Object Orientation Part 2

C++, EIFFEL, C#

C++

WELL IT'S BETTER THAN C

How C++ Came About

- 1980: Smalltalk and Objective-C
- Both attempt to target the new "personal computer" market with casual user programming.
- Smalltalk-80 was a fully realised OO environment, with strict observation of OO rules.
- Objective-C implemented some Smalltalk-like ideas in C.
  - This resulted in a rather strange hybrid of procedural and oo-like concepts that persists in both Obj-C and C++ to this day.

Objective-C and NeXTSTEP

- Objective-C was the primary language for the NeXTSTEP operating system, born out of BSD unix in the mid-1980s.
- NeXTSTEP had all kinds of cool things.
  - If you like Apple's operating systems, you'll find that most of the cool things in them are from NeXTSTEP
- Apple bought NeXTSTEP (and through it the main implementation of Objective-C), and used the concepts to fuel their OS division for ages.
- Naturally, this proprietary nature was not "cool"

C With Classes

- In 1979, an extremely early attempt at a class based C was built by Bjarne Stroustrup
  - "C with Classes"
- This became the go-to object oriented C when Objective-C became bound to NeXTSTEP.
- Renamed in 1983 to "C++"
- Has always, from the very beginning, been an OO supporting language, *not* an OO language.

The Power Of C++

- C++ is a language based around a simple premise:
  - Give the programmer as much power and as much control as possible.
- Highly engineering focused
- Caters to all possible programmer needs/styles
- Allows the programmer to violate it's own rules if the programmer explicitly asks.
- Is one of the most powerful languages in use today.

But why C?

- Everyone likes C (or did in the 1970s)
- C++ is a superset of C
  - These days, modern C is more of a subset of C++
- Pointers are useful, and can form the basis of object orientation.
- Allows for seamless integration of high level OO behaviors and low level assembly like processes, *into the same program!*

What does C++ support?

- You can use procedural programming in C++
  - Design algorithms
  - Share interfaces to those algorithms
- You can use modular programming in C++
  - Design modular algorithms
  - Share modules for code reuse
  - Share interfaces for those modules
- You can use object oriented programming in C++
  - Design objects and behaviors
  - Share objects for code reuse
  - Share services for those objects
- You can do all of these *at the same time!*

C++ Types

- The classic C style types:
  - Int, char, float, double, void
    - You can also use short, long, unsigned, and signed modifiers

C++ Types

- Can define variables as stack based:
  - int x = 0
    - Puts an int x into the stack and sets it as zero

C++ Types

- Can define variables as heap based:
  - int *y;
  - y = new int;
  - *y = 1;
    - This sets y as an integer pointer pointing to an integer somewhere in the heap, and sets that integer to 1. Note the use of “new” instead of “malloc”. You can free the int with:
  - delete y;

C++ Types

- Can define variables as external
  - extern int z;
    - C++ will not allocate memory for z, assuming it will be done somewhere else.

C++ Operators

- Arithmetic:
  - +
  - -
  - *
  - /
  - %
  - ++
  - --

C++ Operators

- Assignment
  - =
  - +=
  - -=
  - *=
  - \=
  - %=

C++ Operators

- Comparison
  - ==
  - !=
  - >
  - <
  - >=
  - <=

C++ Operators

- Bitwise
  - >>
  - <<
  - &
  - |
  - ^
  - ~

Weird things with evaluation

Sequence determined by order of precedence

In C everything is evaluated and returns result

```
Assume          y = 1
Evaluate              sqr(x = y + 3 * 2)
Evaluates to    sqr(x = y + 6)
Evaluates to    sqr(x = 7)
Evaluates to    sqr(7) and puts 7 in x
   (called a side-effect)
Evaluates to    49
```

Where we're going, we don't need Boolean Values

- C++ does not believe in True or False
  - Same as C honestly
- Zero is False
- !False == True
  - The keywords True and False *do not exist*
- This lets you do some *really* weird things, as any time you want to check to see if a program or function worked, usually you check to see if they return *false* (as most main's return 0 when they work)

Back To The Bool Statement 2

- If (function())
  {
  // do thing if function didn't work
  }
- If (x > 10) {…}
  - If x is less than ten, returns 0, otherwise returns some value not zero
  - You have no idea what this is!
- If (x = y) {…}
  - Sets x to y. If y is some number not 0, do the enclosed block. If y *is* zero, don't do the enclosed block.

