5-2005

Ensight

Jonathan Hefner

Ensight

by

Jonathan Hefner

Undergraduate honors thesis under the direction of

Dr. Donald Kraft

Department of Computer Science

Submitted to the LSU Honors College in partial fulfillment of
the Upper Division Honors Program.

May 2005

Louisiana State University
& Agricultural and Mechanical College
Baton Rouge, Louisiana

Ensight

by

Jonathan Hefner

Undergraduate honors thesis under the direction of

Dr. Donald Kraft

Department of Computer Science

Submitted to the LSU Honors College in partial fulfillment of
the Upper Division Honors Program.

May 2005

Louisiana State University
& Agricultural and Mechanical College
Baton Rouge, Louisiana

Ensight

Jonathan Hefner

Introduction

This paper is written to partially fulfill the requirements for the Honors College's Upper Division Honors program. It is the culmination of two semesters of research in programming language design and philosophy. The purpose of this paper is to serve as a description and proposal for a new language of my own design, called Ensight.

The key concepts of Ensight are laid down in this paper. In the final section of the paper, issues for future development are enumerated. These issues branch out into more sophisticated realms such as parallel computing, web services, and Aspect Oriented Programming. During my research period I was able to get a feel for these issues, but was not able to thoroughly delve into them. Thus, I plan to support them in Ensight at a later date.

The format of this paper is different than a typical research paper. In each section of the paper, the main points of the section are presented in bulleted format. Following this bulleted outline are written paragraphs which describe the reasoning of certain design decisions. This format was chosen because of the context of this paper. This format was chosen to reflect the dual nature of this paper; namely 1) to serve as a crash course in the Ensight language, and 2) to present the knowledge of the topic acquired through my research.

As a final note, the syntax presented in this paper for certain features of Ensight is not always final. I am fairly picky and wishy-washy about syntax; during the course of my research I changed my mind several times over on the syntax for numerous Ensight constructs. I can only assume that I will continue to do so as I continue to develop Ensight (it just seems to be an occupational hazard).

Overall Philosophy

Ensight's target audience is a programmer with a basic amount of experience in imperative and object oriented programming (for

instance, who has taken his intro to programming course and his object oriented programming course). Ensight's goal for its target audience is to be the friendliest language the programmer can choose for all parts of the development process. Often times languages sell themselves based on one aspect of software development. PHP and Python, for example, sell themselves on quick prototyping with weak typing (run-time type checking), while Java and C++ sell themselves on designing large, robust, industrial strength software systems. Ensight tries to incorporate the best (and friendliest) features of languages such as C++, Java, PHP, and Python while keeping more in mind than just one aspect of the programmer's task.

By A Developer, For A Developer

There is something to be said for obfuscated C programs (or Ruby scripts, etc.) that accomplish amazing things in just under 15 lines of code. Even so, I have always preferred the 30-line variants of these programs, where each code block was clearly marked by curly brackets on their own lines and each variable was given a duly long and descriptive name--though, I hate to admit all these things, for fear that my 'leet programming skills' will be questioned.

Ensight doesn't do anything so extreme as to force you to make everything a class; furthermore, it has code-hacker-appealing features such as treating functions as first-class variables and allowing for Python-style code blocking. However, pointer based arithmetic isn't anywhere to be found, nor is ASCII-based arithmetic, and 3-letter abbreviated keywords are rare. Thus I expect Ensight, at least at first, will have more of a chance appealing to the software developer rather than the script hacker.

The Kitchen Sink Syndrome: Don't Try To Be Everything To Everyone

Despite trying to appease both camps of programmers (script hackers and software system developers), I was careful to be selective with the features I included in Ensight. In fact, an equally important design goal was to have as simple and small a language core as possible. At first, it was tempting to incorporate every nifty language feature I came across into Ensight. I thought, "Then Ensight could be everything anyone would ever want." I realized very quickly, though, that this is exactly the opposite of what should happen when designing a language. Otherwise, you get stuck with an PL/1 clone--and nobody wants that. I have learned, in fact, that the best thing one can do when designing a programming language is to cut as many features as you can without detracting from the languages power or simplicity.

Basic Syntax

Ensight draws most of its basic syntax from C, in the same way that C++ or Java does. The wide adoption of C (as well as C++ and Java) makes C a logical choice as a parent language if I want Ensight to appeal to the largest possible audience. Additionally, C is my native tongue, and C++ and Java are personal favorites. Where improvements could be made, though, I did not hesitate to abandon C's way of doing things.

Code Blocking and End-of-Line (EOL) Markers

- Ensight, by default, uses carriage returns for end of line (EOL) markers and tabs for code blocks.
- Curly braces ("{ }") override tab-based code blocking. All code blocked off by curly braces must use semicolons (";") for end of line markers and curly braces for any subsequent (child) code blocks.
- **Example:** If-else statement using both styles of code blocking.

```
if x != 0
        x = 0
        y = 1
        if z == 0
                a = 10
                b = 20
else
{
        x = 1;
        y = 0;
        if z == 0
        {
                a = 20;
                b = 30;
        }
```

```
}
```

- When using carriage returns as EOL markers, the continuation operator (~) can be used at the beginning of a line of code, marking it as part of the previous line of code.
- **Example:** If-else statement using line continuation.

```
if age >= 21
        permission =
        ~ true
else
        permission
        ~ = false
```

This area of basic syntax is one of the very few areas where I chose to satisfy two opposing camps of language design (C-style block control vs. Python-style block control). This area seemed important enough and basic enough to warrant the inclusions of both styles. Overall, the inclusion of both seemed easy enough to implement, and it seemed silly to turn away any potential users based on such a simple--though not insignificant--design detail.

