CASE Common Adapting Stream Entitites

Language Synopsis 4

Neither IList<Cat> nor IList<Animal> is a subtype of the other, because IList<T> is invariant on T. 10

Primitives 10

Keywords 11

Input and Output in CASE 19

Control Structures 21

Classes and Inhertance 26

Arrays 35

Tuples 38

Sets 38

Types 38

Type Templates 40

Functions 41

Integrated Functions 45

Pointers 46

String Operations 47

Parameters 49

Equalities 51

Access Modifiers 52

Event Handling 52

Constants 52

Cons/Pairs 53

Recursion 53

Annotations 54

Comments 54

Generator 54

Module Connection 57

Integrated Design Patterns 87

Code Organization 99

Miscellaneous Designs 100

Standard Collections Library 103

Diagram 104

Boolean Operations 106

Regular Expressions 107

Exotic Operations 107

Example Code 110

Compiler Feature 112

CASE Application Standard Library 115

# Language Synopsis

**Root**

The Class Hierarchy goes as follows

Object->String->Char->List

Essentially all classes inherit from the String class.

That also means that every datatype has a String signature. There exists automatic conversion from Strings to any other type. Specifically lists., arrays, queues, and stacks.

**Naming Convention**

Classes, Interfaces, and Abstract Classes and Types are all first letter capital, the rest camel case. Variables and keywords are first letter lowercase, rest camel-case.

**Primitives vs. Classes**

There is no difference between the two. For example, int.parseInt is completely valid, despite the fact it is a primitive.

**Entity Pool**

The entity pool is the main memory pool. From here, streams are allocated.

Also, the garbage collector takes orders from the Entity Pool. The Entity Pool maintains all of the TypeDefinitions, meaning it also in charge of TypeClasses.

Int MyInt = EntityPool.getStreamMemory()

Int GetInt = EntityPool.get(“MyInt”)

//retrieve a variable value from the stream

//stream name variablename

///when values are passed as parameters or returns from functions

//they implicitly are assigned entity memory

Int ListStatus = n.get(“CurrentLocationInList”

**MicroServices**

The architecture grows as demand increases using microservices. The microservices use a we service style architecture. When one microservice dies or completes, CASE replaces it. These things come and go. When a microservice is finished, it is torn down. The process is fast enough that the user should not even know it is done. Services are small and live and die fast, such as AWS lambda, they are spawned for a service then die. This is automatically handled behind the scenes. @microserv annotation turns thise on.

**Initializing Streams**

Streams are by default implemented with mutex

stream n = new TypeName

**Initializing Variables**

Requires an initialized stream to put it on.

TypeName (n) variableName

**Check Value of Streams**

n.get(variableName)

**Instantiating Classes**

**<**TypeName> ClassName myClass = new <>();

No need to be redundant in instantiating a class.

**Atoms**

Since there are no primitives and classes can be tedious to instantiate, a data modifier called an atom allows for a class or primitive to be created without going through the entity manager. This does cause some limitations. Atoms can only be used within their native function.

**Self-Modifying Code**

Code can be written in CASE that self-modifies – typically a no-no but not in AI algorihtms. Use the @selfmodify annotation marking the function that has self-modying code. @selfmodify [function name() body]

**Dynamic Language**

*dynamic* language, which means that some parts of our code can be extended and modified while our apps are running (i.e. after they’ve already been compiled).

Use the **Dynamic** annotation at the top of the class.

**Runtime**

The language is a hybrid. It partially compiles its code into byte-code, but it is bytecode that is generated on the fly, so that it can be used as a scripting language as well.

‘

**Goal: Clearer error messages on execution**

**Garbage Collection**

**release, retain, and autorelease** commands for memory management.

retain is used before the variables initialization and using the entity system can be released exactly when necessary. Can change from retain to release to autorelease anytime using the entity pool.

@retain

Int (n) CurrentLocationInList

Int TempStatus = n.release(“CurrentLocationInList”)

@autorelease Atoms size = 0 (running autorelease on an atoms variable)

**Files**

Keep Java’s style of having a class and its anonymous classes in one single file.

**Retain/Release**

Similar to new and delete in C++. The difference is, this is completely optional. Using retain and release you can circumvent the automatic memory management by dynamically changes memory reserved. Or erase the memory completely.

**Message Passing**

Part of the important part of the architecutre is the passing of messaging. In this case messages are sent to the Entity Pool, into the Force/Delay logic, or passed between streams from the stream mechanism. To pass a message from one to another, [function():destination] or [variable:destination], the array brackets in this case sends the contents to the entity pool waiting for another routine to pickup itse “message”

**No More Nulls**

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

Suppose System.out.println(myClass)*before* new Bob, myClas will just return a blank line.

If a null-like situation arises. The compiler instead of leaving a vague error, will locate the exact position of the null value and why it is null.

**Debug Statements**

I’m sure everyone is tired of typing out print debugging statements or exits statements to debug everything. CASE will allow you to use the following keyword:

**Printbug**(conditional, message). Unlike assert that exits upon on an error, **printbug** outputs data depending on the condition . Example:

Printbug(grass == green, “Grass is green”);

**Try…Catch…Finally**

Try and Catch is a handy utility but it tends to obfuscate the code. Instead,

Try Catch(IOException io) { try-code} { finally-code} code {\}

**Last**

Evaluates to the value of the last expression.,

**Await in Try Catch**

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.

public async void Process()

{

try

{

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);

}

}

**AutoFall**

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:

Try

{  
 FileInputStream fis = new FileInputStream();

fis.doSomething();

} Catch (Exception eo) {

PrintStream;

} Finally {

Error():

}

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;

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.

try

**Nightmare of Null Scoping**

Often times you try to implement a piece of code and the compiler swears back at you. It says that there is a possibility your variable may not have initialized that class. SO you have to go back and assign a null to the class object. Next thing you know you have nulls flying this way and that. So what to do? You don’t have much choice. You can initialize the class before the try catch but then you run into the possibility of a null. What you should be able to do is not null have to instantiate the object before the try catch, you should be able access the objects value after the try catch block. This is tough, I think an example is due:

String myString = null;

Try {

myString = someExceptionThrowingFunction;

}

Catch(IOException)

{  
StackTrace;

}

print(mySttring);

So instead why don’t we just:

**tag** String myString;

myString = someExceptionThrowingFunction;

Taggign the variable does several things. First it tells the compiler that this variable is going to be tracked by an execption system. So if there is an exception that happens, it will throw an exception. Secondmost, and foremost, it even keeps the scope level. So instead of the mess of nulls, your FileStreamReader for example, will be able to be accessed in any of the new lines.

**MultiCatch**

You can MultiCatch is to do multiple catches in one statement like the following:

(notice no semicolon after tag)

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

String myString;

myString = someExceptionThrowingFunction;

**ChainCatch**

Or you can chain exceptions….so an alternative:

**tag ( IOStreamException)**

**tag ( FileInputStreamException)**

**tag(ArrayIndexStream)**

String myString;

myString = someExceptionThrowingFunction;

The three tags are applied to the myString variable;

**Dynamic Typing vs. Static Typing**

CASE is dynamically typed but using the **frozen** keyword on a datatype it will

act as a statically type.

(frozen(int numberOfApples)). NumberOfApples can only hold ints now, where

int numberOfApples could have a function assisgned to it, a string, etc.

(thaw(int numberOFApples) Turns off static typing.

**Runtime Class Construction**

May not be the most useful thing but any **diagram** can use a pipeline and runtime class construction to so that essentially, other applications can create code or if for example, suppose the distant application needs to interface with your application. The distant application can use this and a diagram to establish new, dynamic, functinoality.

**Smart Error Handling**

A real-time error handling utility is employed so that as the code executes it searches for certain patterns in the execution – beyond typesafety.

**Typecasting**

To convert from one typeclass to another.

String list = (String)ArrayList<String>

**Covariance and Contravariance**

Many [programming language](http://en.wikipedia.org/wiki/Programming_language) [type systems](http://en.wikipedia.org/wiki/Type_system) support [subtyping](http://en.wikipedia.org/wiki/Subtyping). For instance, if Cat is subtype of Animal, then an expression of type Cat can be used whenever an expression of type Animal could. Variance refers to how subtyping between more complex types (list of Cats versus list of Animals, function returning Cat versus function returning Animal, ...) relates to subtyping between their components. Depending on the variance of the [type constructor](http://en.wikipedia.org/wiki/Type_constructor), the subtyping relation may be either preserved, reversed, or ignored. For example, in [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)):

* IEnumerable<Cat> is a subtype of IEnumerable<Animal>. The subtyping is preserved because IEnumerable<T> is covariant on T.
* Action<Animal> is a subtype of Action<Cat>. The subtyping is reversed because Action<T> is contravariant on T.

# Neither IList<Cat> nor IList<Animal> is a subtype of the other, because IList<T> is invariant on T.

# Primitives

**Decimal**

This data type is designed to store variables with large decimals

**Primitives**

Collection

Sets

Arrays

ArrayList

Linked List

DoublyLinkedList

DoublyLinkedArrayList

Vector

HashTables

Maps

MultiMaps

Strings

Objects

Lists

Trees

Stacks

Queue

Sort Library

# Keywords

**Assert**

Assert describes a predicate (a true–false statement) placed in a java-program to indicate that the developer thinks that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort. Optionally enable by ClassLoader method.

**Inline**

Allow a block of assembly code to be inserted into the code

**Bytecode**

Allows you to, instead of assembly, write the intermediary byte code

**Null**

Discouraged. The **null** keyword is a literal that represents a null reference, one that does not refer to any object

**as**

if we want to give a module a different alias

**lambda**

creates a new anonymous function

**global**

access variables defined outside functions

**del**

deletes objects

**pass**

does nothing (like a nop)

**commandExecute**

Executes a line of code on the commnad line and returns its result in String form

**Goto**

Discouraged. Branching keywork.

**True and False**

Keywords that represent **the** boolean variables true and false.

**Wait**

The wait until commmand blocks execution until a certain condition is executed.For example:

Int i++;

Wait(classObj.getData() == true)

Remember CASE is by nature a multithreaded language

**Throw**

Causes the declared exception instance to be thrown. This causes execution to continue with the first enclosing exception handler declared by the catch keyword to handle an assignment compatible exception type. If no such exception handler is found in the current method, then the method returns and the process is repeated in the calling method. If no exception handler is found in any method call on the stack, then the exception is passed to the thread's uncaught exception handler.

**Text Literals**

Prefixing a stream of text, you do so by @texts:”My literal text”

**Throws**

Used in method declarations to specify which exceptions are not handled within the method but rather passed to the next higher level of the program. All uncaught exceptions in a method that are not instances of RuntimeException must be declared using the throws keyword.

**Transient**

Declares that an instance field is not part of the default [serialized](http://en.wikipedia.org/wiki/Serialization) form of an object. When an object is serialized, only the values of its non-transient instance fields are included in the default serial representation. When an object is deserialized, transient fields are initialized only to their default value. If the default form is not used, e.g. when a *serialPersistentFields* table is declared in the class hierarchy, all transient keywords are ignored.[16][17]

**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 then releasing the lock. The following example includes a **lock** statement.

class Account

{

decimal balance;

private Object thisLock = new Object();

public void Withdraw(decimal amount)

{

lock (thisLock)

{

if (amount > balance)

{

throw new Exception("Insufficient funds");

}

balance -= amount;

}

}

}

**Synchronized**

Used in the declaration of a method or code block to acquire the [mutex](http://en.wikipedia.org/wiki/Mutex) lock for an object while the current [thread](http://en.wikipedia.org/wiki/Thread_(computer_science)) executes the code.[10] For static methods, the object locked is the class's Class. Guarantees that at most one thread at a time operating on the same object executes that code. The mutex lock is automatically released when execution exits the synchronized code. Fields, classes and interfaces cannot be declared as *synchronized*.

**This**

Used to represent an instance of the class in which it appears. this can be used to access class members and as a reference to the current instance. The this keyword is also used to forward a call from one constructor in a class to another constructor in the same class.

**Namespaces**

The **namespace** keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

**Garbage in/out**

Turns on or off the garbage collector.

**Auto**

Defines a local variable to have a local lifetime.

**Sealed**

When applied to a class, the **sealed** modifier prevents other classes from inheriting from it. In the following example, class B inherits from class A, but no class can inherit from class B.

**Static Keyword**

static int i = 10;

static void PrintCR (void) { putc ('\n'); }

static tells that a function or data element is only known within the scope of the current compile. In addition, if you use the static keyword with a variable that is local to a function, it allows the last value of the variable to be preserved between successive calls to that function. Used to declare a field, method, or inner class as a class field. Classes maintain one copy of class fields regardless of how many instances exist of that class. static also is used to define a method as a class method. Class methods are [bound](http://en.wikipedia.org/wiki/Name_binding) to the class instead of to a specific instance, and can only operate on class fields. (Classes and interfaces declared as static members of another class or interface are actually top-level classes and are *not* inner classes.

**Assert**

Bug-testing. If the assert value is false, the program exits.

Assert(var == 5)

**Final**

Assign a constant variable to a value. Can only be assigned once.

**Uninitialized Conditional**

We may have gotten rid of nulls but we still have situations where there are uninitialized variables that we must check for. We have a new operator for that.

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

**String Concatenation –**

Use the concatenation operator: ( c )

Example: “I could sure use a “ ( c ) “sandwhich”

**Subtraction (Opposite of Concatenation)**

Use the anti-concatenation operator: ( m )

String myString = “This is my string”

>>myString ( r ) string

“This is my””

The match and return fro a String operator is

( m )

**Structs**

Groups variables into a single record

**Union**

A union is similar to a struct except it allows you to define variables that share storage space.

**Polymorphism**

Uses inheritance to implement Polymorphism. A subclass can be instantiated and used where its superclass normally would have been used.

**Native**

Use code from other languages.

**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.

**Volatile**

Can be changed by a background process

**Package**

A group of class files that are grouped and importable. The package keyword declares the hierarchy of classes. I.e.,

Package System.Output.TextHandler;

Or

Package System.Output.\*;

**Import**

Used to import libraries (packages) and data classes

E.g., import System.Output.TextHandler.

