Example Program

Import [home.CASE.main](http://home.CASE.main); //main imperative classes. Low-le3vel

Import [home.CASE.System](http://home.CASE.System); //Handles internal routines for the program

Import [home.CASE.StringRoot](http://home.CASE.StringRoot); //Necessary routines to construcet a program

Import [home.CASE.API](http://home.CASE.API); //Typical routines

Import [home.CASE.STL](http://home.CASE.STL); //Extended Library

Import [home.CASE.Math](http://home.CASE.Math) //Math specific libraries

Category: VirusPackag

Category:3Dgraphics

Category:UnityExtensions

Category:VideoGames

Category:ScienceFunctions

Category:PhysicsRoutines

Category:OpenGL

Category:NLP

**Package** User.DemoProgram;

**Namespace** RandomObjects

**Requires;** or a **satisfies**

**//Requires**

//This keyword in a source file means that anyone who wants to

//Use the source code via an import must specifically declare it using the keyword //**satisfies** in the using source code.

**Object->String->Fruits->**

#Public class Fruits

**EntityPool** pool; //main memory poool and in charge of typeclasses

//streams are allocated here. Youcan use this like

//a global variables but only in the sense that

//they are available all the time. Retain uses this

//so you can release anywhere

//Streams are by default implemented with mutex

**stream** n = new Fruits. //also, you can instantiate a class within that class like java

**stream** m = new Vegetables;

//change threads manually

**branch(“branchname”)** where branchname is th name of the stream you implemented.

CLASS CONSTRUCTOR

//all variables are allocated on a stream outside of the pool

public myCoffeeShop **( n)** nameLess;

**prototype class NewObject extends OldObject**

//new object is initailized by extending OldObject (not class, object so there is state information

//Each class is an instantce of a metaclass, created at runtime. Like Objective //C. We // can define class methods, pass them to functions, pass them into //collections, and ///so on. To create an instance of a class we pass a// //message to the class in the //entitypool with the name of the class and it //returns a class.

public void biscotti = new pool.entityPool.;

**volatile** public printCoffee //prints an object’s String built in //function. Also volatile means

print(nameLess) //can be changed by a background //process

public getFruitCount(long int n) **//primitives are objects**

**tag ( IOStreamException | FileInputStreamException | ArrayIndexStream)**

String myString;

myString = someExceptionThrowingFunction;

**retain** int **(m)** a = float.parseInt(n) //objects are allocated on the //stream

**retain** float **(m)** b = int.parse(float)

string **(m)** c = “Hello World!”

return a;

public n.get(“nameLess”);

print(nameLess); //outputs value of stream variable nameLess

public <PhysicsLibrary> pLibrary = new <>(); //defines and instantiating

endclass Fruits

String->Object->Radar

#public **sealed** class <Radar> myControlPanel = new <>(); //new allocates memory //cannot be subclassed because its sealed

[public construct Radar(int numnberOfComputers)]

[public construct OverLoadedRadar(int numComputers, ]

. [String brandName]

[public **dynamic** void blip()

**{ tag ( IOStreamException) }**

**{ tag ( FileInputStreamException) }**

**{ tag(ArrayIndexStream) }**

{ String myString }

{ myString = someExceptionThrowingFunction }

{public short int **frozen** addBlip}

//NumberOfApples can only hold ints now, where

{(**frozen**(int numberOfApples))}

{(**thaw**(int numberOFApples)} //Turns off static typing

]

[~ControlPanel  
  
 { autorelease <PhysicsLibrary> pLibrary}

{ autorelease <Radar> myControlPanel}

{ autorelease <String> c}

]

endclass Radar

**Object->String->SpaceShip**

Public class Spaceship

{ EntityPool pool, public Radar, boolean statevariable}

{ tuple Coordinates = (10, 10, 10) }

{extern String Walkway } //this variable ties with another in another file

[public messagePassing()

//get a message from a function

{int r = [circleRadius():<CircleType>};

{[r:cyclinderMaker]} //sends r value to the function

]

**[sealed** public void StreamOp //cant be //overridden or overloaded

{ release EntityPool.getStream(objectname) }

{ String list = (String)ArrayList<String> }

//put runtime class construction here

]

[public void ShowKeywords()

{assert(b == 4)}

//@:Alice is an object, Input “Alice” is a String

Try Catch (IOException io) { try Print “@:Alice”; Input Alice”} .