Blocks

- In C++, blocks are the same as C.
- You have used these, I don't know why this is in the lecture content.
- In case you somehow don't know:
- A statement looks like this:
  - Statement;
- A block looks like this:
  - {…}
- Anywhere a statement appears, you can put a block.

Control structures

- All stolen from C.
- If, for, switch, while…. Etc.

Pointers!

- Work almost exactly the same as C.
- *However*
- While you can still use “malloc” and “free”, C++ strongly recommends you use the new and delete keywords.
- This is because they can call destructors and constructors for objects, and are a *lot* safer than just naked malloc.
- There are specific new and delete statements for arrays too.

Arrays!

- Just like C, these allow for any number of dimensions
  - float f[2][4][20];
- Can be initialised
  - int r[3] = {1,2,3};
- They start from zero, and have *no* index checking!
  - G[-409] = 0; //This works, but I hope you know what you're doing
- This stems from the fact that arrays are just thinly veiled pointer aliases
  - G[-409] is the same as *(G-409)

New and Delete with Arrays

- Create an array?
  - int *p = new int[100];
- Delete it?
  - delete[] p;
  - This will call destructors on every element in the list *iff* they have them.

Functions

- Can be declared on the stack, or externally with the extern keyword (just like variables)
- There's also a main function (that must exist outside of any object) that returns int and takes in an int argc and a character pointer pointer argv
  - These correspond to the number of elements in the command line arguments and a list of strings of the arguments.

Function Overloading

- In C++, you can overload a function
  - int sqrt(int i);
  - float sqrt(float i);
- But C does not allow this! How can the two be compatible?
- This is because C++ "mangles" your function names before compiling them, making them unique, and therefore C compatible (and also incomprehensible to most readers).

Local Variables

- Just like C, declare anywhere, created and destroyed as you move through scopes.

Scopes!

- Block Scope:
  - I can see this because I am in the same block or a child of the block it's in
- Parameter Scope:
  - I can see this because it's one of my current function parameters.
- Static Scope:
  - I can see this because it's been given the static declaration somewhere above me
- Extern Scope:
  - This has the extern keyword and therefore refers to something "somewhere else", and via the extern variable I can see it.

Storage Classes

- These are similar to scopes, but define where the variable is stored.

Automatic (auto or default)
    Created and destroyed on stack when in scope

Register (register)
    Request register allocation (otherwise auto)

Static (static)
    Active for entire execution of program

Extern (declared outside functions)
    Active for entire execution of program

Other things it steals from C

- Strong typing
- Function prototypes (or forward declaration)
- Pass by reference:
  - int something(char &reference);
- Constants:
  - const float pi = 3.14;

#import <iostream>

- C++ asks nicely for you to use it's far superior stream based IO
- This is reasonably secure, and easier to use
- cout, cin, and cerr
- << exports to the stream, >> imports from it
- std::cout << "This is an int: " << i << std::endl;
- std::cin >> i;
- You can overload it for your own types
- std::cout << employee << bankaccount;

What on earth is std::?

- Namespaces.
- Namespaces allow you to *modularise groups of objects* for code reuse.
- These are basically how packages work in C++
- “namespace muhnamespace;”
- “using namespace muhnamespace;”
- Namespace std is the default namespace where all C++ standard library items exist.
- *Never* use “using namespace std;”.

What's in a class?

- Public, protected, and private members
  - Variables, also subprograms
- Scopes in objects are specified explicitly
  - Default scope does exist, and is private.