**Void**

Empty Data Type

**Sizeof**

Size in bytes

**Return**

Exits a function

**Short**

Small Int

**Char**

Basic Data Type

**Long**

Long Int

**Boolean**

A variable that holds eithetr a true or false variable

**Int**

Basic data type

**Double**

Floating point type

**Float**

Floating point type

**Fixed**

Fixed point type

**Uint**

Unsigned int

**Ushort**

Unsigned short

**Ulong**

Unsigned long

**Register**

Loads a value into a CPU register

**Extern**

Declares that an identifier is declared elsewhere

**Enum**

Set of constants of type int

**Continue**

Passes control to the beginning of the loop

**String Wildcards**

In addition to use wildcards for Regex and other reasons outlined, Wildcards can be used as a part of Strings for both passing Strings and passing as a variable, wildcards can be used for calling funtions and “forName” reflection.

**Mathematical Operators**

|  |  |
| --- | --- |
| ^ | Exponentiation |
| **- unary** | **Negate target** |
| **+ unary postiive operator** | **Oppsite of Negate** |
| **+** |  |
| **-** |  |
| **Div** | **Integer Division** |
| **/** | **Floating Point Division** |
| **\*** | **Multiplication** |
| **Mod** | **Modulo Operator** |
| **Rem** | **Remainder Operator** |
| **%** | **Percent Operator** |
| **(<<)** | **Shift Left Operator** |
| **(>>)** | **Shift Right Operator** |
| **=** | **Assignment Operator** |
| **==** |  |
| **<** | **Less than Operator** |
| **>** | **Greater than Operator** |
| **<=** | **Less than or Equal Operator** |
| **>=** | **Greater than or Equal Operator** |
|  |  |
| **<>** | **Not Equal To** |
| **!** | **Factorial** |
| **+ binary operator** | **Perform normal addition** |
| **- binary operator** | **Perform normal subtraction** |
| **++ postfix** | **Increment once after any operation it is nested in.** |
| **++prefix** | **Increment once after any operation it is involved at** |
| **--postfix** | **Decrement once after any operation it is nested in** |
| **--prefix** | **Decrement once before any operation it is involved in** |
| **+=** | **Compound addition** |
| **-=** | **Compound subtraction** |
| **/=** | **Compoun division** |
| **\*=** | **Compound multiplication** |
| **` (back tick)** | **Automatically evaluate entity** |
| **‘ (front tick)** | **Turn the expression into a list** |
| **. (period)** | **Use to join two brace contained statements** |
| **{ }** | **Braces to surround a statement or statements. Replace the semicolon. If a line exceeeds a line uses period to joint the two lines** |
| **,** | **Commas can be used to separate lines of code joined within braces. For example, {a =5, print())}** |
| **EndClass** | **Terminates a class** |
| **Exp** | **The exponential to a power** |
| **Abs** | **Absolute Value** |
| **Matrix** | **As in, Linear Algebra Matrices** |
| **Vector** | **As in, Linear Algebra Vectors**  **Similar to tuples** |
| **Membership Test: inMethod** | **Checks to see if a numeral datatype is in a vector, set, or an array. Checks to see if an item is a member of method** |
| **Membership To Class: inClass** | **Checks to see if a data type containsd a datatyp )(method or member variable)** |
| **Convo** | **Turns a string into a list** |
| **car** | **Peels off and returns the first element of a list** |
| **Cdr** | **Peels off the first element and returns the remainder** |

## Input and Output in CASE

**Web Services**

**Database Services**

**File Operations**

**Console Operations**

# Control Structures

**If….else**

Keyword if is used for conditional execution. The basic form of if uses the following syntax:

if (*expression*)

*statement1*

Alternatively, if may be used together with else, using the following syntax:

if (*expression*)

*statement1*

else

*statement2*

**Do…while**

A do while loop

**Where**

A string of instructions who calls an attached a ‘where’ function.

i.e., where

**Until**

Simple control structure. Until will execute blocks of code until the condition in Until condition is held true. For example,

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

**Residue**

Most iterative loops use a counter. The value “residue” contains a loop variable’s value when the loop ends. For example:

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

myROcket.getLaunchDate()

**residue** myRocketheight;

]

The result is:  
myRocketHeight = **reside**;

**Residue** in this case will return the rocket height and assignt it to the lhs.

**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)

{  
 currentCtr = **resolve;**

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)

}

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;

}

}

jelly.remove(remove);

**Iterator**

**Foreach Based Loops**

Foreach (TypeClass iteratorVariableName : Collection)

**Iterative For Based Loops**

For (variable initialize; conditional test; increment)

For exasmple, f(int i=0; i<10;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) )

**A Simpler For Statement**

Often times more information is processed then need be for a for loop.That is, a for loop reallly can be just:

for (<5) { Print “Print 5 times” }

Where the parathesis is the boolean condition and the braces are the execution.

**An automated While Loop**

Typically a while appears as such:

While (a<5)

{

}

We can make it so it is:

While (<5)

{ Print “Printing 5 times” }

{ Print “Current Counter” wcounter }

In this example wcounter is an automated variable: a variaable that’s built into the language.

**Switch**

**We can do better and more compact**

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

}

We obviously know that there are multiple cases and it is repetitive to keep on repeat the case statement as well as the breaks. Instead of doing break everytime, do a command when don’t break,. Also the variable t is obvious, we don’t need to include it in every case statement. Also lets just sample the booleans, we don’t need the variable. Example new case structure:

Switch(t)

{ ==5.0 } {returnString = 50}

{==0 } { returnString = “Wow its cold”}

{Crackers} { return String = “I am so hungry”

That’s all good and dandy, but what if we want the execuction to fall through or to completely exit, and what about default?

Instead of using the keyword using **break** we use the keyword **fall,** causing the execution to fall to the nexdt case. And as for default, we simply use default as normal. Example:

Switch(t)

{ ==5.0 } {returnString = 50, fall}

{==0 } { returnString = “Wow its cold”}

{Crackers} { returnString = “I am so hungry”, fall}

{==5} {returnString = “Where is my space ship?”, break}

{default} {returnString = “No idea”}

**..Case…default**

As of the new version of Java (7) switch statements are able to use switch case statements, switching on the String data type. We have taken it a step further. Why should the switch statement work on only integers and Strings?

CASE’s switch….

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;

break;

case (t == 0)

returnString = “Wow its cold”

break;

case (t==”Crackers”)

returnString = “I’m hungry”

break;

}

}

**Break**

Passes control out of a compound statement.

**Exit**

Passes control out of the current scope.

**Pass**

Passes control to the top of the current scope

**Branch ( ….)**

Go to a different thread

**Goto**

Go to a label

**Label**

A marker in the code for goto to leap to

**Jump**

Go to a function. If the function is in an uninstantiated class, it will create an anonymous innner class temporarily.

**List Comprehensions**

For example, let's say we need to create a list of integers which specify the length of each word in a certain sentence, but only if the word is not the word "the".

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

words = sentence.split()

word\_lengths = []

for word in words:

if word != "the":

word\_lengths.append(len(word))

Using a list comprehension, we could simplify this process to this notation:

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

words = sentence.split()

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

# Classes and Inhertance

**Object Allocation**

Objects can be allocated and initialized at different locations in the code.

**Inner Classes**

(no access modifier)

Class MyInnerClass { class methods and variables }

**Anonymous Inner Classes**

myFunction(new Class { anonymous class });

**No Boxing**

The primitve types act as full-fledge built in classes. More like keywords with priveliges. Therefore instead of having to say int b = Integer.parseInt(mystring) we can just say (in this case since we are using Strings…) int b = mystring; my string casts its type down to that of an integer – b. Say we wanted to use the float primitive and wanted to cast it to a int. All you have to do is say int b = float a.

An even better example is say I have an object Cat and an object Dog. We can do the following: Cat = Dog or Dog = Cat , that is Dog is assigned to Cat and vice versa.

**Object Operators**

|  |  |
| --- | --- |
| Is | Object1 Is Object2  Do the two objects point to the same reference? |
| IsNot or IsNothing | Not of IS |
| typeOf | If typeOf someNumber is an Integer  Compares Compare TypeClasses |
| instanceOf | InstanceOf Class1 - Is an object of a typeclass Class1? Compare ObjectType |
| getType | Returns System.Class info for the getType |
|  |  |
|  |  |
|  |  |

**MetaClass**

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.

**Categories**

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.

**Prototype Inheritance**

Ability to get a new object from another. Not classes, but inheritance directly from an object. **Proto** type.

**Classes**

A type that defines the implementation of a particular kind of object. A class definition defines [instance](http://en.wikipedia.org/wiki/Object_(computer_science)) and class [fields](http://en.wikipedia.org/wiki/Field_(computer_science)), [methods](http://en.wikipedia.org/wiki/Method_(computer_science)), and [inner classes](http://en.wikipedia.org/wiki/Inner_class) as well as specifying the [interfaces](http://en.wikipedia.org/wiki/Interface_(computer_science)) the class implements and the immediate [superclass](http://en.wikipedia.org/wiki/Superclass_(computer_science)) of the class. If the superclass is not explicitly specified, the superclass is implicitly [Object](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html). The class keyword can also be used in the form Class**.class** to get a Class object without needing an instance of that class. For example, **String.class** can be used instead of doing **new String().getClass()**.

**Properties**

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called accessors. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

In this example, the TimePeriod class stores a time period. Internally the class stores the time in seconds, but a property named Hours enables a client to specify a time in hours. The accessors for the Hours property perform the conversion between hours and seconds.

class <TimePeriod>

{

private <double> seconds;

public <double> Hours

{

get { return seconds / 3600; }

set { seconds = value \* 3600; }

}

}

**Expression Body Methods**

Instead of writing an entire function for one or two lines of code, especially when that code is very simplistic you can use expression body methods:

Syntax:

Identifier => { body of code }

Identifier is the lookup value in the entity pool for example:

while (true)

{

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

}

public void

{

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

or simple enough

pool.run(simple\_print\_hello)

}

**Interfaces**

An interface contains only the signatures of [methods](https://msdn.microsoft.com/en-us/library/ms173114.aspx), [properties](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx), [events](https://msdn.microsoft.com/en-us/library/awbftdfh.aspx) or [indexers](https://msdn.microsoft.com/en-us/library/6x16t2tx.aspx). A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition. Interfaces cannot be instantiated on their own.

Public MyClass implements <MyInterface>

**Abstract Class**

*abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. If a class includes abstract methods, then the class itself *must* be declared abstract, as in:

**Abstract Method**

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

**Class Constructors**

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

**construct public int** Apple(int numberOfApples)

{ numberOfApples++;return numberOfApples;}

**Type Constructors**

A type constructor is similar to a Haskell constructor in that it defines the chain of types alllowable in a function or class.

Example:

Object->String->List->Integer

The first element is the root type, the second element is the types allowed as parameters, and the last element is the returned value.

**Data Constructor**

Not defined yet

**Explicit**

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));

}

**Extend**

Define a class that extends from a superclass. Inheritance allows re-use of methods of the super class, re-use of the variables, and new content like methods and variables.

**Implement**

Implements an interface, implements the abstract functions in the interface.

**More on Inhertance**

Public MyClass extends SomeSuperClass

**Super**

Used to access members of a class inherited by the class in which it appears. Allows a subclass to access [overridden](http://en.wikipedia.org/wiki/Method_overriding_(programming)) methods and hidden members of its superclass. The super keyword is also used to forward a call from a constructor to a constructor in the superclass.

Also used to specify a lower bound on a type parameter in Generics.

**String Inheritance**

<Int> b = “This is an integer”

Since everything is a string, a person can, for example, assign a string to an integer, since a String is higher up on the hierarchy.

**Iterators**

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

<MyIteratableClass>

<Iterator> iter = <MyIterableClass>;

For (Iterator <iter> : <MyIterableCLass>)

Iter.printString(\*iter)

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

**is**

Checks if an object is compatible with a given type. For example, the following code can determine if an object is an instance of the **MyObject** type, or a type that derives from **MyObject**:

if (obj is <MyObject>)

{

}

An **is** expression evaluates to **true** if the provided expression is non-null, and the provided object can be cast to the provided type without causing an exception to be thrown.

The **is** keyword causes a compile-time warning if the expression is known to always be **true**

or to always be **false**, but typically evaluates type compatibility at run time.

The **is** operator cannot be overloaded.

Note that the **is** operator only considers reference conversions, boxing conversions, and unboxing conversions. Other conversions, such as user-defined conversions, are not considered.

Anonymous methods are not allowed on the left side of the **is** operator. This exception includes lambda expressions.

**Inclusion**

Often times you have to initialize content within for a for loop, for example, but you can’t because it is dependent on the loop. A Java example will clarify:

<String> output = **null**;

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

{

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.

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

{

inclusion output = output + tags.name;

output = output + tags.name;

}

**rename**

Able to rename one class to something else dynamically

rename <BobsClass> => <TedsClass>

**object Type**

The **object** type is an alias for [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx) type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx). You can assign values of any type to variables of type **object**. When a variable of a value type is converted to object, it is said to be boxed. When a variable of type object is converted to a value type, it is said to be unboxed. For more information, see [Boxing and Unboxing](https://msdn.microsoft.com/en-us/library/yz2be5wk.aspx). The direct subclass of object is String. Converting a data type to String is called **Translating** with translating downwards called **Translating Downwards** and translating upwards to the Object is **Transcribing.**

[Example](javascript:void(0))

The following sample shows how variables of type **object** can accept values of any data type and how variables of type **object** can use methods on [Object](https://msdn.microsoft.com/en-us/library/system.object.aspx) from the .NET Framework.

**C#**

class <ObjectTest>

{

public <int> i = 10;

}

class <MainClass2>

{

static void Main()

{

<object> a;

a = 1; // an example of boxing

Console.WriteLine(a);

Console.WriteLine(a.GetType());

Console.WriteLine(a.ToString());

a = new <ObjectTest()>;

<ObjectTest> classRef;

classRef = (<ObjectTest>)a;

Console.WriteLine(classRef.i);

}

}

/\* Output

1

System.Int32

1

\* 10

\*/

**Auto-Property Initializer**

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.

class <PeopleManager>

{

public List<string> Roles { get; } =

new List<string>() { "Employee", "Managerial"};

**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)

{  
 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;

}

}

**Forname (Reflection)**

<Class<?>> c = <Class<?>.forName(args[0]);

<Object> t = c.newInstance();

is the Java Syntax. We can do better.

Let <?> refer to any unknown class type.

Class<?> c = “name of class”

Since we have String as a root base class, the compiler sees the Class<?> when it should be defining a class (see the following c). So it checks the r.h.s. for any class, finds the root class and creates it.