Comments

- "//" begins single line comments.
- "/*" begins, and "*/" ends multi-line comments
- **Example:** Using single- and multi-line comments.

```
x = 5    // This is a single-line comment
y = 10   /* This is a
                        multi-line
                        comment */
```

Single- and multi-line comments exactly like their counterparts in C++ and Java. I felt there was no need or reason to choose a different style, as the this style works and is already widely accepted.

Modules

- Each source file is considered can be a module.
- The module name must be given at the top of the file. If no name is given, the contents of the source file is considered to be in the default module.
- Module names can contain a virtual path to the module. This virtual path is essentially just a namespace that the module is placed in.
- Each module (source file) can contain multiple classes and functions.
- Each module (source file) can contain one `main()` function. If a module is built into an executable by the compiler, this `main()` function will be called when the executable file is run.
- **Example:** Module Definition

```
#module MyCompany.MyPackage.MyModule
//      |------------------|--------|
//      |This is the virtual|This is |
//      |path               |the     |
//      |                   |module  |
//      |                   |name    |
//      |------------------|--------|

function main()
{
        // function definition will be covered
        // in a later section...
}
```

Two-way Selection Structure

- Two-way selection is done with an `if-else` statement similar to in C.
- Unlike C, the logic test for the `if` statement does not need to be enclosed in parenthesis. However, parenthesis can be added without changing its meaning, in the same way that the statements `x = 5` and `x = (5)` are equivalent.
- Similar to Java, the logic test for the `if` statement must return a boolean value (`true` or `false`). That is, tests which return anything other than a boolean type while cause a compile error.
- When chaining `if-else` statements an `elseif` is used.
- **Example:** A chained if-else expression.

```
if x > 5
        z = 10
elseif x > 10
        z = 20
elseif x > 20
        z = 40
else
        z = 100
```

- Each `if`, `elseif`, and `else` statement is considered a code block label. That is, each of the above keywords along with their associated logical test (if applicable) must be on a separate physical line than the executable code block they refer to, regardless of the code blocking method being used.
- Because `if`, `elseif`, and `else` statements are considered code block labels, they must be properly blocked when nested inside other selection structures.
- **Example:** Invalid and Invalid if statements.

```
// Case 1 -- does not compile
if x > 0
{
        if x > 5
                z = 10;
        else if x > 10  // Invalid -- must be on a separate line
                z = 20;
}

// Case 2 -- does not compile
if x > 0
{
        if x > 5
                z = 10;
        else
                if y > 5 // Invalid -- multiline statements (including
                        z = 40; // labels) must be properly enclosed in {}'s
}

// Case 3 -- compiles
if x > 0
{
        if x > 5
                z = 10;
        else
        {
                if y > 5
                        z = 40;
        }
}
```

The inclusion of `elseif` and the mandate that `if`, `elseif`, and `else` must be on their own lines is to prevent the possibility of creating ambiguous code. For example, given the indentation in the following code:

```
if (x > 0) if (x > 5)  // Invalid -- will give compile error
        z = 10
        y = 20
```

Should the statement `y = 20` be associated with the first `if` clause or the second? Clearly `z = 10` is meant to be for the second, but really, it should be further indented by one more tab. So if we go ahead and assume `z = 10`'s association despite improper indentation, what should we assume for `y = 20`'s association?

Additionally forcing proper blocking for selection constructs prevents ambiguous coding errors like the following:

```
// Case 1 -- will not compile
// In C, compiles to the exact same as Case 2
if (x > 5)
        if (y > 5)
                z = 10;
else
        z = 20;

// Case 2 -- will not compile
// In C, compiles to the exact same as Case 1
if (x > 5)
        if (y > 5)
                z = 10;
        else
                z = 20;
```

For these two cases, the programmer obviously had different intentions. In C the compiler disregards his intentions in one of the two cases; in Ensight, the compiler forces him to properly specify his intentions for each case.

Multiway Selection Structures

- The multiway selection structure uses the `switch` keyword, similar to C.
- Cases in the `switch` construct are denoted with the `case` keyword, similar to C.
- The `case` keyword can be omitted before a particular case to cause a simple boolean test against the qualifier for that case (rather than a comparison between the qualifier and the target of the `switch`). This allows for a `switch` to be used as a substitute for a chain of if-else-if-else-etc.
- If the `case` keyword is omitted for every case, then a target for the `switch` does not have to be specified.
- The case code in a `switch` is specified as a code block, rather than C's method of using a colon (`:`).
- The default behavior of the `switch` statement is to exit after one case is executed. To override this behavior and have a case "fall through" to the next after execution, the `continue` keyword may be specified. (This is the opposite of the behavior of the C, C++, and Java `switch` construct).
- The `default` keyword can be used to mark a case which should be executed if no other matching cases are found.
- Similar to the two-way selection structure, the parenthesis enclosing the target of the `switch` statement may be omitted. However, parenthesis can be added without altering the meaning of the expression. The same is also true for the parameters for each case.
- **Example:** Switch statements

```
switch (name)
{
        case "hefner"
        {
                print "author";
        }

        case "kraft"
        {
                print "mentor (thank you!)";
        }

        case "baumgartner"
        {
                // the continue statement below will cause the
                // next case to be executed as well
        }
        continue;
```

```
        case "ramanujan"
        {
                print "reviewer (thank you!)";
        }

        default
        {
                print "thank you for reading this!";
        }
}


// The following switch statement behaves
//  just as a three-chain of if-else-if-else-etc
switch  // target is (can be) omitted
        // -- all cases omit the 'case' keyword

        x < 0   // here 'case' is omitted, therefor this is
                // simply evaluated as a logical test
                print "less than"

        x == 0
                print "equal to"

        x > 0
                print "greater than"
```

The behavior of the C, C++, and Java `switch` statement to, by default, fall through every case is counterintuitive by common opinion. The only reason I can think that Java left this behavior in is to conform to tradition.