{StackPrint, Processor = new Processor} .

{await Processor.processorAsync}

{try catch { Print “@:Alice”; Input “Alice” } throws (IOException);

]

[public void ShortCutExceptions()  
  
 Old method:

Try {  
 FileInputStream fis = new FileInputStream();

fis.doSomething();

} Catch (Exception eo) {

PrintStream;

} Finally {

Error():

}

]

[public void ShortCutExceptions()  
  
 //New Method:

//With AutoFall you know at one point the current thread will //terminate it //because it finishes a data stream from the file source //or keybaord input, //etc. So we will let the compiler and its magic //handle try….catch, etc. As such,

{ **Autofall** FileInputStream fis = new FileInputStream() }

{ Fis.doSomething. }

//It you can even chain them toegether:

**//AutoFall** FileInputStream fis;

//Fis.doSomething();

**//AutoFall** IOStream ios;

//Ios.doSomething():

//In the above, it executes both in parallel and both only alert you if you //there is an exepction. They are much more compact, syntactically, as well.

]

[public friend void Depth(Float d)

{Print “Depth is d”}

]

[Inline \_asmcode

//asseemble code

]

[Bytecode \_byteLowLevel =

//bytecode

]

[lambda

{x \* t, 5\*x, return lambda()}

{String command = **commandExecute**(“pwn || ls > file.txt”)}

]

[public void moreCommands()

{Int i++}

{Wait(classObj.getData() == true)}

//Remember CASE is by nature a multithreaded language

//example

Set st = { 1,2,3,4,5,6} >> yields {1,2,3,4,5,6}}

Set ot = { 1, 2, 3, 4 , 4, 4} >> yields {1,2,3,4}}

]

[public moreStuff()

{Int k = 5.0f}

//This allows a person to assign a value dynamnically by assigning a value, such as a //float, to another type. The type declared remains the type of the variable, such as //int, despite the fact it is holding a float. Categories?

]

endclass SpaceShip

String->Object->Plane->BattleCruiser->AirCraftCarrier

Static public class MySingleton

//this class can only have one instance of it (singleton)

endlclass MySingleton

String->Object->Ship->Submarine->AircraftCarrier

#void public class

{final int subdepth = 10, final int supdepth = Ten}

{final char flag = “a”, boolean subsunk = false}

//Categories let us define new methods and add them to classes for which we don’t //have the source code (such as the standard Cocoa classes provided by Apple). This //makes it easy to extend classes without resorting to subclassing. Extremely useful //to adapt existing classes to the requirements of frameworks we want to use or //create.

//category

//inner classes

String->Object->Inner->JetTakeOff

#class JetTakeOff

float speedTakeOff;

///anonymous class

String->Object->Inner->MyInner

#myFunction(new MyInner { #Class {

Print speedTakeOff})

String->Object->Inner->Anonymous

myNewFunction(new lambda() { #Class {

{Print speedTakeOff})

If (readyAircraft Is readyTank)

{

{Print “Shows if the two operands are of the same reference”}

}

If (readyAircfraft isNot readyTank)

{ Print “Negates the check Two” (n)}

{ “operands arent of the same reference”}

if (readyAircraft typeOf WorkingAirCraft Class)

{ Print “Checks to see if the operands “ (n)}

{ “are of the same type or family tree.”}

}

if (ShipPanel instanceof ShipHull)

{

{Print”Checks to see if the object is of a certain class”}

}

if (ShipPanel getType ShipHill)

{Return to the typeclass info of the object}

//specific access modifier - shall

**shall AirCraftCarrier to Fruit**

{<T> <ArrayList> myFirstArrayArrayList= new <ArrayList><ArrayList>()}

//the T is a template

{<Integer><ArrayList> myFirstArrayInteger = new<ArrrayList><ArrayList>()}

{<Integer><ArrayList> mySecond ArrayInteger = new <ArrrayList><ArrayList>()}

public void ApplesAccess()

{

**Friend Keyword**

Allows your “friend” class to have a function within your class that it can

Call despite any access modifiers.