Alternatively, you can do something like:

Class<?> i = int[10] and it would create a class of type integer array with ten slots.

Anyway, we don’t need object t or anything of the sort.

**Generics (Circa 2015)**

In the past, often in Java you had to use following syntax for Generics:

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

Now, going along with the new release of Java, generics can be implemented as the following, without the right hand side of the equation.

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

The <> operator in this tense acts as a typeclass wildcard. It is called the diamond operator. The compiler will try to find a match for that operator according to the types given.

**Multiple Statements on one line**

Use the double bar operator

<int> n = 10 || msgBo(“N is “ ( c ) I);

or

<int> n= 10 || k = 20 || n

# Arrays

**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]

**Storing code in an Array**

>>[ for int i=0;i<size;i++;print i]=myLoopArray

>>[myLoopArray]

>>1

>>2

>>3

>>4

>>5

>>6

**Multidimensional Arrays**

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]

You can use as many dimensions as you like.

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble [properties](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx) except that their accessors take parameters.

In the following example, a generic class is defined and provided with simple [get](https://msdn.microsoft.com/en-us/library/ms228503.aspx) and [set](https://msdn.microsoft.com/en-us/library/ms228368.aspx) accessor methods as a means of assigning and retrieving values. The Program class creates an instance of this class for storing strings.

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;

}

}

}

// This class shows how client code uses the indexer.

class Program

{

static void Main(string[] args)

{

// Declare an instance of the SampleCollection type.

SampleCollection<string> stringCollection = new SampleCollection<string>();

// Use [] notation on the type.

stringCollection[0] = "Hello, World";

System.Console.WriteLine(stringCollection[0]);

}

}

// Output:

**Primary Constructor**

Primary Constructor is a feature in which you are allowed to pass the constructor parameters at the class declaration level instead of writing a separate constructor. The scope of the primary constructor parameters values is class level and will be available only at the time of class initialization. It comes to good use when it is used with the Auto-Property initializers.

// Primary constructor

class <Basket>(string item, int price)

{

// Using primary constructor parameter values

// to do auto property initialization.

public string Item { get; } = item;

public int Price { get; } = price;

}

**Base**

The **base** keyword is used to access members of the base class from within a derived class:

* Call a method on the base class that has been overridden by another method.
* Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the **base** keyword from within a static method.

The base class that is accessed is the base class specified in the class declaration. For example, if you specify class ClassB : ClassA, the members of ClassA are accessed from ClassB, regardless of the base class of ClassA.

**New keyword**

In C#, the **new** keyword can be used as an operator, a modifier, or a constraint.

[new Operator](https://msdn.microsoft.com/en-us/library/fa0ab757.aspx)

Used to create objects and invoke constructors.

[new Modifier](https://msdn.microsoft.com/en-us/library/435f1dw2.aspx)

Used to hide an inherited member from a base class member.

[new Constraint](https://msdn.microsoft.com/en-us/library/sd2w2ew5.aspx)

Used to restrict types that might be used as arguments for a type parameter in a generic declaration.

**Sizing Issues**

One thing that is always an annoyance in Java and C like languages is ArrayIndexOutofBound exceptions, NullPointerExceptions, and IOStreamExceptions. CASE’s solution to this is this. The CASE compiler uses AI to verify with 90% accuracy whether those boundaries will be crossed or not. It uses predictive software and a built in syntactc database . That way when you compile, you are warned if there is a possibole arrayindexoutoffbound

**The Solution to Seg Faults**

Seg faults are a problem of low-level languages like C that access memory directly. Most agents have gone away from pointers as their solution to this problem. We would still like to have pointeres so we need to find a way around this. So what we do is this. Built into the compiled exeutable at every memory manipulation point such as the establishment of a pointer or calling the new keyword, a unit called the Recyclabler built into the executable throws an exception but instead of aborting, it continues to run.

# Tuples

Tuples are similar to arrays except they are ordered. With a tuple you can return multiple values without specifying multiple return valuefs. They can be used to represent coordinates as well.

# Sets

Sets are like arrays and tuples except they aren’t ordered and there is only one instance of each They are the same as the mathemical set.

# Types

**Nested Types**

String -> Array -> Float (An array of floats)

**Cross-Typing**

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.

**Type Brackets**

Whenever you see the angle brackets you know you are referring to a type.

<ArrayList<ArrayLIst>> myDoubleList = new <ArrayList<ArrayList>>

or

<T> myFunction = new <T>

or

HashMap <String, String>

or

<T> templateReturn = new <ArrayList<ArrayList>>

or

<Collections<T>> collectiontemplate = new <Colllections<T>>

**Type Generics**

**Shall keyword**

Instead of using private or public access modifiers, CASE can give acces to specific

Classes. So,

Class <mYClass> shall <myFriendsClass>

**Friend Keyword**

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

Call despite any access modifiers.

class <Rectangle> {

int width, height;

public:

Rectangle() {}

Rectangle (int x, int y) : width(x), height(y) {}

int area() {return width \* height;}

friend Rectangle duplicate (const Rectangle&);

};

>

<Rectangle duplicate (const Rectangle& param)

{

Rectangle res;

res.width = param.width\*2;

res.height = param.height\*2;

return res;

}

# Type Templates

The notation <T> is used when a generic type willl suffice.

Examples:

String->Class-><T>

Public Class<T> Matrix

Stream <T>[][] myStream = new <T>[][];

<T> (myStream) myMatrixVar;

String label =@: “This is a matrix class”

# Functions

**Protocols**

Protocols - Programmers gets a flexibility of implementing @optional or @required methods. (From Objective C)

**Extension Methods**

*Extension methods* in C# allow programmers to use static methods as if they were methods from a class's method table, allowing programmers to add methods to an object that they feel should exist on that object and its derivatives.

**Caller**

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 Return Values**

In method definition, can use ret keyword to declare more than one return value

sample definition:

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

{

return (1,2);

}

**Procedure Keyword**

The procedure keyword locks down its type, its type being like a function but not able to return any value. A procedure is essentially a segment of code with parameters.

**Function Keyword**

The function keyword locks down the type, so it can’t be used in another method.

**Targets**

A function can be considered an opening into an object. To reduce coupling, we offer for functions the target keyword.

Example:

target @ function1, function2, function3

myTestTargetFunction { }

Then the only functions that can call the target function is function1, function2, and function3. Any others will be rejected.

**Dependency Injection**

CASE has depdendency built into the compiler.

Say you have the variable int numberOfForkLifts;

And you have a function called injecftForkLists; such as:

int numberOfForkLists = 0;.

Then in another function:

//below gets a pointer to an object in the entity pool

numberOfForkLifts\_Local = entityPool.get(“numberOfForkLifts”, pointer=true)

//below injects – basically sticks in the value 5 into a global variable managed by

//the entity pool

**inject** numberOfFOrkLists\_Local = 5;

//now there are 5 fork lifts

**Anonymous Embedded Functions**

Define and declare a funcction within another function. It can be empty but it doesn’t need to be.

Public void <Volleyball> Class

playVolleyball

[ Serve [ Hit Volleyball; Bump VolleyballBack ]

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

]

**SynOperator**

Used when you want to have one method call with the same signature but able to do different things. It is essentially overriding a function without changing the signature.

**Virtual**

The **virtual** keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

**Nameless Parameters**

public static 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.

**Infix Option**

That is, you write (myFunction arg1 arg2 arg3) in LISP instead of the infix/proper notation like myFunction(arg1, arg2, arg3) in imperative languages - the benefits

**Wildcard Functions**

Similar to function overloading, wildcards allow for a wildcard to be substiuted into a wild card function. For example,

public void addTwo(x, y)

public int addTwo(x,y)

public void addTwo(\_, y)

public int addTwo(x, \_)

**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**

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);

**Delegates**

* Delegates are like C++ function pointers but are type safe.
* Delegates allow methods to be passed as parameters.
* Delegates can be used to define callback methods.
* Delegates can be chained together; for example, multiple methods can be called on a single event.

The declaration of a delegate type is similar to a method signature. It has a return value and any number of parameters of any type:

public delegate void TestDelegate(string message);

public delegate int TestDelegate(MyType m, long num);

* A **delegate** is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure. For applications of delegates, see [Delegates](https://msdn.microsoft.com/en-us/library/ms173171.aspx) and [Generic Delegates](https://msdn.microsoft.com/en-us/library/sx2bwtw7.aspx).
* // Declare a delegate:
* delegate void Del(int x);
* // Define a named method:
* void DoWork(int k) { /\* ... \*/ }
* // Instantiate the delegate using the method as a parameter:
* Del d = obj.DoWork;.
* delegate void Del(int i, double j);
* class MathClass
* {
* static void Main()
* {
* MathClass m = new MathClass();
* // Delegate instantiation using "MultiplyNumbers"
* Del d = m.MultiplyNumbers;
* // Invoke the delegate object.
* System.Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
* for (int i = 1; i <= 5; i++)
* {
* d(i, 2);
* }
* // Keep the console window open in debug mode.
* System.Console.WriteLine("Press any key to exit.");
* System.Console.ReadKey();
* }
* // Declare the associated method.
* void MultiplyNumbers(int m, double n)
* {
* System.Console.Write(m \* n + " ");
* }
* }
* /\* Output:
* Invoking the delegate using 'MultiplyNumbers':
* 2 4 6 8 10

**Implicit**

The **implicit** keyword is used to declare an implicit user-defined type conversion operator. Use it to enable implicit conversions between a user-defined type and another type, if the conversion is guaranteed not to cause a loss of data.

[Example](javascript:void(0))

**C#**

class <Digit>

{

public Digit(double d) { val = d; }

public double val;

// ...other members

// User-defined conversion from Digit to double

public static implicit operator double(Digit d)

{

return d.val;

}

// User-defined conversion from double to Digit

public static implicit operator Digit(double d)

{

return new Digit(d);

}

}

class <Program>

{

static void Main(string[] args)

{

Digit dig = new Digit(7);

//This call invokes the implicit "double" operator

double num = dig;

//This call invokes the implicit "Digit" operator

Digit dig2 = 12;

Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);

Console.ReadLine();

}

}

By eliminating unnecessary casts, implicit conversions can improve source code readability. However, because implicit conversions do not require programmers to explicitly cast from one type to the other, c

are must be taken to prevent unexpected results. In general, implicit conversion operators should never throw exceptions and never lose information so that they can be used safely without the programmer's awareness. If a conversion operator cannot meet those criteria, it should be marked **explicit**. For more information, see [Using Conversion Operators](https://msdn.microsoft.com/en-us/library/85w54y0a.aspx)..

# Integrated Functions

**These functions are part of the library functions that come with CASE. All all are derivatives of the String class.**

**Root Chain of Classes**

String->Object->System-> any implementable

**toString**

This function must be implemented by any class directly inheriting from String.

**fromString**

Initializes the named class from the entity pool.

**convert**

Convert the calling class into another class. For example, if your class is Decimal, it needs to be able to Integer, etc. When convert is called, it will look in the entity pool to see what classes are available. The calling class has to implement them.

# Pointers

**Not so messy Pointers**

At root variables exist in streams. First create a variable in the stream. Then simply assign the address operator & to the variable.

stream n = new TypeName

TypeName (n) variableName

TypeName (n) BobsPointer = &variableName

**Pointer Dereference Shortcut**

As borrowed from C,

(\*myClass).pointer is the same as myClass->pointer

**Pointer Arithmetic**

Pointer Arithmetic is simple.

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)

**Multiple Pointers**

As you can see, pointers can only live in the same stream. This helps to keeep from more than one accessing a single memory location.

**Pass by Const Reference**

In this case, the const keyword makes it so that when you pass a large object reference you don’t’ accidently change the object.

functionName(const int parameter)

# String Operations

**Formatting**

There is a output and input low-level format set of functions similar to pure C.

Unlike the C++ style input stream, scanf and printf can work with any kind of type without having to explicitly define a type for the operator.

Ie.,

Printf(“My name is %s”, “Ted”); 🡨 a normal print statement

Print(“My name is %c %c %c”, Jason, 5, array);

Where Jason is a string, 5 is an integer, and array is an array of ints,

Also there is an unlimited number of parameters for the format identifiers.

Also there is scanf() which reads an unlimited amount of data into a variables.

Scanf(“hello %s”, “world”);

Reads string into world.

**Format Specifier**

|  |  |  |
| --- | --- | --- |
| i | Integer | Any number of digits, optionally preceded by a sign (+ or -).  [Decimal digits](http://www.cplusplus.com/isdigit) assumed by default (0-9), but a 0 prefix introduces octal digits (0-7), and 0x [hexadecimal digits](http://www.cplusplus.com/isxdigit) (0-f).  *Signed* argument. |
| d *or* u | Decimal integer | Any number of [decimal digits](http://www.cplusplus.com/isdigit) (0-9), optionally preceded by a sign (+ or -).  d is for a *signed* argument, and u for an *unsigned*. |
| o | Octal integer | Any number of octal digits (0-7), optionally preceded by a sign (+ or -).  *Unsigned* argument. |
| x | Hexadecimal integer | Any number of [hexadecimal digits](http://www.cplusplus.com/isxdigit) (0-9, a-f, A-F), optionally preceded by 0x or 0X, and all optionally preceded by a sign (+ or -).  *Unsigned* argument. |
| f, e, g | Floating point number | A series of [decimal](http://www.cplusplus.com/isdigit) digits, optionally containing a decimal point, optionally preceeded by a sign (+ or -) and optionally followed by the e or E character and a decimal integer (or some of the other sequences supported by [strtod](http://www.cplusplus.com/strtod)).  Implementations complying with C99 also support hexadecimal floating-point format when preceded by 0x or 0X. |
| a |
| c | Character | The next character. If a *width* other than 1 is specified, the function reads exactly *width* characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end. |
| s | String of characters | Any number of non-whitespace characters, stopping at the first [whitespace](http://www.cplusplus.com/isspace) character found. A terminating null character is automatically added at the end of the stored sequence. |
| p | Pointer address | A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format %p in [fprintf](http://www.cplusplus.com/fprintf). |
| [*characters*] | Scanset | Any number of the characters specified between the brackets.  A dash (-) that is not the first character may produce non-portable behavior in some library implementations. |
| [^*characters*] | Negated scanset | Any number of characters none of them specified as *characters* between the brackets. |
| n | Count | No input is consumed.  The number of characters read so far from [stdin](http://www.cplusplus.com/stdin) is stored in the pointed location. |
| % | % | A % followed by another % matches a single %. |
| Except for n, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends t |  |  |

The above n and paranthesis acts as an operator for concatenation for strings.