Repetition Structures

- The following repetition structures are available: `while`, `for`, and `foreach`.
- Each loop structure acts similar to an `if` structure, in that opening keyword and looping statement is considered a code block label. Thus, they must be on a separate physical line than the executable code block they refer to, regardless of the code blocking method being used.
- The `while` and `for` loop are exactly the same as in Java--including the fact that the logical test for iteration only accepts boolean values.
- **Example:** While loop and For loop

```
for (i = 0; i < 10; i++)
        // loop body goes here

i = 0
while (i < 0)
        // loop body goes here
        i++
```

- The `foreach` loop is the same as in C#. It can be invoked on list or set (sets and lists will be discussed later).
- **Example:** Foreach loop

```
foreach (item in myList)
        print item        // element is the i-th member of myList
```

C#'s version of the `foreach` loop is used because it is the most intuitive. PHP's `foreach` was considered; however, C#'s version is easier to understand by simply reading the code aloud. Really, what makes one version easier than the other depends on the naming convention used for collection-type variables. PHP's version is more intuitive using the convention `cat[]` --> "cat array" (note, "cat" is NOT plural). C#'s version is more intuitive using the convention `cats[]` --> "array of cats" (here "cats" is plural).

Variables

Variables are handled just like Java in that every variable is actually a reference to an object. (Classes--as prototypes for objects--are discussed in a later section.) Objects are 'created' with a `new` operator, just as in Java. Also, as in Java, Ensight provides built-in garbage collection, which systematically deallocates any unreferenced objects to prevent memory leaks. Through these features, Ensight provides pointer-like support without the hassle of low level memory management or cumbersome pointer syntax.

Primitives

- The following primitive types are supported: String, Integer, Real, Complex, Boolean.
- The primitive types correspond to the literals that are accepted by the compiler. Note that hex and binary literals are special cases of an integer literal.
- Each primitive type has its own class which is built into the language. These classes are "on the other side of the wall"--they have a low level implementation, and thus programmers do not have access to their inner workings.
- Each time a primitive literal is involved in a statement, the compiler automatically invokes the proper action to handle literal. This action may be to create an instance of a primitive class, or to invoke a property set method of a primitive class.

Originally I planned to have have no primitives in Ensight. Then I realized that one must have some kind of primitive, some common denominator, so that you can at least express parameters for the constructors of more complex classes. This led to the idea of having each expressible literal type be considered a primitive, and having an associated handler class for each primitive.

String Literal Handling

- Strings literals can be enclosed in double quotes (") or single quotes ('). Within a string begun with double quotes, single quotes may be used freely, and vice versa.
- In any string literal, the escape character (\ ) may be used to mark a following single or double quote as a literal character.
- **Example:** Single and Double Quoted Strings

```
print "Hello, it's nice to meet you.  My name is \"Ensight.\"
        // output: Hello, it's nice to meet you.  My name is "Ensight."
print 'Hello, it\'s nice to meet you. My name is "Ensight."'
        // output: Hello, it's nice to meet you.  My name is "Ensight."
```

- Objects that have a string conversion method defined for them can be included directly within a string literal by placing the object name inside curly braces ({}) inside the string literal.
- To make an opening curly brace a literal character (rather than an object variable marker), place an escape character (\ ) before the curly brace. Closing curly braces do not need this, as they only function as object variable markers when preceded by a non-literal opening curly brace.
- **Example:** Object Inclusion In Strings

```
x = 9
y = 10

print "The value of x + y is {x + y}."
        // output: The value of x + y is 19.

print "The value of x + y is \{x + y}."
        // output: The value of x + y is {x + y}.
```

- Note that object inclusion in strings is **compile-time** feature. Thus strings read at run-time from input cannot make use of this feature (any object inclusion syntax will simply be interpreted as literal characters).
- The escape character sequences from C are supported. An example of such a sequence is `\n` which represents a carriage return.
- If an escape sequence is not recognized as a valid escape sequence (this includes an escape character followed by whitespace), it is printed as a literal.
- To make an escape character a literal character, place another escape character before it.
- **Example:** Escape Characters In Strings

```
print "Let's look at the following: \*   \   \\   \\\{5}   \\\\{5}"
        // output: Let's look at the following: \*   \   \   \{5}   \\5
```

Special Values

- Integer, Real and Complex primitives can assume the value `INF` (Infinite) as well as `NINF` (Negative Infinite).
- Integer and Real primitives can assume the value `NaN` (Not a Number).
- **Example:** Infinite and NaN Valued Variables

```
Int x = 1 / 0
print "X is {x}"  // output: X is INF

Int y = -1 / 0
print "Y is {y}"  // output: Y is NINF

Int z = Math.sqrt(-1)
print "Z is {z}"  // output: Z is NaN
```

- Any variable can assume a `NULL` value to indicate it does not reference any existing object.
- **Example:** NULL Valued Variables

```
Int myInt;
print "myInt is {myInt}";  // output: myInt is NULL
```

Support for `NaN`, `INF` and `NINF` was added to aid numerics programmers. Having these values available ensures that no special tricks have to be done when these values are actually encountered.