A C++ Class

- Split in many cases into a definition and code.
- *You do not write the class code in the class definition*
- This fits with C++'s understanding of classes as data *structs* with attached pointers to a shared group of *behaviours*.
- Naturally, those *behaviours* must be separate to the *struct*, you *declare* the struct, you *implement* the behaviors.
- And that's how you make a class

A C++ Header File

```
1  #pragma once //similar to an ifdef check
2
3  class Badguy
4  {
5      int health;
6      std::string name;
7  protected:
8      double damage;
9  public:
10     Badguy();
11     Badguy(std::string,int);
12     int ShootGuns();
13     void GetShot(int);
14     ~Badguy();
15
16 };
```

C++ Constructors

```
1  #include <string> //new std strings
2  #include <cstdlib> // same as stdlib.h
3  #include <iostream> //cin cout cerr
4
5  #include "badguy.h"
6
7  Badguy::Badguy()
8  {
9      name = "default";
10     health = 100;
11     damage = 0.0;
12 }
13
```

```
14 Badguy::Badguy(std::string iname,int ihealth)
15 {
16     name = iname;
17     health = ihealth;
18 }
19
```

C++ Behaviors

```
20 int Badguy::ShootGuns()
21 {
22     std::cerr << "shooting guns for 100 damage" << std::endl;
23     return 100;
24 }
25
26 void Badguy::GetShot(int idamage)
27 {
28     if ((double)idamage > damage)
29     {
30         damage = sqrt((double)idamage);
31         health -= idamage;
32     }
33     else
34     {
35         std::cout << "Hahaha! I am INVINCIBLE!\n";
36     }
37 }
38
```

C++ Destructors

```
39 Badguy::~Badguy()
40 {
41     std::cout << "Aaargh! I am defeated!" << std::endl;
42 }
```

C++ and Inheritance

- C++'s objects are unlike Smalltalk's, or Java's.
- There is no "object".
- Therefore, inheritance is capable of being far more complicated and nuanced than other OO like languages
- Allows single and multiple inheritance.
- Public, private, and protected inheritance
- Will call constructors and destructors appropriately.

Public Inheritance

- If classes inherit other classes with public inheritance:
  - Public members become public
  - Protected members become private
  - Private members become private

Protected Inheritance

- If inheritance is protected:
  - Public members become protected
  - Protected members stay protected
  - Private members stay private

Private Inheritance

- If inheritance is private
  - Public members become private
  - Protected members become private
  - Private members become *inaccessible*.
- This is the default scheme.

Multiple Inheritance

- C++ allows inheritance from more than one base class

```
class Hovercraft : public Boat, public Car
{
public:
    int IsFullOfEels = 1;
};
```

Multiple Inheritance

- It also allows repeated inheritance
  - *However,* it can't do that by just inheriting from the same class twice.
  - You have to write two intermediary classes that inherit from the first class, and then inherit from both of the two intermediary classes

```
class Unicycle{};

class FrontWheel: public Unicycle{};
class BackWheel: public Unicycle{};

class Bicycle:public FrontWheel, public BackWheel{};
```

Polymorphism in C++

- Limited to child classes of a base class
- Needs to be specified at compile time with the *virtual* keyword.
- *No dynamic or run time polymorphism in C++*

```
class Shape {
   virtual int draw();
   ...
   }

class Circle : public Shape {
   virtual int draw();
   ...
   }
```

Abstract classes in C++

- Abstract classes are any classes with a *Pure Virtual Function*.
- These are defined by setting a function to 0.

```
class A {
public:
    virtual pvf(...) = 0;
}
```

Templates, a useful part of C++

- Allows you to perform some level of polymorphism with function declarations, without having to rewrite the function or class.
- Effectively lets you build data structures that can accommodate any data type
- List<int> vs List<float> vs List<MyClass>

Declaring Templates

```
template<class T> class Vector {
    T x;
    T y;
public:
    Vector(T f, T g);
    T length();
    ~Vector();
};
```