**Escape Characters**

|  |  |
| --- | --- |
| \t | Insert a tab in the text at this point. |
| \b | Insert a backspace in the text at this point. |
| \n | Insert a newline in the text at this point. |
| \r | Insert a carriage return in the text at this point. |
| \f | Insert a formfeed in the text at this point. |
| \' | Insert a single quote character in the text at this point. |
| \" | Insert a double quote character in the text at this point. |
| \\ | Backslash |
| \A | Alarm (Beep, Bell) |
| \t | Horizontal Tab |
| \v | Vertical Tab |
| \? | Quotation Mark |
| \nnn | The character whose numerical value is given by *nnn* interpreted as an [octal](http://en.wikipedia.org/wiki/Octal) number |
| \xhh | The character whose numerical value is given by *hh* interpreted as a [hexadecimal](http://en.wikipedia.org/wiki/Hexadecimal) number |
|  |  |
|  |  |

**Insertion Operator << and Extraction Operator >>**

Use to pass to and from a C style stream. Used mainly for output and input from

the console and files.

# Parameters

**Pass by Value**

Passes a copy of a primitive or an ENTIRE object – including deep copy.

functionName(int parameter)

functionName(deep parameter)

**Pass by Need**

This uses delay and force actions like in Haskell. In this case, a promise is passed and it makes due on that promise when the variable is needed.

functionName(need parameter)

**Pass by Reference**

This is called when there is a large object and it doesn’t make sense to pass an entire copy of it.

functionName(&parameter)

**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)

**Operator Overloading**

(Binary Operator)

int +operator(int rhs, int lhs) { return rhs+lhs }

(or, using ret)

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

(Unary Operator)

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

**Function Overloading**

More than one signature for a function with the same signature.

**Function Overriding**

Same signature except the content of the body of the function is different (overriden). Used on super classes and other things inherited from.

**Creating Your Own Operator**

All custom operators follow the format of having a single letter surounded ny paranethesis and the surrounded by white space.

For example:

Int n;

Iint q;

Int z;

Z = n (h) q

The actual declaration of the operator is as follows:

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]

# Equalities

**Assignment: Deep vs Shallow Copy**

Assume we have two classes instantiated, ClassA and ClassB.

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated.

In this syntax its,

ClassA ~ ClassB (shallow copy)

ClassA = ClassB (deep copy)

**String Equality**

There are three equalities when it comes to strings. The first is the equivalence of their reference, the second is the copy of the content, the final is treating it as a the

Universal base class. CASE treats the first case as an equivalence when the

Following:

StringA &= StringB

The second is the content:

StringA = StringB

Finally, the third as a base class

If (StringA =$ MyFloatClass) { ….. }

# Access Modifiers

**Access Modifiers**

**public** : Access is not restricted.

**protected** : Access is limited to the containing class or types derived from the containing class.

**Internal** : Access is limited to the current library.

[protected internal](https://msdn.microsoft.com/en-us/library/ms173121.aspx): Access is limited to the current library or types derived from the containing class.

**private** : Access is limited to the containing type.

# Event Handling

Event handling is a core part of CASE.

Events enable a [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the publisher and the classes that receive (or handle) the event are called subscribers.

Example:

Subscriber subscribeQueue

Publisher publishQueue

subscribeQueue

.(publishingClass.publishingFunction) { body of code }

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

# Constants

**Const**

When used with a variable: const variablename = variablevalue

Warning: a const variable can be indirectly modified by a pointer, as in the following example:

\*(int\*)&my\_age = 35;

When the const modifier is used with a pointer parameter in a function's parameter list, it uses the following syntax:

*function-name* (const *type* \**var-name*)

You use the **const** keyword to declare a constant field or a constant local. Constant fields and locals aren't variables and may not be modified. Constants can be numbers, Boolean values, strings, or a null reference. Don’t create a constant to represent information that you expect to change at any time. For example, don’t use a constant field to store the price of a service, a product version number, or the brand name of a company. These values can change over time, and because compilers propagate constants, other code compiled with your libraries will have to be recompiled to see the changes. See also the [readonly](https://msdn.microsoft.com/en-us/library/acdd6hb7.aspx) keyword. For example:

const int x = 0;

public const double gravitationalConstant = 6.673e-11;

private const string productName = "Visual C#";

# Cons/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.

# Futures and Promises

Futures are read-only placeholder for a variable

Promise is a one time writable placeholder sets the value of the future

Suppose we call an API, then we will be actually blocking. If we use a Future we establishe a promise, removes blocking? Gets a portion back.

# Recursion

Although it is a very excellent technique to have under ones belt, recursion can be terrible to learn and get right. There are few common aspects to recursion that CASE can help you with the appropriate keywords.

base = { } a scope set aside for the base case of the recursion in the function

tail = { } additional information/code you want added to to your return call

The compiler will take this information and spit out a recursive function.

# Annotations

*Annotations*, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

* **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
* **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
* **Runtime processing** — Some annotations are available to be examined at runtime.

# Comments

// Line Comments

/\* \*/ Block Comments

/\*\*\*\*\*/ Auto-document

# Generator

The idea of generators is to calculate a series of results one-by-one on demand (on the fly). In the simplest case, a generator can be used as a list, where each element is calculated lazily. Lets compare a list and a generator that do the same thing - return powers of two:

>>> # First, we define a list

>>> the\_list = [2\*\*x for x in range(5)]

>>>

>>> # Type check: yes, it's a list

>>> type(the\_list)

<class 'list'>

>>>

>>> # Iterate over items and print them

>>> for element in the\_list:

...     print(element)

...

1

2

4

8

16

>>>

>>> # How about the length?

>>> len(the\_list)

5

>>>

>>> # Ok, now a generator.

>>> # As easy as list comprehensions, but with '()' instead of '[]':

>>> the\_generator = (x+x for x in range(3))

>>>

>>> # Type check: yes, it's a generator

>>> type(the\_generator)

<class 'generator'>

>>>

>>> # Iterate over items and print them

>>> for element in the\_generator:

...     print(element)

...

0

2

4

>>>

>>> # Everything looks the same, but the length...

>>> len(the\_generator)

Traceback (most recent call last):

  File "", line 1, in

TypeError: object of type 'generator' has no len()

Iterating over the list and the generator looks completely the same. However, although the generator is iterable, it is not a collection, and thus has no length. Collections (lists, tuples, sets, etc) keep all values in memory and we can access them whenever needed. A generator calculates the values on the fly and forgets them, so it does not have any overview about the own result set.

Generators are especially useful for memory-intensive tasks, where there is no need to keep all of the elements of a memory-heavy list accessible at the same time. Calculating a series of values one-by-one can also be useful in situations where the complete result is never needed, yielding intermediate results to the caller until some requirement is satisfied and further processing stops.

A good example is a search task, where typically there is no need to wait for all results to be found. Performing a file-system search, a user would be happier to receive results on-the-fly, rather the wait for a search engine to go through every single file and only afterwards return results. Are there any people who really navigate through all Google search results until the last page?

Since a search functionality cannot be created using list-comprehensions, we are going to define a generator using a function with the yield statement/keyword. The yield instruction should be put into a place where the generator returns an intermediate result to the caller and sleeps until the next invocation occurs. Let's define a generator that would search for some keyword in a huge text file line-by-line.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def search(keyword, filename):      print('generator started')      f = open(filename, 'r')      # Looping through the file line by line      for line in f:          if keyword in line:              # If keyword found, return it              yield line      f.close() |

mentation Comments

# Module Connection

Individual modules of code are compiled individually but using an XML web-services like interface between them. This is CASE’s answer to Web Services

Keywords are converted into tags such as:

Print “Hello World”

Becomes

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

**Modularity**

Modularity is the most important thing in CASE. If you can’t limit a function to less than five lines then that function is too long.

Common CASE function in Java Syntax:

Concat(a):

{ String b}n is not

{ for c : a { b ( c ) return last }

Print(str):  
{ Output Concat(a) }

Concat is a function definition

String b is a declaration of a string type

For c: a operates over a, where a is a string list

( c) is the concetenation operator

last evaluates to the last expression

**Code Braces**

This is another issue. Code braces often get out of control. In reality it is not a viable solution. We want a function or a class to be small and compact. So our solution is to differentiate, that we the code doesn’t become obfuscated. Single one-liners of code as above use braces. This eliminates the need for a trailing semicolon. The next up is functions. Definitions of functions are code blocks surrounded by brackets [ ]. Definitions of types, such as Interfaces or Classes scope is determined by indentation and with an # sign in front of the word class. Also, there are three ways to call a function.

1. infix (f a b) (lisp way)

2. C style f(a,b)

3. bracket style [f : a b]

**Code Behind, Code Beside**

In code beside, all code per class is in one file like Java. In code behind mode, the function defintion is in the class file definition and the implementation is in a separate, .cas file.

To use code behind, leave out one or more code implementations in the class file definition (.cah) file and put the implementation in a .cas file. To use the implementation file along with the class definition use the include file.

To include .cah and .cas file,

//Main.cah file

String->Object->Main

#public class Main

[public void main()]

end class

//Main.cas file

include “main.cas”

String->Object->Main

#public class Main

[public void main()

{Print “Hello World”}

]

# Model View Controller Design Pattern Integration

View can only access the model (database) through the controller, as well as the database can only access the view through the controller. Essentially it is divided into three parts. MVCs are connected via the following prototype code:

@Controller:MyAppController

ViewClass

@Controller:MyAppController

ModelClass

The annotations above indicate what the class is connected to. Messages can be passed in one direction from one unit to another.

# Byte Code

**Implementing ByteCode**

It is also legal to write bytecode like one would write assembly. The extension for a bytecode file is .cab and can be included just like an cah file.

**ByteCode commands**

Comments: preceded by // c++ style

Or to span multiple lines /\*\* C style

Label: a label is immediately followed by a colon and consists of a symbol or numeral.

Numbers: Octal, Hex, and Dec are supported

String: A series of characters surrounded by double quotes

Character: A symbol surrounded by single quotes

*Table 2-1* Escape Codes Recognized in Strings

add %g1,’a’-’A’,%g1 ! g1 + (’a’ - ’A’) --> g1

**Escape Code**

\a  
\b  
\f  
\n  
\r  
\t  
\v \*nnn* \x*nn...*

**Description**

Alert  
Backspace  
Form feed  
Newline (line feed) Carriage return Horizontal tab Vertical tab

Octal value nnn Hexadecimal value nn...

Syntax for a symbol name is:

{ letter | \_ | $ | . } { letter | \_ | $ | . | digit }\*

* Uppercase and lowercase letters are distinct; the underscore ( \_ ), dollar sign ($), and dot ( . ) are treated as alphabetic characters.
* Symbol names that begin with a dot ( . ) are assumed to be local symbols. To simplify debugging, avoid using this type of symbol name in hand-coded assembly language routines.
* The symbol dot ( . ) is predefined and always refers to the address of the beginning of the current assembly language statement.
* External variable names beginning with the underscore character are reserved by the ANSI C Standard. Do *not* begin these names with the underscore; otherwise, the program will not conform to ANSI C and unpredictable behavior may result.

1. *Special Symbols - Registers*Special symbol names begin with a *percentage sign* (%) to avoid conflict with

user symbols. Table 2-2 lists these special symbol names. *Table 2-2* Special Symbol Names

Symbol Object Name

General-purpose registers %r0 ... %r31 General-purpose global registers %g0 ... %g7 General-purpose out registers %o0 ... %o7 General-purpose local registers %l0 ... %l7 General-purpose in registers %i0 ... %i7

Stack-pointer register %sp Frame-pointer register %fp

Floating-point registers %f0 ... %f31 Floating-point status register %fsr  
Front of floating-point queue %fq Coprocessor registers %c0 ... %c31 Coprocessor status register %csr Coprocessor queue %cq

Program status register %psr Trap vector base address register %tbr Window invalid mask %wim Y register %y

**Symbol Object**

Unary operators

Ancillary state registers

**Name Comment**

%lo Extracts least significant 10 bits

%hi Extracts most significant 22 bits

%r\_disp32 Used only in Sun compiler-generated

%r\_plt32 code.  
Used only in Sun

compiler-generated code.

%asr1 ... %asr31

There is no case distinction in special symbols; for example,

%PSR

is equivalent to

%psr

The suggested style is to use lowercase letters.

The lack of case distinction allows for the use of non-recursive preprocessor substitutions, for example:

#define psr %PSR

The special symbols %hi and %lo are true unary operators which can be used in any expression and, as other unary operators, have higher precedence than binary operations. For example:

%hi a+b = (%hi a)+b %lo a+b = (%lo a)+b

To avoid ambiguity, enclose operands of the %hi or %lo operators in parentheses. For example:

%hi(a) + b

***Operators and Expressions***

The operators described in Table 2-3 are recognized in constant expressions. *Table 2-3* Operators Recognized in Constant Expressions

%hi a+b = (%hi a)+b %lo a+b = (%lo a)+b

%hi(a) + b

**Binary Operators**

+ Integer addition  
– Integer subtraction  
\* Integer multiplication / Integer division  
% Modulo  
^ Exclusive OR

<< Left shift

>> Right shift

**Unary Operators**

+ (No effect)  
– 2's Complement  
~ 1's Complement  
%lo Extract least significant 10 bits %hi Extract most significant 22 bits

%r\_disp32 Used in Sun compiler-generated code only to instruct the assembler to

generate specific relocation information for the given expression.

%r\_plt32 Used in Sun compiler-generated code only to instruct the assembler to

generate specific relocation information for the given expression.

**Bitwise Operators**

& Bitwise AND  
| Bitwise OR

Since these operators have the same precedence as in the C language, put expressions in parentheses to avoid ambiguity.

To avoid confusion with register names or with the %hi,%lo, %r\_disp32, or %r\_plt32 operators, the modulo operator % must *not* be immediately followed by a letter or digit. The modulo operator is typically followed by a space or left parenthesis character.