Operators

- The operators from C--such as +, -, *, /, ++, --, ==, !=, etc.--are supported
- The double dot (`..`) operator is used as the string concatenation operator, rather than Java-style overloading of the + operator.
- **Example:** String Concatenation

```
print "Hello " .. "World!"
        // output: Hello World!
```

- Operators are parsed in a greedy fashion. Thus the expression `----x` will be interpreted as pre-decrement x twice, rather than some combination of pre-decrementing and negating x
- If greedy operator parsing fails to produce a crystal clear interpretation of the programmer's intentions, the compiler will throw an error, rather than fall back on a complex ruleset for operator evaluation.
- The colon (`:`) operator is used to typecast a variable rather than C's method of placing the desired type in parenthesis before the variable.
- **Example:** Typecasting

```
ClassB x = new ClassB()
ClassA:x.fnInClassA() // Calls method on x from ClassA
```

- The double colon (`::`) operator is used perform an object type conversion. This can only be used when the target object has a conversion function defined for the specified target type.
- **Example:** Variable Conversion

```
ClassC y = new ClassC()
print String::y
```

- Leaky assignment statements are not supported. This means that assignment statements do not return a value. Therefore, an expression such as `x = y = 5;` is not valid. Additionally, loops written in the form `while (x = isNotDone())` are also invalid.

The reason for not allowing leaky assignments is that their potential for abuse or misunderstanding is too large. Although they can help keep programs shorter, they are not worth their cost in readability.

Sets

- Sets are Ensight's implementation of records (for example, C's struct), though they are declared inline only.
- A set is a statically sized (no more, no less) collection of objects, not necessarily of the same type.
- To access objects in a set via its string index, use the dot operator (`.`).
- **Example:** Set Declaration

```
(Int age, String name) personA = (25, "joe");        // A set with an integer and a string.

print personA.name
```

- Every object is considered a set (taking the object member variables as set items), thus every variable is a set.
- However, not every set is an object, because object's can have private members which a set cannot. Thus a set cannot simply be assigned (via the assignment operator) to an object.
- To assign values to more than one item in a set, but not the entire set, the ellipses operator (`...`) is used. The same can be done to extract values from more than one item in a set, but not the entire set.
- **Example:** Evaluating Part Of A Set

```
// Assigning to part of a set
(String make, Int year, String color) the_car = ("Ford", 1995, "white");

the_car = ( ... , 2004, ... );

// Extracting part of a set
String the_make;
Int the_year;

(the_make, the_year, ... ) = the_car;
```

- Any set that is composed solely of a single item becomes that item, itself. This has the effect of making `x == (x)` and `(x) == ((x))`.

Lists

- Lists are Ensight's implementation of collections.
- Syntactically, a list is a special case of a set, where all objects are of the same type, and the collection can be dynamically sized.
- Each item in a list has a numerical index and an optional string index.
- To add or access objects in a set via its string index, use the single bracket operator (`[ ]`). This is the preferred method of indexing.
- Any object with a string conversion method can be used with the single bracket operator. The string-version of the object will be taken as the index; thus, primitive integers can be used with the single bracket operator to provide semantics similar to those of arrays in C.
- To add or access objects in a set via its numerical index, use the double bracket operator (`[[ ]]`). This method of indexing should only be used when lower-level manipulation is needed.
- If a list has been declared with a size limit, that limit will be enforced when trying to add any items to the list.
- **Example:** List Declaration

```
// A list of integers with no size limit--items
// can be added. Initially has 5 items.
Int[] grades = (90, 90, 87, 95, 92);

// A list of strings with a size limit of 100--items can
// be added up until item 100.  Initially has no items.
String[100] students;
```

- If an object is added to a set via the single bracket operator it is automatically given a numerical index which is one greater than the previous given numerical index. The same is true for an object is added via either the single or double bracket operator but when no index is specified. If no numerical index was previously given, a numerical index of 0 is assigned.
- **Example:** Adding Items To A Set

```
myList[[0]] = "Hello."      // Numerical index = 0, String index = n/a
myList[[1]] = "How "        // Numerical index = 1, String index = n/a
myList[] = "are " // Numerical index = 2, String index = n/a
myList[[]] = "you "         // Numerical index = 3, String index = n/a
myList[0] = "today"         // Numerical index = 4, String index = "0"
myList['last'] = "?"        // Numerical index = 5, String index = "last"
```

- Lists can have multiple dimensions. This is comparable to multidimensional arrays in C.

Functions

Function Definition

- Every function has three associated sets: a set for input variables, a set for output variables, and a set for in/out variables.
- The input variable set is for parameters to the function which cannot be changed in value. This set can be left empty, but must be shown in the function definition. It declared immediately following the function name.
- The output variable set is for the return variables of the function; using this set, a function can return multiple values. This set can be left empty, in which case it does not need to be specified in the function definition. If it is included, it must be declared somewhere between the input set declaration and the function body, and it must be preceded by the keyword `returns`.
- The in/out variable set is for variables that the function both takes as input and uses for output. These variables are comparable to parameters passed by reference to functions in C. This set can be left empty, in which case it does not need to be specified in the function definition. If it is included, it must be declared somewhere between the input set declaration and the function body, and it must be preceded by the keyword `uses`.
- The variables contained in the input set cannot be altered in any way within the body of the function. This includes placing the variable on the left side of an assignment operator, or in the in/out set of another function call.
- A function returns its output set in the same way it would return a single variable in C.
- A function returns value via its in/out set by simply assigning values to the individual members of the in/out set.
- **Example:** Function Definition

```
// Case 1: Typical add() function definition
function add(Int x, Int y) returns (Int)
{
        return (x + y);
}

// Case 2: add() function using in/out set for
// return-by-reference style functionality
function add(Int x, Int y) uses (Int sum)
{
        sum = x + y;
}
```

- A functions signature--its identification--is given by its name, its input set, and its in/out set.
- More than one function can be given the same name as long as each function has a unique signature. This provides polymorphic (same name, different behavior) definition of functions based on function parameterization.