[public void openShip()

{double myDouble}

{float myFloat}

{fixed myFixed}

//wildcard

if (myFunction.”\_ 10 \_ “)

{

{Print “myFunction matches “}

}

If (myList ??) // - passes true if myList is initialized

{Subsunk = true}

{“Islands List.” ( c ) “Hawaii” ( c ) “Puerto Rico” ( c ) “Cuba”}

{ ( m) Islands List ( r ) “Cuba” }

//finished is “Islands List Puerto Rico

]

[public void unsignedvalues()

{

{**enum** unsignedNums{ Uint unsignedInt Ushort unsignedShort} .

{Ulong unsignedLong }

\_**register**

[

//loads values into cpu register

]

]

[label:output]

[public void output(String p)

{PrintHashMap(String p)}

]

[public void output2(String p)

{**goto** [label:output]}

]

[public void output3(String p)

{**jump = this}**

//if the home function is in an uninstantiated class

//this takes on the role of an inner clas

//this takes the class and gives it object propreties

//putting it into entitypool temporarily

]

[public void listComprehension(String list)

{

{sentence = "the quick brown fox jumps over the lazy dog"}

{words = sentence.split()}

{word\_lengths = [len(word) for word in words if word != "the"]}

]

endclass AircraftCarrier

Object->String->SpaceShip->MainNavigation

#Public class MainNavigation extends SpaceShip

{

//Constructors purpose is to pass parameters that the class needs in order to //intitialize. Constructors can return values in CASE..

[SpaceShip() //class constructor

{TypeName ClassName myClass} 🡨 declaration evaluates to the {empty array[] myClass = new Bob} <- instantiates the class.

]

SpaceShipTemprature

Public void SpaceShipTemperature

//The **explicit** keyword declares a user-defined type conversion //operator that must be invoked with a cast. For example, this operator //converts from a class called Fahrenheit to a class called Celsius:

//public static explicit operator Celsius(Fahrenheit fahr)

//{

// return new Celsius((5.0f / 9.0f) \* (fahr.degrees - 32));

//}

**[**public void veryErrorProne()

**//**automated debugging statements

**{Printbug**(myClass == -1, “Wrong data types”)}

{**Printbug**(pool.Radar == “Status”, “Pool contains Radar ”)}

////exceptions are done before the inflicitng statement

//output

Try Catch(ConsoleIOStream io)

{

{public async void Process()}

{Processor processor = new Processor()}

{**await** processor.ProccessAsync()}

}

catch (Exception exception)

{

{ExceptionLogger logger = new ExceptionLogger()}

// Catch operation also can be aync now!!

{**await** logger.HandleExceptionAsync(exception)}

}

**{Print** lastValue= **Last;} //evaluates to the last expression within**

**//the try block.**

` ]

endclass MainNavigation

//The await keyword can be called inside the catch and finally blocks. This //opens up the way to perform an async exception handling or fallback //process in case an exception happened during an async process call.

//An improvement on try….catch statements. Classes and or operations that //are normally required to have to declare a try statement blocks like //IOStream exceptions and FileStream etc. all must, if they use this new// //method, implement autofall. For example

//old method

//With AutoFall you know at one point the current thread will terminate it //because it finishes a data stream from the file source or keybaord input, //etc. So we will let the compiler and its magic handle try….catch, etc. As //such,

Object->String->SpaceShip->Engine

public class Engine

[public void engineFunction()

**{AutoFall** FileInputStream fis}

{Fis.doSomething()}

It you can even chain them toegether:

**{AutoFall** FileInputStream fis}

{Fis.doSomething()}

**{AutoFall** IOStream ios}

{Ios.doSomething()}  
]

In the above, it executes both in parallel and both only alert you if you there is an exepction.

{**tag** String myString}

[Public void engineFunction2()

{myString = someExceptionThrowingFunction()}

]

///now, here, myString’s contents from the engine function

///it still executable

endlclass Engine

**//Lock**

//The **lock** keyword marks a statement block as a critical section by obtaining the //mutual-exclusion lock for a given object, executing a statement, and the releasing //the lock. The following example includes a **lock** statement.

String->Object->Account

#class Account **synchronize(“a”)**

{decimal balance}

{private Object thisLock = new Object()}

{**mutex** thisLock}

[public void **synchronize(“thisLock”)(** Withdraw(decimal amount)

//synchronize mutlithreading’

{**Synchronize}**

{ lock (thisLock)}

TryCatch (ThrownCustomError) {amount>balance}

{throw new Exception (“Insuffience funds”}

{balance -= amount}

}

}

]

3 levels of synchronization.

