on

conc> def multiplier(what int) => what*mul
|  ERROR 1:33 in multiplier(int) - mul cannot be resolved to a variable

conc> mul=10;
|    update modified multiplier(int)

conc> multiplier(10)
$0 ==> 100

conc>
```

42.2.6 Imports

When Concurnas is operating in REPL mode we can import any code provided on the classpath and within the standard Java JDK via the usual means of import. For instance, ArrayList:

```
conc> from java.util import ArrayList

conc> mylist = ArrayList<String>()
mylist ==> []

conc>
```

42.2.7 Deleting definitions

The `del` keyword has special meaning when Concurnas is operating in REPL mode. Any defined top level element such as a variable, function or class may be removed from top level scope (and deleted) by preceding its name with the `del` keyword as follows:

```
conc> myvar = 100;
conc> myvar
myvar ==> 100

conc> del myvar

conc> myvar //we expect this not to exist anymore!
|  ERROR variable myvar does not exist

conc>
```

```
This of works for functions, classes etc:

conc> def bar() => 100

conc> bar()
$0 ==> 100

conc> del bar

conc> bar() //no longer exists!
|  ERROR 1:0 Unable to find method with matching name: bar

conc>
```

42.3 Commands

The Concurnas REPL offers a number of commands which can be of assistance when using the REPL. The full listing is viewable by using the `/help` command:

```
conc> /help
|  Type a Concurnas language expression, statement, or declaration.
|  Or type one of the following commands:
|  /help
|      to show this message
|  /exit
|      to close the REPL
|  /verbose
|      to turn verbose mode on/off
|  /bc
|      to turn bytecode listing on/off
|  /imports
|      to show all imports
|  /usings
|      to show all usings
|  /vars
|      to show all defined variables
|  /defs
|      to show all defined functions
|  /classes
|      to show all defined classes
|  /typedefs
|      to show all typedefs
|  To exit use command /exit or press Ctrl+D
```

```
conc>
```

42.4 Adding the REPL to the path

It is recommended that the `<installDir>/bin` be added to the system path in order to permit the Concurnas REPL via `conc` to be run from any directory. The Windows installer for Concurnas will perform this automatically, as will installation via the SDKMAN! platform.

VI

Others

43. Compiler Warnings

Like most good compilers, Concurnas supports error reporting. Concurnas also supports warnings in the form of a large number of compiler facets designed to help you build 'better' - less buggy, easier to read and maintain software. These warnings are for cases where the outlined problem is not so severe as to warrant compilation failure (an error), but still deserves attention for its likely pathology.

43.1 Disabling warnings

Warnings can be disabled by using the `@SuppressWarnings()` annotation. For example, say we have a redefine import warning:

```
class String(a int)//redefines the default imported class: String
```

We can disable this warning by attaching the `@SuppressWarnings()` annotation to the method or class/actor/trait encapsulating the code triggering the warning:

```
@SuppressWarnings("redefine-import")
class String(a int)//no longer triggers a warning
```

43.1.1 Disabling multiple warnings

Multiple warnings may be disabled by passing an array of warnings to disable to the `@SuppressWarnings()` annotation. For example: `@SuppressWarnings("all")`. Though this is not recommended best practice.

43.1.2 Disabling all warnings

All warnings may be disabled by using the annotation: `@SuppressWarnings(["all"])`.

43.2 Available Warnings

Concurnas offers the following list of warnings. The below strings can be used in conjunction with the aforedescribed `@SuppressWarnings()` annotation in order to disable them:

- **enum-match-non-exhaustive** - Match statement on enum is missing entry for enum element.
- **typedef-arg-use** - Argument has been defined in a typedef but omitted from the right hand side declaration.
- **generic-cast** - Attempted object cast to generic type.
- **redefine-import** - Defined class name or variable overrides existing imported class name (including auto imported).

44. Domain Specific Languages

Concurnas is a great language for building Domain Specific Languages (DSL's). DSL's enable us to define our own programming languages on top of the functionality offered by the host language in order to achieve the following objectives:

- Narrow the semantic gap between our host language and the domain in which the problems we're trying to solve persist.
- Make it easier for domain experts to express solutions to problems without requiring full knowledge of the host programming language.
- Reduce time to market of solutions.
- Reduce the amount of code we need to write and maintain.

In this chapter we shall briefly summarize the aspects of Concurnas which make achieving the above objectives possible.

44.1 Operator overloading

Operator overloading helps us reduce the amount of code one would otherwise be required to write and reduces the distance between the problem domain we are operating in, and the underlying language one is programming in - since one may use normal operators to achieve computation.