Having sets to separate input, output and in/out parameters of a function solves several problems. First, it allows functions to return multiple values without unnecessarily exposing a variable's value to a called function. Second, it takes away any ambiguity that might be involved when passing a variable reference to a function. That is, the calling programmer now knows whether the parameter variable can be altered or not.

Making the output of a function into a set also allows for the rules of set value assignment to be used. Therefore, a function can return multiple values using the output set, and the calling code and ignore whichever ones it does not need. This ability to ignore

output values is the reason that a function's output set is not a part of its signature.

Function Calling

- Each function call specifies the signature of the function to be called, as well as the parameter variables to be used with the function.
- Which function to execute is determined by the signature specified in the function call. This is the mechanism that provides polymorphism based on function parameters.
- The input set for a function call is specified directly after the function name, just as in the function definition.
- The in/out set for a function call is specified just as a class would be specified when calling the class's member function. That is, the in/out set is placed before the function name, and a dot operator (.) separates the two.
- **Example:** Function Calling

```
// Case 1: Using the add function from Case 1 in the previous section
Int answer;

answer = add(10, 115);

// Case 2: Using the add function from Case 2 in the previous section
(answer).add(10, 115);     // Because the in/out set of the function consists of a single
answer.add(10, 115);       // variable, these two calls are equivalent.
```

Classes

Class Definition and Instantiation

- Classes definitions begin with the keyword `class`, followed by the class name. The body of the class goes after the class name. This syntax is exactly the same as in Java.
- Class member variables and methods (functions) are defined inside the class body. Normal methods of declaration are used, EXCEPT that each function in the class--each method--cannot declare an in/out set. This is because the class, itself, serves as the function's in/out set.
- Class methods also have the class self-reference keyword `this` available inside their bodies. This keyword is used to explicitly access member variables of the class inside a method body.
- Class member variables and methods are not exposed to calling code by default. To make a member variable or method accessible to outside code, use the keyword `expose` directly preceding the variable or method declaration.
- Each class has a constructor function which has the same name as the class itself.
- Class member variables can have default values assigned to them. These values are assigned upon the use of the `new` operator. Thus, if the constructor is called without the `new` operator (to re-initialize the class without reinstantiating it), these default values are NOT reassigned to the member variables.
- **Example:** Class Definition

```
class MyClass

        Int age = 1
        String name

        MyClass(Int age, Int name)
                this.age = age
                this.name = name

        expose function howOldAreYou()
                if age == 1
                        print "I am 1 year old."
                else
                        print "I am {this.age} years old."

        expose function hello()
                print "Hi.  My name is {this.name}."

        expose function hello(String iAm)
```

```
print "Hello, {iAm}.  My name is {this.name}."
```

Conversion Functions

- Conversion functions are used to convert a class to another type of class.
- They are defined within the body of the class that is to be converted.
- They have no input set, and their output set has only one object of the conversion target type.
- Conversion functions are defined using the keyword `conversion` followed by the conversion target (class) type.
- Conversion functions are always exposed in the public interface and therefore do not need to be preceded by the `expose` keyword.
- **Example:** Conversion Function Definition

```
class Celsius

        Real temperature

        Celsius(Real t)
                temperature = t

        conversion String
                String ret_val

                ret_val = "{temperature} degrees Celsius"

                return ret_val
```

A conversion function with 'String' as its target type is Ensight's answer to Java's `toString()` method. Delegating the conversion operation to a specialized function type, however, also allows the conversion operator to be used with any target type, without any special tricks in function name recognition.

Inheritance

- There are three methods of inheritance, each with its own keyword: `extends`, `implements`, `wraps`.
- Inheritance via `extends` can only be used by classes in the same module.
- Inheritance via both `implements` and `wraps` can be used by any class regardless of what package it is in.
- `extends` can only be used for single inheritance; that is, there can be only one target class to inherit from via the `extends` keyword.
- Both `implements` and `wraps` can be used for multiple inheritance; that is, a class can inherit from multiple class via `implements` as well as via `wraps`.
- `extends`, `implements` and `wraps` can all be used simultaneously.
- Inheriting via `extends` provides the inheriting class with full access to the public and private interface of the target class.
- Inheriting via either `implements` or `wraps` effectively provides the inheriting class with its own internally maintained instance of the target class. The inheriting class then only gets access to the public interface of the target class.
- Inheriting via either `extends` or `wraps` does not provide subtyping. Any inherited methods and member variables are not public by default, and must be exposed with the `expose` keyword. When dealing with methods using the same name and input set, only one method may be exposed per name/input set combination.
- Inheriting via `implements` provides subtyping. Any inherited methods and member variables are public, but are only accessible when the inheriting class is cast to the same type as the target class.
- Inherited methods and member variables can be overridden when inheriting via `implements`.
- Name conflicts inside the inheriting class--when dealing inheriting from multiple targets--are resolved by using the target class and the dot operator (.) as a prefix to the method or member variable in question.
- **Example:** Class Inheritance

```
class MyStandardBaseClass
{
        // ...

        function report_error(String err)
        {
                print err;
```

```
        }
}

class MyClass extends MyStandardBaseClass wraps SomeModule.FileLogger
implements SomeOtherModule.GuiComponent
{
        // ...

        function error(String err)
        {
                FileLogger.report_error(err);                // Resolving a naming conflict
                MyStandardBaseClass.report_error(err);       // by prefixing the function
                                                             // with the class name

                GuiComponent:this.msgBox(err);       // Calling a method from an
                                                     // "implemented" class
        }

        expose FileLogger.ClearLog();        // Exposing a method inherited via "wraps"
}
```

Three different inheritance methods were provided in order to adequately address the primary uses of inheritance: code reuse and subtyping. Problems that plague inheritance, such as the breaking of encapsulation and the naming conflicts of multiple inheritance, come from a "one size fits all" approach to inheritance. By refining the type of inheritance used for a given purpose, these problems become easier to solve (particularly inheritance breaking encapsulation).