"bgeu,a label"

*Table 5-3*

SPARC to Assembly Language Mapping

|  |  |  |
| --- | --- | --- |
| **Mnemonic** | **Argument List** | **Operation** |
| add addcc addx addxcc | *reg*rs1*, reg\_or\_imm, regrd reg*rs1*, reg\_or\_imm, regrd reg*rs1*, reg\_or\_imm, regrd reg*rs1*, reg\_or\_imm, regrd* | Add Add and modify icc Add with carry |
| and andcc andn andncc | *reg*rs1*, reg\_or\_imm, regrd reg*rs1*, reg\_or\_imm, regrd reg*rs1*, reg\_or\_imm, regrd reg*rs1*, reg\_or\_imm, regrd* | And |

**Opcode**

ADD ADDcc ADDX ADDXcc

AND ANDcc ANDN ANDNcc

|  |
| --- |
|  |
| *Table 5-3*  SPARC to Assembly Language Mapping *(Continued)*  *5*  **Comments**  branch never synonym: bnz  synonym: bz  synonym:  bgeu synonym: blu  synonym: b  branch never   |  |  |  | | --- | --- | --- | | **Mnemonic** | **Argument List** | **Operation** | | bn{,a}  bne{,a} be{,a} bg{,a} ble{,a} bge{,a} bl{,a} bgu{,a} bleu{,a} bcc{,a}  bcs{,a} bpos{,a} bneg{,a} bvc{,a} bvs{,a} ba{,a} | label  label label label label label label label label label  label label label label label label | Branch on integer condition codes | | call | label | Call subprogram | | cbn{,a} cb3{,a} cb2{,a} cb23{,a} cb1{,a} cb13{,eo} cb12{,a} cb123{,a} cb0{,a} cb03{,a} cb02{,a} cb023{,a} cb01{,a} cb013{,a} cb012{,a} cba{,a} | label label label label label label label label label label label label label label label label | Branch on coprocessor condition codes |   **Opcode**  BN  BNE BE BG BLE BGE BI BGU BLEU BCC  BCS BPOS BNEG BVC BVS BA  CALL CBccc  *Instruction-Set Mapping*  33 |
|  |
| *5*  **Opcode**  FBN FBU FBG FBUG FBL FBUL FBLG FBNE  FBE FBUE FBGE FBUGE FBLE FBULE FBO FBA  FLUSH  JMPL  LDSB LDSH LDSTUB  LDUB LDUH  LD LDD  LDF LDFSR  LDDF  LDC LDCSR LDDC  *Table 5-3*  SPARC to Assembly Language Mapping *(Continued)*   |  |  |  | | --- | --- | --- | | **Mnemonic** | **Argument List** | **Operation** | | fbn{,a} fbu{,a} fbg{,a} fbug{,a} fbl{,a} fbul{,a} fblg{,a} fbne{,a}  fbe{,a} fbue{,a} fbge{,a} fbuge{,a} fble{,a} fbule{,a} fbo{,a} fba{,a} | label label label label label label label label  label label label label label label label label | Branch on floating-point condition codes | | flush | *address* | *Instruction cache flush* | | jmpl | *address*, *reg*rd | Jump and link | | ldsb ldsh ldstub  ldub lduh  ld  ldd  ld ld  ldd  ld ld ldd | *[address], reg*rd *[address], reg*rd *[address], reg*rd  *[address], reg*rd *[address], reg*rd  *[address], reg*rd *[address], reg*rd  *[address], freg*rd *[address], %*fsr  *[address], freg*rd  *[address], creg*rd *[address], %csr [address], creg*rd | Load signed byte Load signed halfword Load-store unsigned byte  Load unsigned byte Load unsigned halfword  Load word Load double word  Load floating-point register  Load double floating-point Load coprocessor  Load double coprocessor |   **Comments**  branch never  synonym:  fbnz synonym: fbz  *regrd* must be even  *fregrd* must be even  34 *SPARC Assembly Language Reference Manual—November 1995* |

|  |
| --- |
|  |
| *Table 5-3*  SPARC to Assembly Language Mapping *(Continued)*  *5*  **Comments**  *regrd* must be even   |  |  |  | | --- | --- | --- | | **Mnemonic** | **Argument List** | **Operation** | | ldsba  ldsha lduba lduha lda ldda | *[regaddr]asi, reg*rd  *[regaddr]asi, reg*rd *[regaddr]asi, reg*rd *[regaddr]asi, reg*rd *[regaddr]asi, reg*rd *[regaddr]asi, reg*rd | Load signed byte from alternate space | | ldstuba | *[regaddr]asi, reg*rd |  | | mulscc | *reg*rs1*, reg\_or\_imm, reg*rd | Multiply step (and modify icc) | | nop |  | No operation | | or orcc orn orncc | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Inclusive or | | rd rd  rd rd rd | %asr*n*rs1*, reg*rd %y*, reg*rd  %psr*, reg*rd %wim*, reg*rd %tbr*, reg*rd |  | | restore | *reg*rs1*, reg\_or\_imm, reg* rd |  | | rett | *address* | Return from trap | | save | *reg*rs1*, reg\_or\_imm, reg*rd |  | | sdiv sdivcc | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Signed divide Signed divide and modify icc |   **Opcode**  LDSBA  LDSHA LDUBA LDUHA LDA LDDA  LDSTUBA MULScc NOP  OR ORcc ORN ORNcc  RDASR RDY  RDPSR RDWIM RDTBR  RESTORE  RETT SAVE  SDIV SDIVcc  See synthetic instructions. See synthetic instructions. See synthetic instructions. See synthetic instructions.  See synthetic instructions.  See synthetic instructions.  *Instruction-Set Mapping*  35 |
|  |
| *5*  **Opcode**  SMUL SMULcc  SETHI  SLL SRL SRA  STB  STH  ST STD STF STDF STFSR  STDFQ  STC STDC STCSR STDCQ  STBA  STHA  STA STDA  *Table 5-3*  SPARC to Assembly Language Mapping *(Continued)*   |  |  |  | | --- | --- | --- | | **Mnemonic** | **Argument List** | **Operation** | | smul smulcc | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Signed multiply Signed multiply and modify icc | | sethi sethi | *const22, reg*rd %hi(*value*)*, reg*rd | Set high 22 bits of register | | sll srl sra | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Shift left logical Shift right logical Shift right arithmetic | | stb  sth  st std st std st  std | *reg*rd*,* [*address reg*rd*,* [*address*]  *reg*rd*, [address] reg*rd*, [address] freg*rd*, [address] freg*rd*, [address]* %fsr*, [address]*  %fq*, [address]* | Store byte Store half-word  Store floating-point status register Store double floating-point queue | | st std st std | *creg*rd*, [address]*  *creg*rd*, [address]* %csr*, [address]* %cq*, [address]* | Store coprocessor  Store double coprocessor | | stba  stha  sta stda | *reg*rd *[regaddr]asi reg*rd *[regaddr]asi*  *reg*rd*, [regaddr]asi reg*rd*, [regaddr]asi* | Store byte into alternate space |   **Comments**  See synthetic instructions.  Synonyms: stub, stsb  Synonyms: stuh, stsh  *regrd* Must be even  *fregrd* Must be even  *cregrd* Must be even  Synonyms: stuba, stsba Synonyms: stuha, stsha  *regrd* Must be even  36 *SPARC Assembly Language Reference Manual—November 1995* |

|  |
| --- |
|  |
| *Table 5-3*  SPARC to Assembly Language Mapping *(Continued)*  *5*  **Comments**  Trap never Synonym: tnz  Synonym: tz  Synonym: tcs Synonym: tcc  Synonym: t   |  |  |  | | --- | --- | --- | | **Mnemonic** | **Argument List** | **Operation** | | sub subcc subx subxcc | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Subtract Subtract and modify icc Subtract with carry | | swap swapa | *[address], reg*rd *[regaddr]asi, reg*rd | Swap memory word with register | | tn tne  te tg tle tge tl tgu tleu  tlu  tgeu  tpos tneg tvc tvs ta | software\_trap\_number software\_trap\_number  software\_trap\_number software\_trap\_number software\_trap\_number software\_trap\_number software\_trap\_number software\_trap\_number software\_trap\_number  software\_trap\_number software\_trap\_number  software\_trap\_number software\_trap\_number software\_trap\_number software\_trap\_number software\_trap\_number | Trap on integer condition code  **Note:** Trap numbers 16-31 are reserved for the user. Currently- defined trap numbers are those defined in /usr/include/sys/trap.h | | taddcc  tsubcc taddcctv  tsubcctv | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd  *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Tagged add and modify icc  Tagged add and modify icc and trap on overflow |   **Opcode**  SUB SUBcc SUBX SUBXcc  SWAP SWAPA  Ticc  TADDcc  TSUBcc TADDccTV  TSUBccTV  *Instruction-Set Mapping*  37 |
|  |
| *5*  **Opcode**  UDIV UDIVcc  UMUL UMULcc  UNIMP  WRASR WRY  WRPSR WRWIM WRTBR  XNOR XNORcc  XOR XORcc  *Table 5-3*  SPARC to Assembly Language Mapping *(Continued)*   |  |  |  | | --- | --- | --- | | **Mnemonic** | **Argument List** | **Operation** | | udiv udivcc | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Unsigned divide Unsigned divide and modify icc | | umul | *reg*rs1*, reg\_or\_imm, reg*rd | Unsigned multiply | | umulcc | *reg*rs1*, reg\_or\_imm, reg*rd | Unsigned multiply and modify  icc | | unimp | const22 | Illegal instruction | | wr wr  wr wr wr | *reg\_or\_imm,* %asr*n*rs1 *reg*rs1*, reg\_or\_imm,* %y *reg*rs1*, reg\_or\_imm,* %psr  *reg*rs1*, reg\_or\_imm,* %wim *reg*rs1*, reg\_or\_imm,* %tbr |  | | xnor xnorcc | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Exclusive nor | | xor xorcc | *reg*rs1*, reg\_or\_imm, reg*rd *reg*rs1*, reg\_or\_imm, reg*rd | Exclusive or |   **Comments**  See synthetic instructions See synthetic instructions See synthetic instructions See synthetic instructions  38 *SPARC Assembly Language Reference Manual—November 1995* |

*5*

***Floating-Point Instruction***

Table 5-4 shows floating-point instructions. In cases where more than numeric type is involved, each instruction in a group is described; otherwise, only the first member of a group is described.

*Table 5-4*

Floating-Point Instructions

|  |  |
| --- | --- |
| **Mnemonic\*** | **Argument List** |
| fitos fitod fitoq | *freg*rs2*, freg*rd *freg*rs2*, freg*rd *freg*rs2*, freg*rd |
| fstoi fdtoi fqtoi | *freg*rs2*, freg*rd *freg*rs2*, freg*rd *freg*rs2*, freg*rd |
| fstod fstoq | *freg*rs2*, freg*rd *freg*rs2*, freg*rd |
| fdtos fdtoq | *freg*rs2*, freg*rd *freg*rs2*, freg*rd |
| fqtod fqtos | *freg*rs2*, freg*rd *freg*rs2*, freg*rd |
| fmovs fnegs fabss | *freg*rs2*, freg*rd *freg*rs2*, freg*rd *freg*rs2*, freg*rd |
| fsqrts fsqrtd fsqrtq | *freg*rs2*, freg*rd *freg*rs2*, freg*rd *freg*rs2*, freg*rd |
| fadds faddd faddq | *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd |

**SPARC**

FiTOs FiTOd FiTOq

FsTOi FdTOi FqTOi

FsTOd FsTOq

FdTOs FdTOq

FqTOd FqTOs

FMOVs FNEGs FABSs

FSQRTs FSQRTd FSQRTq

FADDs FADDd FADDq

**Description**

Convert integer to single Convert integer to double Convert integer to quad

Convert single to integer Convert double to integer Convert quad to integer

Convert single to double Convert single to quad

Convert double to single Convert double to quad

Convert quad to double Convert quad to single

Move  
Negate Absolute value

Square root

Add

\* Types of Operands are denoted by the following lower-case letters: i integer  
s single  
d double

q quad

*Instruction-Set Mapping* 39

*5*

*Table 5-4*

Floating-Point Instructions *(Continued)* **Description**

Subtract

Multiply

Multiply double to quad Multiply single to double

Divide

Compare

Compare, generate exception if not ordered

|  |  |
| --- | --- |
| **Mnemonic\*** | **Argument List** |
| fsubs fsubd fsubq | *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd |
| fmuls fmuld fmulq | *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd |
| fmulq fsmuld | *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd |
| fdivs fdivd fdivq | *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd *freg*rs1*, freg*rs2*, freg*rd |
| fcmps fcmpd fcmpq fcmpes  fcmped fcmpeq | *freg*rs1*, freg*rs2 *freg*rs1*, freg*rs2 *freg*rs1*, freg*rs2 *freg*rs1*, freg*rs2  *freg*rs1*, freg*rs2 *freg*rs1*, freg*rs2 |

**SPARC**

FSUBs FSUBd FSUBq

FMULs FMULd FMULq

FdMULq FsMULd

FDIVs FDIVd FDIVq

FCMPs FCMPd FCMPq FCMPEs

FCMPEd FCMPEq

\* Types of Operands are denoted by the following lower-case letters: i integer  
s single  
d double

q quad

*Coprocessor Instructions*

All *coprocessor-operate* (cpop*n*) instructions take all operands from and return all results to coprocessor registers. The data types supported by the coprocessor are coprocessor-dependent. Operand alignment is also coprocessor-dependent. Coprocessor-operate instructions are described in Table 5-5.

If the EC (PSR\_enable\_coprocessor) field of the processor state register (PSR) is 0, or if a coprocessor is not present, a cpop*n* instruction causes a *cp\_disabled* trap.