```
class Complex(real double, imag double){
   def +(other Complex) => new Complex(this.real + other.real, this.imag +
       other.imag)
   def +=(other Complex) => this.real += other.real; this.imag += other.imag
   override toString() => "Complex({real}, {imag})"
}

c1 = Complex(2, 3)
c2 = c1@//deep copy of c1
c3 = Complex(3, 4)

result1 = c1 + c3 //result1 == Complex(5.0, 7.0)
c2 += c3 //compound plus assignment, c2 == Complex(5.0, 7.0)
```

44.2 Extension functions

Extension functions allow for functionality to be added to classes without needing to interact with the existing class hierarchy (e.g.extending the class etc). They are a convenient alternative to having to use utility functions/methods/classes which take an instance of a class and often permit a more natural way of interacting with objects:

```
def String repeat(n int) String {//this is an extension function
   return String.join(", ", java.util.Collections.nCopies(n, this))
}

res = "hi".repeat(2) //creates a reference
rep() // returns: 'value, value'
```

Extension functions may be defined on any type including primitive types:

```
def int meg() = this * 1024L * 1024L

12.meg() //-> 12582912L
```

44.3 Expression lists

Concurnas provides expression lists, this is a neat feature which enables a more natural way of writing expression related code that would otherwise have to be written as a set of chained together calls using the dot operator and/or function invocation brackets.

Expression lists may be combined with operator overloading and extension functions to achieve some very impressive results. Using this feature one can create very succinct and readable DSLs:

```
from java.time import Duration, LocalDateTime

enum Currency{ GBP, USD, EUR, JPY }
enum OrderType{Buy, Sell}

class CcyAmount(amount long, ccy Currency){
   override toString() => "{amount} {ccy}"
}

abstract class Order(type OrderType, ccyamount CcyAmount){
   when LocalDateTime?

   def after(dur Duration){
      when = LocalDateTime.now()+ dur
   }

   override toString() => "{type} {ccyamount} at {when}"
}
class Buy(ccyamount CcyAmount) < Order(OrderType.Buy, ccyamount)
class Sell(ccyamount CcyAmount) < Order(OrderType.Sell, ccyamount)
//extension functions
def long gbp() => CcyAmount(this, Currency.GBP)
def int mil() => this*1000000L
def int seconds() => Duration.ofSeconds(this)


///////////Expression list:
order = Buy 1 mil gbp after 10 seconds

orderStr = "" + order
//orderStr == Buy 1000000 GBP at 2018-02-15T06:32:11.099
```

44.4 Language extensions

Language extensions are the most powerful, but also most difficult to implement way to provide DSL's. Although most often the three previously mentioned methods of creating DSL's suffice, we have language extensions for those instances where the functionality offered is not enough.

We can support virtually any syntax and semantics from an existing or new programming language within a language extension. Furthermore, this functionality may be provided at any point within a Concurnas program. But we must do most of the hard work in terms of language compilation to achieve these results.

Here is an example language extension implementation from the Language extensions chapter:

```
from com.concurnas.lang.LangExt import LanguageExtension, Context, SourceLocation
from com.concurnas.lang.LangExt import ErrorOrWarning, Result, IterationResult
from com.concurnas.lang.LangExt import Variable, Method, MethodParam
from java.util import Stack, Scanner, ArrayList, List

/*
* sample inputs:
*  (+ 1 2 3 )
*  (+ 1 2 n )
*  (+ 1 2 ( * 3 5) )
*  (+ 1 2 ( methodCall 3 5) )
*/

///////////////// AST /////////////////
enum MathOps(code String){PLUS("+"), MINUS("-"), MUL("*"), DIV("/"), POW("**");
   override toString() => code
}

open class ASTNode(-line int, -col int){
   nodes = new ArrayList<ASTNode>()
   def add(toAdd ASTNode){
      nodes.add(toAdd)
   }

   def accept(visitor Visitor) Object
}

class LongNode(line int, col int, along Long) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}

class MathNode(line int, col int, what MathOps) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}
```

```
class MethodCallNode(line int, col int, methodName String) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}

class NamedNode(line int, col int, name String) < ASTNode(line, col){
   def accept(vis Visitor) Object => vis.visit(this);
}