The `extends` method is the method most like C++ or Java's own inheritance method; however, Ensight's `extends` provides code reuse only. For code reuse, `wraps` is recommended because it is encapsulation safe (providing access to only the public interface of the target class); however, `extends` is provided for classes in the same module because it is assumed that classes in the same module already share a certain amount of intimacy. For subtyping `implements` is provided; it, too, is encapsulation safe.

Inside the inheriting class, name conflicts are resolved via further qualification of the method or member variable owner. Outside the inheriting class this is highly impractical. However, the limitations of the three different inheritance methods prevent naming conflicts from arising in the inheriting class's public interface. With both `extends` and `wraps`, inherited methods and member variables must be manually exposed. Additionally, the inheriting class is not allowed to manually expose any naming conflicts. With inheriting via `implements`, the inheriting class must first be cast to the type of the target class. Since no naming conflicts can exist in the public interface of the target class, no naming conflicts can exist in the casted version of the inheriting class.

Sources

1. Thoughts on Aspect Oriented Programming [anderson-1] Ken Anderson. 4 pages. (Score: 4 of 10)
2. I Have a Feeling We're Not In Emerald City Anymore [baker97] (1997) Henry Baker. 7 pages. (Score: 1 of 10)
3. A Considered Response To "Nuts To OOP!" [barr-1] Michael Barr. 7 pages. (Score: 3 of 10)
4. Some Insights on the Use of AspectJ and Hyper/J [cgl01] (2001) Christina Chavez, Alessandro Garcia, and Carlos Lucena. 5 pages. (Score: 2 of 10)
5. Subject-Oriented Programming [coggins96] (1996) James Coggins. 8 pages. (Score: 5 of 10)

   > "The vision of implementing a megalibrary containing every imaginable data structure attracts many naive programmers."

6. Best practices for programming in C [dh03] (2003) Shiv Dutta and Gary Hook. 10 pages. (Score: 1 of 10)
7. The D Programming Language Overview [digmars04-1] (2004) Digital Mars. 13 pages. (Score: 8 of 10)

   > " C++ programmers tend to program in particular islands of the language, i.e. getting very proficient using certain features while avoiding other feature sets. While the code is usually portable from compiler to compiler, it can be hard to port it from programmer to programmer."

   > " There's a perception that garbage collection is for lazy, junior programmers. I remember when that was said

about C++, after all, there's nothing in C++ that cannot be done in C, or in assembler for that matter. Garbage collection eliminates the tedious, error prone memory allocation tracking code necessary in C and C++. This not only means much faster development time and lower maintenance costs, but the resulting program frequently runs faster! "

"...compilers will not generate warnings for questionable code. Code will either be acceptable to the compiler or it will not be."

8. Modular Mixin-Based Inheritance for Application Frameworks -- A Review [dt01] (2001) Nicholas Chapman. 5 pages. (Score: 4 of 10)
9. Buzz Words - OOP vs. TOP [findy02-1] (2002) Findy Services. 8 pages. (Score: 7 of 10)

"I have heard OOP programmers complain about finding OOP code from other programmers where inheritance was overused, creating "spaghetti" inheritance. This is the 90's version of "goto" spaghetti coding of the 60's."

"Dan Rahmel, et al., has this to say about inheritance: "Inheritance is a hotly contested topic in the object community because...a single change to a class can have far-reaching effects on all the children defined from that class. (From 'Client/Server Applications with Visual Basic 4, Sams Publishing, 1996.)" ... Auto-propagation of changes can cause just as many problems as the lack of change propagation, if not more."

" The Graduation Problem -- This is when some wimpy collection container, such as arrays (including hash arrays), suddenly need more operations or power beyond their original container. Changes in such requirements are very common in the business world. The programmer then has to rewrite all the array manipulation operations to use a more powerful collection container, such as an SQL database engine."

10. OOP Criticism Part 2 [fj02-2] (2002) Findy Services and B. Jacobs. 19 pages. (Score: 8 of 10)

"For example, take a typical Java statement: data = new DataInputStream(new BufferedInputStream(astream)); ... You may have to even follow a long chain of classes to sufficiently understand the original module or method."

"There is more to simplicity than simply hiding the implementation from the component or class user."

"I once asked a Smalltalk programmer why there were so many Set and Get methods in a sample RDBMS connectivity example. The reply was that he was creating an interface to a database entity, so that the programmer did not have to deal with the "raw" table and SQL data. I thought it odd that he used the word "raw." An SQL/RDBMS interface is already an interface. Thus, he was essentially creating an interface to an interface. In other words, doubling up. There is something wrong here. I am not against abstraction layers, per se; but each layer should contribute something significant to the process. His approach was mostly converting one paradigm into another, not providing any real, additional abstraction benefit or detail hiding."

11. Aspects and OO [fj02-3] (2002) Findy Services and B. Jacobs. 10 pages. (Score: 3 of 10)

"When thinking about an issue, usually there is one primary verb per issue. However, there are often multiple primary nouns: Noun1 -> Bus, Verb -> hit, Noun2 -> Larry. In OO, the designer has to choose between lumping "hit" with either bus or Larry, or perhaps even a Hit class."