40 *SPARC Assembly Language Reference Manual—November 1995*

|  |
| --- |
|  |
| *5*  The conditions that cause a *cp\_exception* trap are coprocessor-dependent. *Table 5-5* Coprocessor-Operate Instructions  **SPARC Mnemonic**   1. CPop1  cpop1 2. CPop2  cpop2   *Synthetic Instructions*  **Synthetic Instruction**  btst bset bclr btog  call  clr clrb clrh clr  cmp  dec dec deccc  deccc  **Argument List**  *opc, reg*rs1*, reg*rs2*, reg*rd *opc, reg*rs1*, reg*rs2*, reg*rd  **Name Comments**  Coprocessor operation Coprocessor operation May modify ccc  Table 5-6 describes the mapping of synthetic instructions to hardware instructions.  *Table 5-6*  Synthetic Instruction to Hardware Instruction Mapping  **Comment**  Bit test Bit set Bit clear Bit toggle  Clear (zero) register Clear byte Clear halfword Clear word  Compare  *reg\_or\_imm, reg*rs1 *reg\_or\_imm, reg*rd *reg\_or\_imm, reg*rd *reg\_or\_imm, reg*rd  reg\_or\_imm  *reg*rd [*address*] [*address*] [*address*]  reg, reg\_or\_imm  *reg*rd *const13, reg*rd *reg*rd  *const13, reg*rd  andcc or andn xor  jmpl  or stb sth st  subcc  sub sub subcc  subcc  *reg*rs1*, reg\_or\_imm,* %g0 *reg*rd*, reg\_or\_imm, reg*rd *reg*rd*, reg\_or\_imm, reg*rd *reg*rd*, reg\_or\_imm, reg*rd  *reg\_or\_imm,* %o7  %g0*,* %g0*, reg*rd %g0*,* [*address*] %g0*,* [*address*] %g0*,* [*address*]  *reg*rs1*, reg\_or\_imm,* %g0  *reg*rd*, 1, reg*rd *reg*rd*, const13, reg*rd *reg*rd*, 1, reg*rd  *reg*rd*, const13, reg*rd  Decrement by 1 Decrement by *const13* Decrement by 1 and set icc Decrement by *const13* and set icc  *Instruction-Set Mapping*  41  **Hardware Equivalent(s)** |
|  |
| *5*  **Synthetic Instruction**  inc inc inccc  inccc  jmp  mov mov mov mov mov mov mov mov mov  not not neg neg  restore save  set set set  *Table 5-6*  Synthetic Instruction to Hardware Instruction Mapping *(Continued)* **Comment**  Increment by 1 Increment by *const13* Increment by 1 and set icc Increment by *const13* and set icc  *reg*rd *const13, reg*rd *reg*rd  *const13, reg*rd  *address*  *reg\_or\_imm,reg*rd %y*, reg*rs1 %psr*, reg*rs1 %wim*, reg*rs1 %tbr*, reg*rs1 *reg\_or\_imm,* %y *reg\_or\_imm,* %psr *reg\_or\_imm,* %wim *reg\_or\_imm,* %tbr  **Hardware Equivalent(s)**  add add addcc  addcc  jmpl  *reg*rd*, 1, reg*rd *reg*rd*, const13, reg*rd *reg*rd*, 1, reg*rd  *reg*rd*, const13, reg*rd  *address,* %g0  %g0*, reg\_or\_imm, reg*rd %y*, reg*rs1 %psr*, reg*rs1 %wim*, reg*rs1  %tbr*, reg*rs1 %g0*,reg\_or\_imm,*%y %g0*,reg\_or\_imm,*%psr %g0*,reg\_or\_imm,*%wim %g0*,reg\_or\_imm,*%tbr  One's complement One's complement Two's complement Two's complement  Trivial *restore*  Trivial *save trivial save* should only be used in supervisor code!  if -4096 ≤ *value* ≤ 4095 if ((*value* & 0x3ff) == 0) otherwise Do not use the set synthetic instruction in an instruction delay slot.  42  *SPARC Assembly Language Reference Manual—November 1995*  *reg*rs1*, reg*rd regrd *reg*rs1*, reg*rd *reg*rd  *value,reg*rd *value,reg*rd *value,reg*rd  or rd rd rd rd wr wr wr wr  xnor xnor sub sub  restore  save  or sethi sethi or  *reg*rs1*,* %g0*, reg*rd regrd*,* %g0*, reg*rd %g0*, reg*rs2*, reg*rd %g0*, reg*rd*, reg*rd  %g0*,* %g0*,* %g0  %g0*,* %g0*,* %g0  %g0*, value, reg*rd %hi*(value), reg*rd %hi*(value), reg*rd*; reg*rd*,* %lo*(value), reg*rd |

|  |
| --- |
|  |
| *5*  *Table 5-6*  Synthetic Instruction to Hardware Instruction Mapping *(Continued)* **Comment**  if *z* is set, ignores next instruction if *z* is not set, ignores next instruction  test  **Synthetic Instruction**  skipz skipnz  tst  *reg*  *Instruction-Set Mapping*  43  **Hardware Equivalent(s)**  bnz,a .+8 bz,a .+8  orcc  *reg*rs1*,* %g0*,* %g0 |
|  |
| *5*  44  *SPARC Assembly Language Reference Manual—November 1995* |

***Pseudo-Operations***

The pseudo-operations listed in this appendix are supported by the SPARC assembler.

***Alphabetized Listing with Descriptions***

.alias  
Turns off the effect of the preceding .noalias pseudo-op.

(Compiler-generated only.)

.align boundary  
Aligns the location counter on a boundary where ((“location counter”

modboundary)==0); *boundary* may be any power of 2. .ascii string [, string"]

Generates the given sequence(s) of ASCII characters.

.asciz string [, string]\*

Generates the given sequence(s) of ASCII characters. This pseudo-op appends a null (zero) byte to each *string.*

.byte 8bitval [, 8bitval]\*

Generates (a sequence of) initialized bytes in the current segment.

45

|  |
| --- |
|  |
| *A*  46  *SPARC Assembly Language Reference Manual—November 1995*  .common symbol, size [, sect\_name] [, alignment]  Provides a tentative definition of *symbol*. *Size* bytes are allocated for the object represented by *symbol*.   * **If the symbol is not defined in the input file and is declared to be *local* to the file, the symbol is allocated in *sect\_name* and its location is optionally aligned to a multiple of *alignment*. If *sect\_name* is not given, the symbol is allocated in the uninitialized data section (*bss*). Currently, only .bss is supported for the section name. (.data is not currently supported.)** * **If the symbol is not defined in the input file and is declared to be *global*, the SPARC link editor allocates storage for the symbol, depending on the definition of *symbol\_name* in other files. Global is the default binding for common symbols.** * **If the symbol is defined in the input file, the definition specifies the location of the symbol and the tentative definition is overridden.**   **.double 0rfloatval [, 0rfloatval]\***  **Generates (a sequence of) initialized double-precision floating-point values in the current segment. *floatval* is a string acceptable to atof(3); that is, an optional sign followed by a non-empty string of digits with optional decimal point and optional exponent.**  **.empty**  **Suppresses assembler complaints about the next instruction presence in a delay slot when used in the delay slot of a Control Transfer Instruction (CTI).**  **Some instructions should not be in the delay slot of a CTI. See the S*PARC Architecture Manual* for details.**  **.file string**  **Creates a symbol table entry where *string* is the symbol name and STT\_FILE is the symbol table type. *string* specifies the name of the source file associated with the object file.** |

A

.global symbol [, symbol]\* .globl symbol [, symbol]\*

Declares each symbol in the list to be global; that is, each symbol is either defined externally or defined in the input file and accessible in other files; default bindings for the symbol are overridden.

* A global symbol definition in one file will satisfy an undefined reference to the same global symbol in another file.
* Multiple definitions of a defined global symbol is not allowed. If a defined global symbol has more than one definition, an error will occur.
* A global psuedo-op oes not need to occur before a definition, or tentative definition, of the specified symbol.

Note – This pseudo-op by itself does not define the symbol. .half 16bitval [, 16bitval]\*

Generates (a sequence of) initialized halfwords in the current segment. The location counter must already be aligned on a halfword boundary (use .align 2).

.ident string

Generates the null terminated string in a comment section. This operation is equivalent to:

.pushsection .comment .asciz string .popsection

.local symbol [, symbol]\*

Declares each symbol in the list to be local; that is, each symbol is defined in the input file and not accessible in other files; default bindings for the symbol are overridde

1. .local symbol [, symbol]\*

Declares each symbol in the list to be local; that is, each symbol is defined in the input file and not accessible in other files; default bindings for the symbol are overridden. These symbols take precedence over weak and global symbols.

Since local symbols are not accessible to other files, local symbols of the same name may exist in multiple files.

.pushsection .comment .asciz string .popsection

Pseudo-Operations 47

|  |
| --- |
|  |
| A  Note – This pseudo-op by itself does not define the symbol. .noalias %reg1, %reg2  %reg1 and %reg2 will not alias each other (that is, point to the same destination) until a .alias pseudo-op is issued. (Compiler-generated only.)  .nonvolatile  Defines the end of a block of instruction. The instructions in the block may not be permuted. This pseudo-op has no effect if:  • The block of instruction has been previously terminated by a Control Transfer Instruction (CTI) or a label  • There is no preceding .volatile pseudo-op .optim string  This pseudo-op changes the optimization level of a particular function. (Compiler-generated only.)  .popsection  Removes the top section from the section stack. The new section on the top of the stack becomes the current section. This pseudo-op and its corresponding .pushsection command allow you to switch back and forth between the named sections.  .proc n  Signals the beginning of a procedure (that is, a unit of optimization) to the peephole optimizer in the SPARC assembler; n specifies which registers will contain the return value upon return from the procedure. (Compiler- generated only.)  .pushsection sect\_name [, attributes]  Moves the named section to the top of the section stack. This new top section then becomes the current section. This pseudo-op and its corresponding .popsection command allow you to switch back and forth between the named sections.  48  SPARC Assembly Language Reference Manual—November 1995 |

A

.quad 0rfloatval [, 0rfloatval]\*

Generates (a sequence of) initialized quad-precision floating-point values in the current segment. floatval is a string acceptable to atof(3); that is, an optional sign followed by a non-empty string of digits with optional decimal point and optional exponent.

Note – The .quad command currently generates quad-precision values with only double-precision significance.

.reserve symbol, size [, sect\_name [, alignment]]

Defines symbol, and reserves size bytes of space for it in the sect\_name. This operation is equivalent to:

If a section is not specified, space is reserved in the current segment.

.pushsection

.align symbol:

.skip .popsection

sect\_name alignment

size

.section section\_name [, attributes]

Makes the specified section the current section.

The assembler maintains a section stack which is manipulated by the section control directives. The current section is

is the section that is currently on top of the stack. This pseudo-op changes the top of the section stack.

• If section\_name does not exist, a new section with the specified name and attributes is created.

.pushsection

.align symbol:

.skip .popsection

sect\_name alignment

size

Pseudo-Operations 49

A

• If section\_name is a non-reserved section, attributes must be included the first time it is specified by the .section directive.

See the sections “Predefined User Sections” and “Predefined Non-User Sections” in Chapter 3, “Executable and Linking Format,” for a detailed description of the reserved sections. See Table 3-2 in Chapter 3, “Executable and Linking Format,” for a detailed description of the section attribute flags.

Attributes can be:

#write | #alloc | #execinstr

Changes the current section to one of the predefined user sections. The assembler will interpret the following SunOS 4.1 SPARC assembly directive:

.seg text, .seg data, .seg data1, .seg bss,

to be the same as the following SunOS 5.x SPARC assembly directive:

.section .text, .section .data, .section .data1, .section .bss.

Predefined user section names are changed in SunOS 5.x. .single 0rfloatval [, 0rfloatval]\*

Generates (a sequence of) initialized single-precision floating-point values in the current segment.

Note – This operation does not align automatically.

#write | #alloc | #execinstr

.seg text, .seg data, .seg data1, .seg bss,

.section .text, .section .data, .section .data1, .section .bss.

50

SPARC Assembly Language Reference Manual—November 1995

A

.size symbol, expr

Declares the symbol size to be expr. expr must be an absolute expression. .skip n

Increments the location counter by n, which allocates n bytes of empty space in the current segment.

.stabn <various parameters>

The pseudo-op is used by Solaris 2.x SPARCompilers only to pass debugging information to the symbolic debuggers.

.stabs <various parameters>

The pseudo-op is used by Solaris 2.x SPARCompilers only to pass debugging information to the symbolic debuggers.

.type symbol, type

Declares the type of symbol, where type can be:

#object #function #no\_type

.uaword 32bitval [, 32bitval]\*

Generates a (sequence of) 32-bit value(s).

.version string

Identifies the minimum assembler version necessary to assemble the input file. You can use this pseudo-op to ensure assembler-compiler compatibility. If string indicates a newer version of the assembler than this version of the assembler, a fatal error message is displayed and the SPARC assembler exits.

.volatile

Defines the beginning of a block of instruction. The instructions in the section may not be changed. The block of instruction should end at a .nonvolatile pseudo-op and should not contain any Control Transfer Instructions (CTI) or labels. The volatile block of instructions is terminated after the last instruction preceding a CTI or label.

.weak symbol [, symbol]

Declares each symbol in the list to be defined either externally, or in the input file and accessible to other files; default bindings of the symbol are overridden by this directive.

Note the following:

* A weak symbol definition in one file will satisfy an undefined reference to a global symbol of the same name in another file.
* Unresolved weak symbols have a default value of zero; the link editor does not resolve these symbols.
* If a weak symbol has the same name as a defined global symbol, the weak symbol is ignored and no error results.

Note – This pseudo-op does not itself define the symbol. .word 32bitval [, 32bitval]\*

Generates (a sequence of) initialized words in the current segment.

1. .xstabs <various parameters>

The pseudo-op is used by Solaris 2.x SPARCompilers only to pass debugging information to the symbolic debuggers.

symbol =expr

Assigns the value of expr to symbol.