[public void pancakes(Stack a, Stack b)  
 {String newObject = pancakeflavors}

{String newString = this} ///assign a string type to this

{this->super()} //calls constructor for parent class

]

public struct typedef myStruct

[  
 int n;

int o;

int j;

int messager;

string lazystring;

]

endclass Account

public union

String->Object->Extendable-><>

#public class Extendable extends <Object>

<Extendable> extended = new <Object>();

[public void poly()

{Object obj = extended}

Do { int i++; i<10; String “Hello”; docs.read() } (Until i<20)

]

[public void size(int object)

For (int i=0;i<sizse;i++)  
[ {myAddress.changeAddress()}

{myROcket.getLaunchDate()}

{**residue** myRocketheight};

]

Print sizeof(extended); //see above

]

[public void Iterations

{if (String j == “Jacket”)}

{for(String i : Jacket)}

{if (String i == “Leather”)}

{continue;}

{elseif (String i== “Vinyl”))

{else(String != “Jean”)}

do

{

{int **ctr++**}

{Print “Hello Jupiter”}

{Print “Hello Aliens”}

} while(ctr <30)

while (value == token)

{

{value++}

} where ((f:value))

]

//each iteration runs the while loop, where f (passed the element vlaue)

//it is a way to call a function f with the operands value that is generated in the loop

]

endclass lambda

native

mathematical operators

String->Object->Devices->Vehicles->Tank

#public class Tank()

//this is a primary constructor

class Tank(string item, int size) //class constructor

//need access to a portion of code via functions

//the list below represents that you can

//now have access to Tank(), Vehicles, etc…

//up the vine, all the way to the root String class.

**Vehicles : Tank (->)**

public string Item { get; } = item;

public int Price { get; } = price;

[public Thank()

//**Resolve**

//When using a for each loop often it is handy to see the //iterative side, that is, the index side of the for each loop, instead of //the blind iterator.

For (String s : myStrings) //a foreach loop

{  
 {currentCtr = **resolve} //**shows the iteration of the current loop

{Print “At element “ ( c ) currentCtr}

**}**

{System.out.println(“Current counter is “ (c) **resolve** (c) “)}

]

**Leap**

//Typically when you want to modify a control structure, say, you want to alter or //remove an item jelly(i) in the loop for (int i=0;i<jelly.size();i++) . Typically you //can’t becauase altering jelly(i) will change the state of the loop. For example, if by //altering jelly(i) you might skiup over a whole block of jelly since you removed an //item, and then the next time through you removed another item. Anyway, the //solution to this is the **leap** keyword. Returning to our jelly example, say we //identified a can of jelly numbered 5, and we want to remove it from our list of jelly //jars. Simply do this:

For (int i=0;i<jelly.size();i++)

{Print “Hello World”, Print “I don’t like strawberry jam”}  
 {**leap** jelly.remove(Strawberry)}

}

{jelly.remove(remove)}

This is equivalent to C:

Int remove = 0;

For (int i=0; i<jelly.size();i++)  
{  
 {cout << “Hello World” << endl}

{cout << “I don’t like strawberry jam}

if (I == Strawberry)

{

{remove = I}

}

}

//public foreach based loops.

[Public void landfill(String location, double tons)

//calculate total tons

//String is default for classes

for (String st: tons)

{

{Print “ More Tons” ( c ) st.tons[i]}

}

for (int i=0;i<tons.size();i++)

{

{Print “More Tons – amt: “ ( c ) tons[i]}

}

**List Comprehensions and Generator Expresions:**

[(i,j) for i in range(3) for j in range(i) ]

((i,j) for i in range(4) for j in range(i) )

[Public void Temperature today (Temp t)

//since its our root type, we use it to pass back the result

{String returnString};

Switch(t)

{

case (t == 5.0)

{returnString = 50}

**exit**; //passes control out of the current scope

case (t == 0)

{returnString = “Wow its cold”}

**pass;** //passes control compeltely to the root scope

case (t==”Crackers”)

{returnString = “I’m hungry”}

**break**; //escapes from the switch statement

}

pool.get(“landfill”)