12. Object Oriented Myths [fj04-1] (2004) Findy Services and B. Jacobs. 11 pages. (Score: 3 of 10)
13. How To Design A Programming Language [fj04-2] (2004) Findy Services and B. Jacobs. 32 pages. (Score: 9 of 10)

"Rapid development is the primary reason for using a scripting language. This is often achieved by having powerful, concise operations that leverage common needs. Parsing, sorting, searching, and file processing are common examples... Another way to simplify development is simplify the syntax. This is done by avoiding the need for excessive type conversion functions, wrappers, mandatory error handling, etc. It is also achieved by allowing or assuming common defaults."

"Just because a program is easy to write does not mean it is easy to read. Maintenance is an important part of

the programming profession. There is nothing more irritating than inheriting cryptic or poorly designed code to maintain or fix. Maintainability is often considered a weak point in scripting languages."

"Programmers are more heavily rewarded for finishing on time than for producing readable code. This is primarily because meeting deadlines is easier to measure than code readability. Everybody knows when a programmer is late, but few if any know how maintainable their code is."

"There is a saying in the Unix community that one should not prevent idiots from abusing something because it may prevent someone else from making good use of it. In other words, give everyone chain-saws because (hopefully) more people will build useful things than the number who will damage something or someone. However, it is my observation that more programmers are more likely to abuse a language than make good use of it. This is usually because the incentive to finish fast is greater (and easier to measure) than the incentive to make coherent and maintainable systems. Thus, I unfortunately have to disagree with the "Unix Chain-saw" rule."

"In other languages, such as XBase and Python, only assignments can provide implied declarations. Any referenced variable must have been declared or assigned previously, otherwise a run-time error is triggered. I consider this more logical since there is no sense in reading an unassigned variable. The only possible reason I can think of for accepting undeclared references is to reduce run-time halts at the expense of having garbage output."

"It is my opinion that some languages have taken symbols too far. Most symbols almost completely lack "mnemonics". Words and abbreviations serve as better (but not always perfect) memory aides than symbols. However, I also realize that there perhaps should be some compromises due to common traditions. Symbols are best used for common operations that tend to occur in long expressions. A good example is string concatenation."

14. Objects Have Failed [grabriel02] (2002) Richard Gabriel. 3 pages. (Score: 5 of 10)

"[I]ncreasingly our systems will be made up of dozens, hundreds, thousands, or millions of disparate components, partial applications, services, sensors, and actuators on a variety of hardware, written by a variegated set of developers, and it won't be incorrect to say that no one knows how it all works. In the old world, we focussed on efficiency, resource limitations, performance, monolithic programs, standalone systems, single author programs, and mathematical approaches. In the new world we will foreground robustness, flexibility, adaptation, distributed systems, multiple-author programs, and biological metaphors for computing."

"[W]e find that object-oriented languages have succumb to static thinkers who worship perfect planning over runtime adaptability, early decisions over late ones."

"The apparent commercial success of objects and our love affair with business during the past decade have combined to stifle research and exploration of alternative language approaches and paradigms of computing. University and industrial research communities retreated from innovating in programming languages in order to harvest the easy pickings from the OO tree. The business frenzy at the end of the last century blinded researchers to diversity of ideas, and they were into going with what was hot, what was uncontroversial."

15. Being Popular [graham01-1] Paul Graham. 20 pages. (Score: 7 of 10)

"[A new language] has to be good for writing throwaway programs. A throwaway program is a program you write quickly for some limited task: a program to automate some system administration task, or generate test data for a simulation, or convert data from one format to another. The surprising thing about throwaway programs is that, like the "temporary" buildings built at so many American universities during World War II, they often don't get thrown away. Many evolve into real programs, with real features and real users."

"Of course the ultimate in brevity is to have the program already written for you, and merely to call it. And this brings us to what I think will be an increasingly important feature of programming languages: library functions. Perl wins because it has large libraries for manipulating strings. This class of library functions are especially important for throwaway programs, which are often originally written for converting or extracting data."

16. Five Questions about Language Design [graham01-2] (2001) Paul Graham. 9 pages. (Score: 6 of 10)

"If you look at the history of programming languages, a lot of the best ones were languages designed for their own authors to use, and a lot of the worst ones were designed for other people to use."

"And it's not only programs that should be short. The manual should be thin as well. A good part of manuals is taken up with clarifications and reservations and warnings and special cases. If you force yourself to shorten the manual, in the best case you do it by fixing the things in the language that required so much explanation."

"I think a lot of the most exciting new applications that get written in the next twenty years will be Web-based applications, meaning programs that sit on the server and talk to you through a Web browser... Instead of having one or two big releases a year, like desktop software, server-based apps get released as a series of small changes. You may have as many as five or ten releases a day."

"[I]n desktop software there is a big bias toward writing the application in the same language as the operating system... Server-based software blows away this whole model. With server-based software you can use any language you want. Almost nobody understands this yet (especially not managers and venture capitalists)... What this means for us, as people interested in designing programming languages, is that there is now potentially an actual audience for our work."

"More and more we were starting to hear about byte code, which implies to me at least that we feel we have cycles to spare. But I don't think we will, with server-based software. Someone is going to have to pay for the servers that the software runs on, and the number of users they can support per machine will be the divisor of their capital cost... It may turn out that byte code is not a win, in the end. Sun and Microsoft seem to be facing off in a kind of a battle of the byte codes at the moment. But they're doing it because byte code is a convenient place to insert themselves into the process, not because byte code is in itself a good idea."

17. Succinctness Is Power [graham02] (2002) Paul Graham. 9 pages. (Score: 2 of 10)
18. The Hundred-Year Language [graham03] (2003) Paul Graham. 12 pages. (Score: 8 of 10)

"I have found in my long career as a slob that cruft breeds cruft... I have a hunch that the main branches of the [programming language] evolutionary tree pass through the languages that have the smallest, cleanest cores."