[bytecode \_

# Integrated Design Patterns

***Creational Patterns***

**Abstract Factory –** Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

String->Object->Abstract->Walls

#public abstract class <Walls>

[public concreteWall getWall() //returns a concrete class

return

]

[public brickWall getWall() //returns a concrete class

return

]

]

end class Walls

String->Object->AbstractFactoryExample

#public class <AbstractFactoryExample>

[public void example

<Walls> abstractFactoryWalls = new <Walls>()

//get wall

Wall newWall = abstractFactoryWalls.getWall()

]

end class AbstractFactoryExample

//Clients only have to commit to an interface defined by a class

//not a particular concrete class

**Builder-** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

String->Object->RTFReader

#Class RTFReader

[public void ParseRTF

<TextConverter> myTextConverter = new <TextConverter>

//get text conversion object

<TeXConverter> converter = myTextConverter.getTeXText();

]

end class RTFReader

//top level

String->Object->TextConverter

#public class <TextConverter>

public void get

[ get new ASCIIConverter ]

[ get new TeXText ]

[ get new TextWidgetController ]

end class TextConverter

//subclass of TextConverter

String->Object->TextConverter->ASCIIConverter

#public class <ASCIIConverter>

public void get

[get ASCIIConverter ]

Return ASCIIText

End class ASCIIConverterd

//sublass fo TextConverter

String->Object->TextConverter->TeXConverter

#public class <TeX Converter >

public void get

[get TeXConverter]

Return TeXText

End class TeXConverter

//subclass of TextConverter

String->Object->TextConverter->TextWidgetController

#public class <TextWidgetController>

public void get

[get TextWidget Controlller>

Return TextWidget

Endclass TextWidgetController

**Factory Method** – Define an interface for creating an object, but let subclasses choose which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

String->Object->BombedMazeGame->MazeGame

#public class MazeGame

public bombedMazeGame()

public makeWall()

[ return new BombedWall ]

public makeRoom(int n)

[ return new RoomWithABomb(n) ]

end class MazeGame

//enchanted example

String->Object->EnchantedMazeGame->MazeGame

#public class MazeGame

public enchantedMazeGame()

public makeRoom(int n)

[ return new EnchantedRoom(n,castSpell, ()) ]

public makeDoor(Room1 r1, Room r2

[ return new DoorNeedingSpell(r1, r2) ]

protected Spell castSpell

endclass MazeGame

**Prototype –** Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.

String->Object->Client

#Class Tool

[manipulate()]

endclass Toool

String->Object->Client ->Tool->RotateTool

#Class RotateTool

[manipulate()]

endlcass RotateTool

String->Objecft->Client->Tool->GraphicTool

#Class GraphicTool

[manipulate()]

endclass GraphicTool

//(Prototype) graphic provides an *interface* for cloning itself

String->Object->Client->Graphic

#Class Graphic

[drawPosition]

[clone]

end Graphic

String->Object->Client->Graphic->Staff

#Class Staff

[drawPosition]

[clone]

endclass Staff

//concrete prototype implements an *operation* for cloning itself

String->Object->Client->Graphic->MusicalNote

#Class MusicalNote

[ DrawPosition]

[ Clone]

endclass MusicalNote

String->Object->Client

#Class Client

//a client creates a new object by asking a prototype to clone itself

[prototype.clone()]

endclass Client

**Wrapper**

String->Object->PenguinsClass

#public class PenguinsClass

[ printPenguinsGame(int time, int endscore, String teams[]) ]

endclass PenguinsClass

String->Object->WrapperClass

#public class WrapperClass

public <PenguinsClass> penguinsClass = new< PenguinsClass>()

[public printHockeyGame(int time, int endscore, String teams[]

{ penguinsClass.printPenguinsGame(time, endscore, teams) }

]

**Adapter –** Convert the interface of a class into another interface clients would expect. An adapter takes two classes and lets them work together when they typically wouldn’t be able to. Acts as the essential “middle-man.”

String->Object->Client //entry point

#class Client

endclass Client

//one side of the adapter

String->Object->Target

#public class Target

[Request()] ///called by adapter

endclass Target

//adapter takes request from target, converts it to specific request for the adaptee //and to send back to adapter

//to send out new format

String\_>Object->Adapter

#public classs Adapter

//takes out adaptee implement results and sends out specific request

[Request()]

endclass Adapter

(implemetation of the adapter)

String->Object->Adaptee

#public class Adaptee

[SpecificRequest()] //

endclass Adaptee

Essentially Target wants to execute some commands but doesn’t know how to communicate with Adaptee. Adapter serves as a middle man. Target communicates its request to adapter, and adapter convert that into a request that adaptee can understand. This process occurs vice versa.

**Bridge –** Decouple an abstraction from its implementation so that the two can vary independently.

Platform depenent code and abstractions should be kept apart. In the Bridge pattern, the implementation and its abstraction are connected only by a single connection, known as the Bridge.

//left of bridge

String->Object->Window

#public class Window

[DrawText()]

[DrawRect()]

Imp.DevDrawLine()

Imp.DevDrawLine()

Imp.DevDrawLine()

Imp.DevDrawLine()

endclass

String->Object->Window->IconWindow

#public class IconWindow

[DrawBorder()

[DrawRect()]

[DrawText()]

]

endclass

String->Object->Window->TransientWindow

#public class TransientWindow

[DrawCloseBox()

[DrawRect()]

]

endclass

String->Object->WindowImp

#public class WindowImp

[DevDrawText()]

[DevDrawLine()]

endclass WindowImp

String->Object->WindowIMP->XWindowIMP

XWindowIMP  
 [DevDrawText()]

[DevDrawLine()

[XDrawLine()]

[XdrawString()]

]

String->Object->PMWindowIMP->PMWindowIMP

#public class PMWindowIMP

[DevDrawText]

[DevDrawLine]

*Window* exposes an interface to its subclass heiarchy by exposing drawTex() and drawRect(). WindowIMP (the other side of the bridge) is the implementation of the abstract/interface of Window.

**Composite** – Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composite objects uniformly.

Composite layers of hiearchies take classes and forms of trees of classes. It is an alternative to inheritance.

String->Object->AComposite

#public class Acomposite

public <Acomposite> [] aComposite = new <Acomposite>()

public <Leaf> [] leaf = new <Leaf>()

endclass

**Decorator** – Attach additional responsibilites dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Suppose I have a window without scroll bars and I’d like to add scroll bars to the window when the time arrives. A decorator should be able to attach itself to any of the window classes, or moreover, there should be functionality to add othe rthings like background pictures or animated images time bars.

String->Object->TextView

#Public class TextView

[AttachDecorator(DecoratorInterface dif)]

Endclass

String->Object->DecoratorInterface->ScrollBarInterface

#public interface ScrollBarInterface

[ funcitonality() ]

Endclass

**Façade –** Provide a unified interface to a set of interfaces in a subsystems.. Façade defines a higher level interface that makes the subsystem easier to use.

String->Object->Apples

#Public class Apples

endclass Apples

String->Object\_>Oranges

#public class Oranges

endclass Oranges

String->Objec t->Pears

#public class Pears

endclass Pears

String->Object->Watermelon

#public class Watermelon

endclass Watermelon

String->Object->HoneyDew

#public class HoneyDew

endclass HoneyDew

String->Object->Bananas

Public class Bananas

End class Bananas

String->Object->Cantelope

Public class Cantelope

End class Cantelope

becomes

//Fruit which connects to all of the prior

//This way all of the subclasses can be reached

//through one interface

String->Object->Interface->Fruit

//example

String->Object->Fruit->Apples

#public class Apples

endclass Apples

String->Object->Fruit->HoneyDew

#public class HoneyDew

endclass HoneyDew

String->Object->Fruit->Bananas

#Public class Bananas

endclass Bananas

String->Object->Fruit->Cantelope

#public class Cantelope

endclass Cantelope

**Flyweight –** Use sharing to support large numbers of fine-grained objects efficiently.

**Proxy** – Provide a surrogate or placeholder or or surrogate for another object controll access to it.

**Reactor**

***Behavioral Patterns***

**Chain of Responsibility** – Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

handlerObject = retrieveFromEntityPool(“request string goes here”)

**Command –** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable requests.

command “command text string goes here”

**Interpreter –** Given a language , define a representation for its grammar, along with an interpreter that uses the representation to to interpret sentences in its language.

(XML XSD)

**Iterator –** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

This is already defined.

**Mediator** – Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referringt o each other explicitly, and it lets you vary their interaction independently.

Instead of:  
String->Object->myFirstClass

String->Object->Interface->myFirstInterface

String->Object->myAbstractClass

String->Object->implementAbstractClass

Etc..

Mediate mediator{ myFirstClass, myFirstInterface, AbstracftClass, implementAbstractClass }

Public class myFIrstClass

[testExample print “ TestRun” ]

Mediator.myFirstClass.print();

**Memento –** Without violating encapsulation., capture and externalize an object’s internal state so that this object’s state can be reinstated later.

store(“signature phrase”)

retrieve(“signature phrase”)

**Observer –** Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated as well.

observer publisher myClass ::dependent1 ::dependent2 ::dependent3

observer subscriber dependent1 [ public void ….]

**State** – Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

state publisher myClass :->bobsClass

**Strategy –** Define a family of algorithms, encapsulate each one, , and make them interchangeable. Strategy alllows the algorithm to vary indendently from clients that use it.

Set up a type hiearchy as thus:

String->Object->Strategy

String->Object->Strategy->ClassAlgorithmOne

String->Object->Strategy->ClassAlgorithmTwo

String->Object0>Strategy->ClassAlgorithmThree

**(Skeleton) Template Method –** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

Already implemented

**Visitor –**Represent an operation to be performed on the elements of an object

Structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

int +operator(int rhs, int lhs) { return rhs+lhs }

class of elements being operated on: Set class.

String->Object->+operator->AbstractVisitor->Integer

Public MyIntegerClass

{…}

and

String->Object->+operator->Float

Public MyFloatClass

{…..}

Now I can do this:

MyIntegerClass myIntegerClass

MyFloatClass myFloatClass

**Actor Pattern**

An actor is a computational entity that, in response to a message it receives, can concurrently:

* send a finite number of messages to other actors;
* create a finite number of new actors;

# Code Organization

**Multiple Code** is assemble into a box. A box is like a package in Java.

**Parallel Project code** is cross-source code boxes (multiple projects) in the compiler. So one project can reference another project. The projects don’t have to be compiled, you can have multiple projects made up only of source.

**Parallel Cannon –** Compiled to the “assembly” level. Multiple cannons can exist and work with each other without special commands or configurations. Similar to a jar in java.

**Kernel Compiled** – compiled one or more boxes into a library. A kernel compiled works at the device driver/kernel level. A kernel compiled is compiled fully into assembly language.

# Miscellaneous Designs

**Singleton –** Instantiation of a class to one and only one object.

Public Singleton Class

**Flux –** Given an objecct denoted with the keyword Flux, it can take on any type.

Instantiation: flux MyClass myObject

Change the Type: flux MyOtherClass myObject

**Anti-Imperative**

CASE is an “anti-imperative” language, What this means is that the code for CASE doesn’t revolve around step by step interusctions, and it doesn’t revole around recursion like functional languages. CASE is entirely StateFul. An instantiateed class in CASE is an object similar to C or Java, however it is considred “live” that is, it has a specific state. One object is tied via code to another object. The state of that one object is now bound to that of the other. Example:

Public Class CoffeeMug

Int a = “Coffee Mugs” //multiple coffee mugs

Public Class Coffee

**Stateful** CofffeMug.a = Int a = “Thirsty Drinkers” **Tie** (operator)

In the above, **stateful** ties the state of the Coffee Mugs to Int a of class coffee. Now whenever CoffeeMug is changed, a correspondign change occurs in Int a of Class Coffee. The  **statement Tie** followed byan operator (in this case a generic ‘operator’)

, the same operator, that is always called whenever a change occures in Coffee Mugs. Here is another example:

Public Class Video Game Contest WInner

expertInMarioBros;

Public Class VideoGameContest

Contestants [ **stateful** Bob, Joe, Jason, Jack ] **tie** expertInMarioBros ( equiv. Bob)

Above there are four stateful contestants and one winner. According to the **tie-in** Bob is set to work, kicking backwards to the **stateful** keyword array where Bob state is sent to expertInMarioBros.

**Equiv statement**

As seen above in the anti-imperative statements, **equiv is like an assignment statement,** however equiv only takes one operand, not two. Equiv is used when the context is clearly know, just as above where we undertsand tie in experitinMarioBros is going to be the winner and we know that winnner is going to be Bob. So the first, implicit, operand is expertInMarioBros and the second, given via **equiv**, is Bob.

**A more simpler example of Stateful language**

When the number of apples increases, I want the number of oranges to decrease.

Public class Apples

Int apples = 0;

Public class Oranges

**Stateful** Apples .apples = Int Oranges = “Orange Lovers” **Tie** (equiv ++)

If the following syntax holds true, when apples changes (decreases or increases) , the variable in class Oranges, Int Oranges, tied by the tied keyword, is assigned (because of the equiv keyword is assinging to the variable to t he left, ,in this case Int Oranges) increases. So essentially, when the number of apples increases, the number of oranges increases (by ++).

**Spawn**

Automatic handling of threading and concurrency in general. Like how on a function call the current code is pushed on the stack, and a new execution begins, CASE uses spawning. That is, essentially, CASE approaching a large function will automaticallty spawn a thread and “push the function” onto a new thread. CASE’s process manager will automatically handle the fractured execution to prevent deadlock, etc.

# Dependency Injection

**Setter Injection**

This method requires the client to provide a [setter method](https://en.wikipedia.org/wiki/Setter_method) for each dependency.

*// Setter method*

**public** void setService(Service service) {

*// Save the reference to the passed-in service inside this client*

**this**.service = service;

}

**Interface Injection**  
This is simply the client publishing a role interface to the setter methods of the client's dependencies. It can be used to establish how the injector should talk to the client when injecting dependencies.

*// Service setter interface.*

**public** **interface** **ServiceSetter** {

**public** void setService(Service service);

}

*// Client class*

**public** **class** **Client** **implements** ServiceSetter {

*// Internal reference to the service used by this client.*

**private** Service service;

*// Set the service that this client is to use.*

@Override

**public** void setService(Service service) {

**this**.service = service;

}

}

**Constructor injection**

This method requires the client to provide a parameter in a [constructor](https://en.wikipedia.org/wiki/Constructor_(object-oriented_programming)) for the dependency.