{

{String landfillmountain = new landfillmountain}

{Pool.set(“marsh < 10”)}

}

]

endclass TanString->Object->Abstract->House

#public abstract House

public abstract void Kithen = { return kitchen};

public void Living Room = {livingRoom = Living Room”;

public abstract void BedRooms;

public abstract void Bathroom; //abstract function because declared without an implementation,

endclass

//The abstract keyword isn’t necessary. One abstract function and the whole class is one.//

[Public myFavoriteRooms extends abstract House

public void BedRooms; {return 4}

public void BathRoom; { return dog}

]

[public void TimePeriod

private double seconds;

]

[public double Hours

get { return seconds / 3600; }

set { seconds = value \* 3600; }

]

[public void computers()

//When it isn’t necessary to write a function for a little bit of code:

*identifier* => { body of code }

`

//Identifier => { body of code }

//Identifier is the name of a class to attach to. Body of code is //obviously code. //Numeral is a number to give it a unique identifier so you //can create more than //once.

//You can then retrieve the code from the entity pool using the *identifier* //name. Or you can just call it using the fully qualified name including its numeral. And a ‘snippet’

//example

while (true)

{

{pool => simple\_print\_hello => { Input hello, Print hello}}

}

public void printSimpleHello(true)

[

{function a = pool.get(simple\_print\_hello)}

//or simple enough

{pool.run(simple\_print\_hello)}

]

]

String->Object>Telescope->Results

[public Telescope extends

//since String is root class

{Int b = “This is an integer”}

//Iterators are two-part. First you get the iterator, then you get the variable.

{Iterator iter = String;} //String is iterable as tokens, (it doesn’t make sense iterating over String[]

For (Iterator iter : MyIterableCLass)

{Iter.printString(\*iter)}

//The iter star dereferences the iterator back to the host class.

]

endclass Results

String->Object->Internal->Class

**#internal** class

//inclusion

String output = **null**;

**for** (XMLPatternTag tags : source.XMLPatternType)

{

{output = output + tags.name;}

}

Replace with:

**for** (XMLPatternTag tags : source.XMLPatternType)

{

{inclusion output = output + tags.name}

{output = output + tags.name}

}

//In this case we want to iniitalize output to some value that’s not **null**, but is //the first element of XMLPatternType. Otherwise, the string is initiailized to //null and then the rest of the loop content is carried over. This is a common //problem . The **inclusion** keyword is meant to solve this. It is an essentially //an initializer that executes code that only executes the first iteration of the //loop.

//The **inclusion** code is the only code that executes the very first iteration //and it only executes once.

//Able to rename one class to something else dynamically

**rename** BobsClass => TedsClass

endclass lambda

String->Object->ObjectTest

#class ObjectTest

{public int i = 10}

endclass ObjectTest

String->Object->MainClass2

#class MainClass2

[static void Main()

{object a, a=1, Console.WriteLine(a), Console.WriteLine(a.GetType())}]

{Console.WriteLine(a.ToString())}

{a = new ObjectTest(), ObjectTest classRef}

{classRef = (ObjectTest(), Console.WriteLine(classRef.i)}

endclass MainClass2

/\* Output

1

System.Int32

1

\* 10

\*/

//With the Auto-Property initialization feature, the developer can initialize //properties //without using a private set or the need for a local variable. Following //is the sample //source code.

Object->String->Insects->FlyingInsects

#public class insects

public List<string> Roles { get; } = new List<string>() { "Bee", "Queen Bee"};}

endclass FlyingInsects

**Iterating Property Arrays**

In multiple values for one get it is legal to say:

String->Object->TextConverter

#public class <TextConverter>

public void get

[ get new ASCIIConverter ]

[ get new TeXText ]

[ get new TextWidgetController ]

Where the get values are matched by type. In this case the class TextConverter’s get function is called.