///////////////// Parser /////////////////

def parse(line int, col int, source String) (ASTNode, Result) {
   rootNode ASTNode?=null

   warnings = new ArrayList<ErrorOrWarning>()
   errors = new ArrayList<ErrorOrWarning>()

   sc = new Scanner(source)
   nodes = Stack<ASTNode>()

   while(sc.hasNext()){
      if(sc.hasNextInt()){
         llong = sc.nextLong()

         if(nodes.isEmpty()){
            errors.add(new ErrorOrWarning(line, col, "unexpected token: {llong}"))
         }
         else{
            nodes.peek().add(LongNode(line, col, llong))
         }
      }else{
         str = sc.next()
         match(str){
            ")" => {

               if(nodes.isEmpty()){
                  errors.add(new ErrorOrWarning(line, col, "unexpected token:
                     {str}"))
               }else{
                  rootNode = nodes.pop()
                  if(rootNode <> null and not nodes.isEmpty()){
                     nodes.peek().add(rootNode)
                  }
               }
            }
            else => match(str){
               "( +" => nodes.push(MathNode(line, col, MathOps.PLUS))
               "( -" => nodes.push(MathNode(line, col, MathOps.MINUS))
               "( *" => nodes.push(MathNode(line, col, MathOps.MUL))
               "( /" => nodes.push(MathNode(line, col, MathOps.DIV))
               "( **" => nodes.push(MathNode(line, col, MathOps.POW))
               else =>{
                  if(str.startsWith("(")){
                     str = str.substring(1, str.length());
                     nodes.push(MethodCallNode(line, col, str))
```

```
                }else{
                   if(nodes.isEmpty()){
                      errors.add(new ErrorOrWarning(line, col, "unexpected
                          token: {str}"))
                   }else{
                      nodes.peek().add(NamedNode(line, col, str))
                   }
                }
             }
          }
       }
    }
 }

 rootNode?:LongNode(0, 0, 0), new Result(errors, warnings)
}

////////////////Visitors////////////////

trait Visitor{
   def visit(longNode LongNode) Object
   def visit(mathNode MathNode) Object
   def visit(namedNode NamedNode) Object
   def visit(methodNode MethodCallNode) Object
}

class CodeGennerator ~ Visitor{
   def visit(longNode LongNode) Object{
      "{longNode.along}"
   }

   def visit(mathNode MathNode) Object{
      String.join("{mathNode.what}", ("" + x.accept(this)) for x in
          mathNode.nodes)
   }

   def visit(methodCall MethodCallNode) Object{
      "{methodCall.methodName}(" + String.join(", ", ("" + x.accept(this)) for x
          in methodCall.nodes ) + ")"
   }

   def visit(namedNode NamedNode) Object{
      "{namedNode.name}"
   }
}
cg = CodeGennerator()

private numericalTypes = new set<String>()
{
   numericalTypes.add("I")
   numericalTypes.add("J")
   numericalTypes.add("Ljava/lang/Long;")
   numericalTypes.add("Ljava/lang/Integer;")
}

class NodeChecker(ctx Context) ~ Visitor{
```

```
errors = ArrayList<ErrorOrWarning>()
warnings = ArrayList<ErrorOrWarning>()

def visit(longNode LongNode) Object => "I"
def visit(mathNode MathNode) Object{
   for( x in mathNode.nodes){
      x.accept(this)
   }

   "I"
}

private def raiseError(line int, col int, str String){
   errors.add(new ErrorOrWarning(line, col, str))
}

def visit(namedNode NamedNode) Object{
   vari Variable? = ctx.getVariable(namedNode.name)
   type = vari?.type

   if(type){
      if(type not in numericalTypes){
         raiseError(namedNode.line, namedNode.col, "Variable: {namedNode.name}
             is expected to be of numerical type")
      }
   }else{
      raiseError(namedNode.line, namedNode.col, "Unknown variable:
          {namedNode.name}")
   }

   return type
}


def visit(methodCall MethodCallNode) Object{
   meths List<Method> = ctx.getMethods(methodCall.methodName)

   found = false

   argsWanted = methodCall.nodes
   wantedSize = argsWanted.size()

   for(method in meths){
      ret = method.returnType
      if(ret=="V" or ret in numericalTypes){
         margs ArrayList<String> = method.arguments
         if(wantedSize == margs.size()){
            found=true
         }else{
            //check to see if an arg is an array - then can attempt to vararg
                in
            found = margs.stream().anyMatch(a => a <> null and
                a.startsWith("["))
         }