*// Constructor*

Client(Service service) {

*// Save the reference to the passed-in service inside this client*

**this**.service = service;

}

**Dependency Injection:**

|  |  |
| --- | --- |
|  | Basically, instead of having your objects creating a dependency or asking a factory object to make one for them, you pass the needed dependencies in to the constructor or via property setters, and you make it somebody else's problem (an object further up the dependency graph, or a dependency injector that builds the dependency graph). A dependency as I'm using it here is any other object the current object needs to hold a reference to.  One of the major advantages of dependency injection is that it can make testing lots easier. Suppose you have an object which in its constructor does something like:  public SomeClass() {  myObject = Factory.getObject();  }  This can be troublesome when all you want to do is run some unit tests on SomeClass, especially if myObject is something that does complex disk or network access. So now you're looking at mocking myObject but also somehow intercepting the factory call. Hard. Instead, pass the object in as an argument to the constructor. Now you've moved the problem elsewhere, but testing can become lots easier. Just make a dummy myObject and pass that in. The constructor would now look a bit like:  public SomeClass (MyClass myObject) {  this.myObject = myObject;  }  Most people can probably work out the other problems that might arise when not using dependency injection while testing (like classes that do too much work in their constructors etc.) Most of this is stuff I picked up on the [Google Testing Blog](http://googletesting.blogspot.com/), to be perfectly honest... |

**HTML Class**

ORM

**MVC – Model –View- Controller-**

A typical collaboration of the MVC components

The central component of MVC, the *model*, captures the behavior of the application in terms of its [problem domain](http://en.wikipedia.org/wiki/Problem_domain), independent of the user interface.[5] The model directly manages the data, logic and rules of the application. A *view* can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. The third part, the *controller*, accepts input and converts it to commands for the model or view.[6]

**Interactions**[[edit](http://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&action=edit&section=3)]

In addition to dividing the application into three kinds of components, the model–view–controller design defines the interactions between them.[7]

* A **controller** can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).
* A **model** stores data that is retrieved by the controller and displayed in the view. Whenever there is a change to the data it is updated by the controller.
* A **view** requests information from the model that it uses to generate an output representation to the user.

**Controller** – Designate a class as a controller

(Controller ties the view and model togethr)

Public class (View::myHTML, View ::myOtherHTML, Model::myORMClass, )

**View –** Designates a view for the controller

Public class(Controller::myController)

**Model –** Designates a locates a data repository.

Public class(Controller::myController)

**Targets can be constructed with events fired to them to trigger snippets of code.**

# Standard Collections Library

**SortedSet**

**SortedMap**

**HashSet**

**TreeSet**

**AbstractCollection**

**Priority Queue**

**PriorityBlocking Queue**

**Map**

Map is a smarter map. Instead of the obfuscated Map class of Java, we use the standard CASE Collections Type whose syntax is similar to an array.

<Map> <String> < String><String> myMap = <Map><String><String><String>

<> because it is a Type.

From here, the Map is accesed just like an array

myMap[1][1][1] = myMap[2][3][4];

Or by key and value

String coordinates = myMap[myLocation][myTime];

**HashMap**

**MultiMap**

# Diagram

Diagrams are used to link one schemata to another schemata making code reuse very easy.

An example of a pipeline is:

<int:ctr>

An example of a diagram is:

<classname: record>

<int:ctr>  
<String: name>  
</classname:recor>

If the above two pipelines are connected (int:ctr) whenever one is updated the other is updated to reflect it.

i.e.,

<int:ctr>| <int:ctr>

<int:ctr + 1> | <int:ctr + 1>

# Boolean Operations

It doesn’t make sense to use symbols for boolean operations and only mkes a difficult process of logic more difficult

|  |  |
| --- | --- |
| True |  |
| False |  |
| Not |  |
| IsTrue |  |
| IsFalse |  |
| And |  |
| Or |  |
| Xor |  |
| NotOr | Prefixing a word with Not negates it |
| NotAnd |  |
| Like | Pattern Comparison Operator |
| & | Bitwise Operator |
| | | Bitwise Operator |
| \| | Bitwise XOR |
| == | Boolean Equals? |
| Also | Often times in C/C++/Java you have to test two boolean expressions at a time.  This results in a mess of operators that is frustrating to read .Using Also….  If (h==5 also n>5) replaces something like: if ((h==5) && (n>5)) |
|  |  |

|  |  |
| --- | --- |
| boolean isLetter(char ch)  boolean isDigit(char ch) | Determines whether the specified char value is a letter or a digit, respectively. |
| boolean isWhitespace(char ch) | Determines whether the specified char value is white space. |
| boolean isUpperCase(char ch)  boolean isLowerCase(char ch) | Determines whether the specified char value is uppercase or lowercase, respectively. |
| char toUpperCase(char ch)  char toLowerCase(char ch) | Returns the uppercase or lowercase form of the specified char value. |
| toString(char ch) |  |

# Regular Expressions

The LIKE operator is used to match a string against a pattern or vice-versa.

  If (testString Like "[A-Z]#") Then

matches a capital letter followed by a digit. Uses standard regular erxpression standards.

# Exotic Operations

**Unpacking**

a,b,c = 1,2,3

>> a,b,c

(1,2,3)

**Unpaking for swapping variables**

>>a,b = 1,2

>>a,b = b,a

(2,1)

**Extended unpacking**

>>a, \*b, c = [1,2,3,4,5]

>>a

1

>>b

[2,3,4]

>>c

5

**Negative Indexing**

>>a = [0,1,2,3,4,5,6,7,8,9,10]

>>at[-1]

10

>>at [-3]

8

**List slices**

>>a = [0,1,2,3,4,5,6,7,8,9,10]

>>a[2:8]

[2,3,4,5,6,7]

**List slices with negative indexing**

>>a = [0,1,2,3,4,5,6,7,8,9,10]

>>a [::2]

[0, 2, 4, 6, 8, 10]

>>a[::3]

[0,3,6,9]

>>at[2:8:2]

[2,4,6]

**List Slices with negative step**

>>a = [0,1,2,3,4,5,6,7,8,9,10]

>>at[::-1]

[10,9,8,7,65,4,3,2,1,0]

>>a[::-2]

[10,8,6,4,2,0]

**List Slice Assignment**

>>a= [1,2,3,4,5]

>>a[2:3] = [0, 0]

>>a

[1,2,0,0,4,5]

>>a[1;1] = [8,9]

>>a

[1,8,9,2,0,0,4,5]

>>a[1:-1]= []

>>a

[1,5]

<http://sahandsaba.com/thirty-python-language-features-and-tricks-you-may-not-know.html>

http://stackoverflow.com/questions/101268/hidden-features-of-python

https://docs.python.org/2/library/stdtypes.html

http://www.cplusplus.com/reference/cstdio/scanf/

# Example Code

**Java:**

[**Two Dimensional String Array Example in Java**](http://java.sampleexamples.com/java-examples/two-dimensional-string-array-example-in-java/)

It is a 2D array nested loop example. Two loops require, first for row and second for column. It is a String 2D Array example.

**Two Dimensional String Array Example in Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | **import** java.util.Scanner;    **public** **class** NestedStringArray2DExample  {  **public** **static** **void** main(String[] args)      {          Scanner in=**new** Scanner(System.in);          System.out.println("Enter the number of rows");  **int** noOfRows=in.nextInt();          System.out.println("Enter the number of columns");  **int** noOfColumns=in.nextInt();          String [][] names=**new** String[noOfRows][noOfColumns];          System.out.println("Enter String for array: ");          //reading elements from user  **for**(**int** rows=0;rows<names.length;rows++)          {  **for**(**int** cols=0;cols<names[rows].length;cols++)              {                  names[rows][cols]=in.next();              }          }          //displaying elements          System.out.println("2D String array elements are: ");  **for**(**int** rows=0;rows<names.length;rows++)          {  **for**(**int** cols=0;cols<names[rows].length;cols++)              {                  System.out.print(names[rows][cols]+"      ");              }              System.out.println();          }      }  } |

**CASE:**

# Compiler Feature

Auto-save

Correct Indentation

Generate Code

Search for called (if your code calls a function, you can look at that function and find it.

Plugin Abilities

Visual Code for debugging – stacks look like stacks, trees like trees

Drag and drop files into IDE

Facilitate adding of code from other languages (using native keyword)

Match braces, brackets, paranthesis, and quotes

Bookmarks

Breakpoints

Built in debugger

Project Explorer for multiple projects (Boxes, Parallel Projects, and Cannons)

Easy to debug. Verbose errors. Indicate the errors location in code, the contents on the stack (in english) and suggestrions on how to fix the issue. It shows a stack trace and a listing of variables in use at each step of the ttrace. Shows type signature of offending code and type signature of functions called in the stack trace.

Uses “the boot.” That is, it takes a trace of code , keeping a reord for x number of routines in the program and upon reaching a bug, shows visually (like in the visual languages) the code and routines that led up to that point.

Git support built in

Refactoring

Toggle Comments

UnToggle Comments that works!

Automated API Generation (with suggestions by the compiler)  
Auto-Generate (for example, imports or properties)

Cross-File Search

Find, Replace

Cut, Copy, Paste

Build

Build All

Clean

Run

Auto-Plugin (search like Eclipse – automatically installs a plugin from a library)

Auto-git (automatically does a backup, like autosave in Word)

Difference in files (shows difference in two or more soure files)  
Verify (verifies certain language things are implemented – like a main function

A main class, libraries import statements

Auto-complete – fills in the rest of keywords

Smart-fill – compiler predicts upcoming code

Self-Documenting – Takes the code and creates an API documentation using NLP. To ammend the documentation, use the tags: /\*\*\*\*\*\*\*\*\*\*\*/

Split screens as many times as you like

Move around panels

If you need to import a library to use a class the IDE will suggest it

If you need to make a correction to your code the IDE will suggest it

The Compiler has a built in comparison “diff” function for comparing files

Its annoying to dig through stack traces but its also annoying to have a one-line

error message. So CASE does both. A thorough stack trace for exceptions in ENGLISH, and a quick summary of the error. CASE is designed so that your stack trae should read like a good book. Using nouns , adjectives, verbs, etc…

Tooltips – Hovering over a word/statement/keyword displays a small box that is the API entry . IF you hover over an operator it tells you what the operator doies and if you hover over a word like ‘for’ for example, it will tell you what it does. If you hover over one of your classes or objects it either shows your JAumentation or the definition/declaration, with a magnifying glass. Clicking the glass shows you where it is you.

HelpTips – As you code tips popup like the tooltips, suggesting possible code. Such as if you type “Class myclass” it will suggest “= new” or if you type public it will add “void” as in “public void class” or when you use the increment operator, it will fill it in for you.

OftenTips – If you have a certain pattern, such as typing “System.out.println” the ompiler will use that information to give you suggestions. For example, if you have a habit of typing “Int.parseInt(10)” it will fill in “parseInt(“

Search by function name, by class name, by primitive

Lines of code change color, underland, turn to italics, etc depending on errors

Use a + and a – to collapse scope

*dynamic* language, which means that some parts of our code can be extended and modified while our apps are running (i.e. after they’ve already been compiled).

Use the **Dynamic** keyword.

**Runtime**

The language is a hybrid. It partially compiles its code into byte-code, but it is bytecode that is generated on the fly, so that it can be used as a scripting language as well.

**On the Road: Tablet and Google Drive**

CASE has a tablet/phone/tablet comptuer/computer application that is the front end of the language. That is, it is an app called Mobile CASE. You can run the app, say your Apple iPad. The code is hosted on Google Drive. (see Google Drive- integrated). The app then is compiled by CASE on the app and run on VMWare’s Virtual Machine. **NOW CAN RUN ENTIRE APPLICATION ON YOUR BROWSER**

Application sits on the cloud. Hard-wired cloud language

# CASE Application Standard Library

DSL Library – (Domain-Specific Language) support such as regex, XML, or javascript

Math Library

Network Library (better version of JDBC)

Database Library

Graphics Library

IO Library

GUI Library

File Library

Web Serivces Library

# HEAL

Heal is an infrastructure within the CASE programming language that works like Chaos Monkeys. IT WILL BREAK YOUR CODE DURING RUNTIME. It is your job to repair/relaunch the broken service in your code.

# POLYGLOT

The ability to use multiple languages together. For example, in CASE, a *bowtie* can be used to act as a bridge from one language into CASE. Plans are, that CASE will have at least several bowties allowing a multittude of foreign languages to be covnerted to CASE or code CASE projects with bowtie code embedded with the CASE code.

# AUTOMATIC SCALABILITY

Automated Language Built in Scalability (Netflix Superbowl Example. Getting hit all at once) The language will reduce its number of microservices when input is low and scaled it up when input is heavy. Microservices communite with each other using SOA like messaging.

# AUTO-DOC

Similar to Java Docs. AUTO-DOC automatically generates an API in HTML as well

as a UML diagram for the application.

DOCUMENTATION IS VERY VERY VERY IMPORTANT

Package

Class Name

Implemented Interfaces

Direct Known Subclasses

Detailed Description of Class

Nested Classes

Fields

Constructors

Methods

Class Members

To manually add to the AUTO-DOC before generation do the following:

%% Description for below goes here %%

For example:

int test;

%%Counter Variable%%

int cftr;

Documentation written now in six or seven years may be as clear as mud. The solution is AUTO-DOC, that will update the documentation and designs automatically – just by running it over again. There is only one problem and that was the description of the class section of the documentation. This literal data (contained in the percentage signs) has to be automatically updated. This must be implemented.

Part of the AUTO-DOC for percentage desfriptions are the IDE Tagged automated documenter. Designed to make design simple easy and quick. AUTO-DOC tags, similar to the percent description, indicate a description. However, AUTO-DOC tags represent a description in CODE form, so that when functionality changes, so does AUTO-DOC tags, changing the documentation when generated and is essentially a programmatic version of the natural language descriptions.

# Techniques to prevent software from aging

Several important aspects contriubute to preventing software againg and becoming obsolete. The first is upkeep. CASE will report when a section of code is signficantly old, deprecated, or simply out of use.

Documentation is important as well. AUTO-DOC tracks changes to documenation indicating changes to the actual software. AUTO-DOC will report as well if code is signficantly old, deprecase, or out of use.

# Integrated Database Implementation

***TODO:***

LOOK INTO SCALA

**DATABASE INTERFACE \_> Just like Highmark, our Database Structures are way to complex, obtuse, large, and difficult to maintain and manage. CASE removes this complication somehow and turns its DB code directly into SQL for the Database essentially by a wave of the hand. Database should be easily readable and understandable in its architecture by its architectect.**

**Increase Cohesion, Reduce Coupling, end up with something more manageble.**

**LOOK AT PYTHON’s Standard API (their libraries)**

**Assume Failure**

We can pretty much assume that at some point the system being developed is going to fail. During compile time, CASE compiler generates a series of tests that is applies, intentionally trying to break the program being created – in the spirit of Netflixes chaos monkeys. If the app does fail, a report sort of like a stacktrace is printed out.

**In the Spirit of Design**

A sort of divide and conquer idea goes on and is built into the functionality of CASE. High level architecture such as client/server is built on top of individual objects which contain state and behavior, which takes the form of algorithms and code. CASE comes with a tool installed called the “Architect” which takes in information and automatically creates a hiearchy of UML, then after a little tweaking on the design creates an object and their connections, then finally recommends algorithms to solve the problem.