TeXText = myTextConverter.get;

**//Easily Nest Types within Types**

//Type names are explicitly defined upon the delaration of the variable or class. To //ensure concreteness of the types in the class and of the path of inheritance, types //must be declared as such:

ObjectClass->String->myClass

#Class myClass

myClass->int->int->int //myClass, return, x, y

public int addition(int x, int y)

{

//Let assigns a variable

{Let <?> int = pool.get(“space\_probe\_function”)}

{Class<?> cls = pool.get(“space\_prob\_function class”)}

{Object<?> obj = pool.get(“space\_prob\_instance”)}

{return x+y}

}

myClass->int->int->int

[public int subtration(int x, int y)

{return x-y}

]

myClass->String->int->int

public String division(int x, int y)

[

{return x/y}

]

//iterate

//new generics

//below creates a map of Strings of Lists of Trade

//and instantiates it as TreeMap

{Map<String><List><Trade> trades = new <TreeMap><>()}

[Public void MultipleStatements()

//Use the double bar operator

{int n = 10 || msgBo(“N is “ ( c ) I)}

{int n= 10 || k = 20 || n }

]

[Public void Ellipses()

{array[5] myArray = {1, 2, 3, 4, 5}}

>>myArray[1..3]

[1, 2, 3]

{array[5] myArray2 = { a, b, c, d, 1, 3, 2 }}

>>myArray2[1...5]

[a,b,c,d,1]

]

[public void multidimensionalArrays()

Array: {int [] myArray = new []}; //Always use the data type and //brackets

Matrix: {int [][] myArray = new [][]};

//Multi-dimensional arrays allocate their own memory by need.

Oblong Array: {int [5][] myArray = new int[5][]}

Two ways to initialize this.

First: for (int myCounter:myArray)

{ myCounter[] = new int[] }

Second: {myArray = new int[5][5]}

Accessing Oblong Array: myArray[x][y]

Accessing 2-dimensional Array: myArray[1][3];

Accessing 3—dimensional Array: myArray[0][1][8]

]

endclass MyClass

}

//INDEXERS

String->Object->SampleCollection<T>

#class SampleCollection<T>

// Declare an array to store the data elements.

private T[] arr = new T[100];

// Define the indexer, which will allow client code

// to use [] notation on the class instance itself.

// (See line 2 of code in Main below.)

public T this[int i]

[get

// This indexer is very simple, and just returns or sets

// the corresponding element from the internal array.

return arr[i];

]

[set

arr[i] = value;

]

endclass SampleCollection<T>

String->Class-><T>

#Public Class<T> Matrix

{Stream <T>[][] myStream = new <T>[][]}

{<T> (myStream) myMatrixVar}

{String label =@: “This is a matrix”}

//

//For purposes of clarity and compiler optimization, you can mark in the functions //definitiion line what calls may and do call it the method. For example…

[Public int HelloWorld(String input) : caller(main.class, interface.class, db.class)

]

.//multiple returns

[public int HelloWorld(String input, input2) : ret (int output1, int output2)

return (1,2);

]

public procedure swatFlies()

[  
 {int i++}

{Print i}

]

[public String function swatFliefs(int z)

{int++};

{(z + 1)} ///infix operations

{Print I}

{(f 1 n) } ///function taking 1 and n as operators

{return I};

]

end class <T> Matrix

String->Object->Volleyball

#Public class VolleyBall

//declaring a function within another function

[playVolleyball

{**SynOperator** Serve [ Hit Volleyball; Bump VolleyballBack ] }

for (Integer int: VolleyCounter;i++)  
 { {Serve[];} }

]

[secondHandFunction

{**SynOperator** Serve[ Hit Vollyeball, Bump VolleyballBack]}

for (Integer int: HitVolleyball--;)

{ {Serve} }

]

[thirdFunction

//overriding the function without changing its signature

//**SynOperator** examines the signature of the entire function

//To see if it can override, it doesn’t just check the header.

{ Serve.HitVolleyball : Bump VolleyballlBack }

]

[public void Method(int age, String firstname="John", double salary=4000.99)

]

**//Assigning Functions**

//In CASE, you can establish a function and then later on assign that function to a //variable. In this case, the variable is the same as the function and can be moved //around to be accesssible from any place the parameter (funcrtion) is passed.

[Public void Assignment(int a, String b)

{ function a = String b}

.

.

.

{ a.execute()} //function was stored from String b to functon a

//function composition

//

// //The colon operator : can be used to compose one function into another.