         if(found){
               break
            }
         }
      }

      if(not found){
         raiseError(methodCall.line, methodCall.col, "Cannot find method:
             {methodCall.methodName}")
      }

      for(arg in argsWanted; idx){
         atype = ""+arg.accept(this)
         if(atype not in numericalTypes){
            raiseError(methodCall.line, methodCall.col, "method argument {idx+1}
                is expected to be of numerical type not: {atype}")
         }
      }

      return "I"
   }
}

///////////////// Lang /////////////////

class SimpleLisp ~ LanguageExtension{
   line int
   col int
   rootNode ASTNode?=null

   def initialize(line int, col int, location SourceLocation, source String)
       Result {
      this.line = line
      this.col = col

      //build abstract syntax tree...
      (this.rootNode, result) = parse(line, col, source)

      result
   }

   def iterate(ctx Context) IterationResult {
      rootn = rootNode
      if(rootn){
         nc = NodeChecker(ctx)
         rootn.accept(nc)

         code = ""
         if(nc.errors.isEmpty() and nc.warnings.isEmpty()){
            code = ""+rootn.accept(cg)
         }

         new IterationResult(nc.errors, nc.warnings, code)

      }else{
         errors = ArrayList<ErrorOrWarning>()
         warnings = ArrayList<ErrorOrWarning>()
```

```
            new IterationResult(errors, warnings, "")
        }
    }
}
```

We can now use or previously defined language extension:

```
from com.mycompany.myproduct.langExts.miniLisp using SimpleLisp

result = SimpleLisp||(+ 1 2  3 3 ) )||//result == 9
```

As we can see, this is a lot of work, but the results are very impressive.

Life is just a lesson
Time is a perception
Love will kill depression
This is my confession

You are my obsession

- You Are My Obsession by Trevor Something (2020).

45. Other

45.1 Assert

The assert keyword is a nifty one liner which enables us to check that a condition is true, and throw an exception if it's not. It is commonly used to sanity check input variables to functions. Example:

```
def afunction(a int){
   assert a > 1
}
```

If the above assertion expression `a > 1` resolves to false then an exception of type `java.lang.AssertionError` will be thrown with an error message including the offending line of code.

We can provide a custom text string which will be used to populate the `AssertionError` exception thrown in case of the assertion expression `a > 1` resolving to false as follows:

```
def afunction(a int){
   assert a > 1 "a cannot be larger than 1"
}
```

Now if the assertion expression `a > 1` resolves to false then an `java.lang.AssertionError` exception will be thrown with message: "a cannot be larger than 1".

Unlike languages such as Java, assertions are switched on in Concurnas by default and cannot be disabled.

45.2 Shutdown handlers

The `com.concurnas.lang.concurrent` class includes a function: `addShutdownHook(onShutdown ()void)` which consumes a single method reference. If this function is called then the passed method reference will be invoked upon shutdown of the JVM process upon which the Concurnas program is running.

Shutdown may occur for instance, in linux via a `SIGTERM` being sent to the process, or via the program terminating *naturally* via a call to `System.exit` etc.

More than one shutdown hook may be registered though the call order is non deterministic. More information regarding shutdown hooks at the JVM level may be found here.

The `com.concurnas.lang.concurrent` class is auto imported thus we are able to write code such as the following:

```
concurrent.addShutdownHook(def() void { System.out.println("System shutdown
   initiated") })
```

Note that the code associated with the method reference passed to `addShutdownHook` may not create new isolates.

45.3 Auto imported classes

Concurnas automatically imports a number of commonly used classes. This saves us the effort of having to explicitly import the following. The import name and full paths auto imported are as follows:

- `*` - `java.lang.*`
- `*` - `com.concurnas.lang.*` Including:
  - `*` - `com.concurnas.lang.ranges`
  - `*` - `com.concurnas.lang.tuples`
  - `*` - `com.concurnas.lang.datautils`
  - `*` - `com.concurnas.lang.nullable`
  - `*` - `com.concurnas.lang.concurrent`
- `Remote` - `com.concurnas.lang.dist.Remote`

45.4 Custom classloaders

Sometimes it can be useful to define custom classloaders. This is supported in Concurnas with some minor caveats:

- Custom classloaders defined for use with a Concurnas program must be subclasses of `com.concurnas.ConcurnasClassLoader` - this classloader performs runtime class augmentation to permit the functionality of isolates and actors within the Concurnas runtime.
- Custom classloaders must define a method of signature: `getBytecode(name String)byte[]` which should return the bytecode as a byte array of the requested class.
- Custom classloaders may not spawn isolates, but they can call methods on actors.

Here is an example of a custom classloader with a publicly accessible code store:

```
from com.concurnas.runtime import ConcurnasClassLoader

class MyClassLoader(parent ConcurnasClassLoader) < ConcurnasClassLoader{
   public classStore = new map<String, byte[]>()

   override getBytecode(name String) byte[]{
      got = parent.getBytecode(name)
      if(got == null) {
         got = classStore[name]
      }
      got
   }
```

```
  override loadClass(name String) Class<?>{
    ret Class<?> = parent.loadClass(name)
    if(ret == null) {
      if(name in classStore) {
        return super.defineClass(name, classStore[name])
      }
    }

    return ret;
  }
}
```

45.5 Security Managers

Concurnas has support for security managers. These enable you to run, in a controlled environment, code which you semi-trust or at least wish to exert some restrictions over.

One security manager is defined per Concurnas program instance[1]. This is accessible via the following singleton call: `com.concurnas.bootstrap.lang.ConcurnasSecurityManager.getInstance()`. This applies across the entire virtual machine, all isolates, actors etc.

Security policies are definable on a per classloader basis. One must explicitly reference the classloader to which a security policy applies. This binding is achieved by calling the `registerClassloader` method of `ConcurnasSecurityManager` as follows:

```
clsLoader java.lang.ClassLoader = //our classloader
permissions java.security.PermissionCollection = //our security policy here

ConcurnasSecurityManager.getInstance().registerClassloader(clsLoader,
    permissions)
```

With binding complete as above, every class loaded by the `clsLoader` will be subject to the security policy which we have defined and registered for it. Two very simple example policies are defined below:

```
from java.security import PermissionCollection, Permissions, AllPermission;

allPerms PermissionCollection = new Permissions()
allPerms.add(new AllPermission())//code can do anything! This is the default case

restrictPerms PermissionCollection = new Permissions()//no permissions added
```

Real security policies are of course much more complex. More information on them can be found here: Java Security

It's worth remembering that although the security manager approach to managing code seen here is very effective, it's not bulletproof in the sense that even with a fully "locked down" classloader can run code which either accidentally or maliciously can cause a denial of service by say running a set of infinite loops or consuming a huge amount of ram. For this reason we'd never recommend running untrusted code in a Concurnas shared JVM environment along with other mission critical code. The recommended approach here would be a sandbox provided by the operating system upon which one's JVM operates, or even a dedicated virtual machine (this is the solution which many cloud providers use).

[1]This is a JVM restriction

45.6 Escaping Keywords

Keywords in Concurnas can be escaped, and therefore used as method names, variable names etc by simply prefixing them with a backslash, for example:

```
\for = "a string"
```

This feature is useful in instances where one is using library code written in languages other than Concurnas which has assets (classes, variables, fields etc) with names that happen to be keywords in Concurnas.

45.7 Native code caveats

Although Concurnas itself does not have direct support for native code, if calling Java code (or other JVM languages supporting native code), then native code may be callable. One caveat to bear in mind with native code concerns the mechanism by which Concurnas isolates operate.

Behind the scenes, in order to facilitate the pausing of code at arbitrary points in program execution (e.g. when accessing a ref which has not yet a value set) Concurnas makes use of a 'fiber' Object attached to every method invocation. This causes a problem for native code since it does not capture this fiber upon execution. This means that it is not possible to call idiomatic concurrent Concurnas code (e.g. which uses actors and refs) as a callback from within native code.

We must be cognizant of this limitation concerning callbacks when working with native code. In practice however this turns out something which is often easily worked around/not a critical issue.

45.8 CObject

In Concurnas, all objects are subclasses of class `java.lang.Object` and class `com.concurnas.lang.CObject`. The `CObject` class defines some useful methods such as `toBoolean` which can be optionally overridden in order to support instance objects of a class being usable in branching tests:

```
class MyHolder<X>(~held X?){
   override toBoolean() => held &<> null
}

//used as:

inst = new MyHolder<int>(654)

if(inst){
   System.out.println("A value has been set")
}
```