"I can already tell you what's going to happen to all those extra cycles that faster hardware is going to give us in the next hundred years. They're nearly all going to be wasted. ... The thought of all this stupendously inefficient software burning up cycles doing the same thing over and over seems kind of gross to me. But I think my intuitions here are wrong. I'm like someone who grew up poor, and can't bear to spend money even for something important, like going to the doctor. ... There's good waste, and bad waste. I'm interested in good waste--the kind where, by spending more, we can get simpler designs."

"Semantically, strings are more or less a subset of lists in which the elements are characters. So why do you need a separate data type? You don't, really. Strings only exist for efficiency. ... Having strings in a language seems to be a case of premature optimization."

"Inefficient software isn't gross. What's gross is a language that makes programmers do needless work. Wasting programmer time is the true inefficiency..."

"How far will this flattening of data structures go? ... Will we get rid of arrays, for example? After all, they're just a subset of hash tables where the keys are vectors of integers. Will we replace hash tables themselves with lists?"

19. Why Arc Isn't Especially Object Oriented [graham-1] Paul Graham. 2 pages. (Score: 4 of 10)

"There is a danger in designing a language based on one's own experience of programming. But it seems more dangerous to put stuff in that you've never needed because it's thought to be a good idea."

20. Nuts to OOP! [niemann99] (1999) Thomas Niemann. 8 pages. (Score: 2 of 10)

"Objects derived from the Standard Template Library have been used by thousands of programmers, and tested millions of times. This fact effectively removes them from consideration when bug-tracing. In a developing system, bugs may exist in the code being examined or any of the user-coded objects that support this code. In other words, I think it's okay to use a vendor-supplied string class, but I suggest you don't try to write one."

21. Encapsulation and Inheritance in Object-Oriented Programming Languages [snyder86] (1986) Alan Snyder. 15 pages. (Score: 7 of 10)

"...we must consider what external interface is provided by a class to its children. This external interface is just as important as the external interface provided to users of the objects, as it serves as a contract between the class and its children, and thus limits the degree to which the designer can safely make changes to the class."

"For example, consider a class that is defined using inheritance. Suppose the designer of that class decides that the same behavior can be implemented more efficiently by writing a completely new implementation, without using the previously inherited class. If the previous use of inheritance was visible to clients, then this reimplementation may require changes to the clients. The ability to safely make changes to the inheritance hierarchy is essential to support the evolution of large systems and long-lived data."

22. A Conversation with Java's Creator, James Gosling [venners01] (2001) Bill Venners. 26 pages. (Score: 8 of 10)

"Bill Venners: ...In one interview, you told this really good story about moving to a new apartment and something about keeping things in boxes.
James Gosling: That's actually a general principle for life that works really well. When you move to a new apartment, don't unpack. Just sort of move in, and as you need things, pull them out of the boxes. After you've been in the apartment for a couple of months, take the boxes -- don't even open them -- and just leave what's in there and throw them out.
Bill Venners: The 'don't even open them' part is important because it's very hard to throw things away once you know what they are.
James Gosling: Right, because if you open them, you say, 'oh, I can't part with that.'"

"James Gosling: I would use an immutable whenever I can.
Bill Venners: Whenever you can, why?
James Gosling: From a strategic point of view, they tend to more often be trouble free. And there are usually things you can do with immutables that you can't do with mutable things, such as cache the result. If you pass a string to a file open method, or if you pass a string to a constructor for a label in a user interface, in some APIs (like in lots of the Windows APIs) you pass in an array of characters. The receiver of that object really has to copy it, because they don't know anything about the storage lifetime of it. And they don't know what's happening to the object, whether it is being changed under their feet. You end up getting almost forced to replicate the object because you don't know whether or not you get to own it."

"Bill Venners: Why are there primitive types in Java? Why wasn't everything just an object?
James Gosling: Totally an efficiency thing. There are all kinds of people who have built systems where ints and that are all objects. There are a variety of ways to do that, and all of them have some pretty serious problems. Some of them are just slow, because they allocate memory for everything."

"[On weakly typed languages...] And so, you can end up with systems that perform just fine, so long as the real work is being done in libraries and what you're doing in the language is stitching things together. But it does mean that the language then becomes something that you can't actually use for everything. You can't use it for writing a string matching algorithm in something like PERL. Without the built-in string matching algorithms, it would be pretty slow. Whereas in Java, the string matching libraries are all written in Java. In the world of mathematics, it's a property generally known as completeness."

23. Josh Bloch on Design [venners02] (2002) Bill Venners. 21 pages. (Score: 7 of 10)

"One extreme programming tenet advocates you write the simplest thing that can solve your problem. That's a fine tenet, but it's easy to misconstrue. The extreme programming proponents don't advocate writing something that will barely work as fast as you can. They don't advise you to forgo any design. They do advocate leaving out the bells, whistles, and features you don't need and add them later, if a real need is demonstrated."

"There are APIs that do let you pass around nulls as an abbreviation for zero length arrays or for an empty string, etc. And those APIs are in a sense bad citizens, because once you mix them with APIs that don't, you're in trouble."

24. Strong versus Weak Typing [venners03] (2003) Bill Venners. 8 pages. (Score: 2 of 10)
25. Freshmans First Language [wiki04] (2004) Various Authors. 11 pages. (Score: 5 of 10)

"You must learn C and assembly. Those are your low-level languages, and they are not debatable (with some flexibility for replacing C with C++). The choice of a high-level language is what's up for grabs because the industry hasn't standardized on problems to solve like they have on the hardware used to solve it. That is, while we all use von Neumann machines, we don't all program n-tier web applications, medical scanners, flight simulators, or wristwatches."