WrappingPaper:Box:ChristmasPresent(int color, int size\_x, int size\_y, int size\_z)

]

**//Function Concatenation**

[public void assignment2()

**//** Similar to composition but instead of passing one funtion into another, the same //parameters are passed into each function, then their results are concatenated. myFunction@YourFunction@MomsFunctino(1,2,3);

**]**

[public void typeconversions(String horse; Object stalls, String cows)  
  
 {Barn mybarn = **implicit** stalls}

{**fromString** numberOfCattle = pool.fromString(“numberOfCattle”).size()}

]

endclass Volleyball

String->Object->DecimalLikeClasss

**#protected** class DecimalLikeClass

{**convert: Integer}**

**{stream (n) = new Pancakes}**

**{Pancakes (n) myStackOfPancakes}**

**{Pancakes (n) BobsPointer = &BobsObject}**

public void PointersAndBob(“const Integer convert, )

[

**//dereference**

**(\*myStackOfPancakes).pointer.Next == myStackOfPancakes->Next**

{BobsPointer = BobsPointer + (char)} (Bumps up BobsPointer by a char) (smallest)

{BobsPointer = BobsPointer + (int) } (Bumps up BobsPointer by an int) (second)

//Pointer Arithmetic

{BobsPointer = BobsPointer + (char) } (Bumps up BobsPointer by a char) (smallest)

{BobsPointer = BobsPointer + (int) } (Bumps up BobsPointer by an int) (second)

**Dereferencing Pointers**

{BobsPointer = \*BobsPointer} (yields BobsPointer’s value)

]

functionName(int parameter)

functionName(deep parameter)

public class ParameterPractice(int parameter) //pass by Value

public class ParameterPractice(deep parameter) //pass by Value deep company

public class ParameterPractice(need parameter) // delay and force  
public class ParameterPractice(&parameter) //large object doesn’t make sense to pass all of it. Pass by reference

public class ParameterPractice(parameter) //Pass by reference by value like Java

**Pass by Reference by Value**

Like Java, variables can be passed into a function altered and then upon closing of that function, the variables data is kept and passed back to the calling function.

functionName(parameter)

end class DecimalLikeClass

String->Object->Color->Red

#public class Red

int +operator(int rhs, int lhs) { return rhs+lhs } //override +operator binary

int –operator(int rhs, int lhs) {return rhs+lhs:ret (int output1, int output2) }

public !operator(int operand) { return String=”not”:ret(String output)}

//function overloading

public returnString(int output) { return string }

//create your own operator

Public **operator** (h) (int lhs, int rhs) [ return lhs+rhs; ]

//A more advanced operator using a tuple:

Z = n (h) (q,e)

Publi**c operator** (h) (int lhs, tuple q, tuple e) [ return lhs + tuple q + tuple e]

[Public void equalities()

{SkeletonClass ~ ShallowCopyClass} //shallow copy

{DeepCopyClass = SkeletonClass} //deep copy

{StringA &=StringB} //equivalence of reference

{StringA = StringB} //equivalence of value

{StringA =$ MyFloatClass} //third is as a base class

]

[public void EventRoutines

{Subscriber suscribeQueue;}

{Publisher publishQueue;}

{subscribeQueue(publishingClass.publishingFunction {int n=5}}

{publishQueue.(publisherClass.subscribe} . {publishingClass.publishingFunction) { publishing code }

]

endclass Red

String->Object->Apples

#public class Apples

{const macintosh = red}

{final String fruit = “watermelon strawberries banannas grapes”}

public pick-apples(const vegetables \*green)

//cons and pairs

//Since CASE subclasses everything through Strings (its base class), ///the following features of LISP/Scheme languages are available. //**Cons/Pairs in LISP:** Calling (cons `a `b) in LISP creates a //Pair of two elements. You can individually access each element //using car and cdr respectively.

{convo fruit fruitlist}

>>car fruitlist

watermelon

>>cdr fruitlist

strawberries bannas grapes

>>cons car fruitlist car (cdr ( fruitlist))

**Recursion Automatic**

{Public recursive void moreFruit()}

{Base = { ctr = 0}}

{Tail = { ctr++}}

**Annotations**

**Generator**

Generators have the same syntax as a list, however, they do not

retain the intermediary processing information.

{list myList = [2^x for x in range(5)]}

{len(myList)}

>>5

for element in myList

{print element}

//Generator

{generator myGenerator = (x + x for x in range(3))}

for element in myGenerator

{print element}

Everything is ok except

len(myGenerator) fails because of the reason above.

Endclass Apples

//Print “Hello World”

//Becomes

//<keyword command=”Print”>Hello Word</keyword>

Public class myFirstModule