Using Templates

```
Vector<float> vf(1.3, 8.0);
Vector<int> vi(-2, 4);
```

Exception Handling in C++

- Similar to ADA, rather than it's parent language C
- Has heirachical (inheritance based) or even default handling!
- Considerably more regular and readable than ADA.
- However, not *usually* the cause of crashes in C++
- This is because the error mechanisms in C are still present!
- Catching exceptions doesn't work if the kernel throws a SIGSEGV

Defining an Exception

```
class Stack {
...
class Stack_Exception { ... };
...
}
```

Throwing Exceptions

```
Stack::pop() {
   ...
   if ( stack_empty ) then raise Stack_Exception;
   ...
   }
```

Handling Exceptions

```
try {
...
// Operations which may raise exceptions
...
}
catch Stack::Stack_Exception {
     ...
     // Operations to handle exception
}
catch ... {
     // Other exception handler
     ...
}
```

C++ Summary

- Very powerful but complex language, combining the expressive power of OO with the programmatical power of C
- Aimed at expert C users, OO users, system developers
- Allows single and multiple inheritance
- Has exception handling, but doesn't always use it.
  - Not regular!
- Designed on giving the *programmers* the choice, rather than any OO or programming principles.

Eiffel

AS IN THE TOWER

Eiffel, the language that could have been king.

- Developed in the mid 1980s with all the other OO languages
- Uses a strange development mechanism called the "Design by Contract"
- Based reasonably strongly on OO principles
- Strongly typed. Everything (pretty much) is an object
- Basically loved by program language scholars for ages
- Turned out to just be a fad.
- Killed by non-free, non-open source nature.

Eiffel Contracts

- Eiffel is built on assertions
  - Precondition assertions
    - Requirements that must be satisfied by the client before calling the routine
  - Postcondition assertions
    - Conditions that must be guaranteed by the routine before it can return
  - Class Invariants
    - Conditions that must be satisfied by all instances of a class that are externally accessible.

Why was Eiffel cool?

- Eiffel was considered the uber-language by quite a few program language designers.
- Didn't have dialects, still doesn't
- Was high level
- Worked across systems
- Provided a strong OO environment (GC, etc)
- Pascal-like
- Offered Multiple and Repeated inheritance
- Enabled an assurance-first programming style

Eiffel Design Goals

- Strongly OO language that enforces better program design, and is at least as efficient as the competition
- Creates software that is:
  - Correct - Software is able to exactly perform tasks defined by its requirements and specification
  - Robust - Software performs in abnormal conditions
  - Extendible – Software is easily adaptable to changes in specification.
  - Reusable – Software is able to be reused, in whole or in part, for new applications.

Eiffel Concepts

- Eiffel only has classes and objects, no primitives as such, and no disembodied functions ala C++
- A combination of classes and objects is called a system in Eiffel, owing to it's strong following of OO
- There is no "program" per se in Eiffel
- The System has a "root" class, which is like a main object. The root object is created by Eiffel at the start of execution, and acts as the prime mover, initiating all other action in the system.

Root Class Example

```
class CALCULATOR
inherit
   SET_UP
creation
   make
feature
   make is
        local
                d: REAL
                f: double
        do
                d := .15; f := .15;
                io.putreal (d); io.putreal (f);
                io.putstring("%N*********************************%N");
                io.putstring("Calculator in reverse Polish form%N");
                io.putstring("*********************************%N");
                initialize;
                session
        end;
```

More Root Class

```
feature {NONE}  -- Implementation
   current_state: CALC_STATE;
   qst: QUESTION;
   pls: PLUS;
   quit_state: QUIT;
   mns: MINUS;
   mlt: MULTIPLY;
   dvd: DIVIDE;
   help_state: HELP;
   remove: EMPTY;
   session is
        do
                from
                        start
                until
                        over
                loop
                        action
                        next
                end
        end
```

Design By Contract

- Really awesome design concept that never took off.
- Basically, *every* effect and side effect a behaviour has is explicitly defined as either pre or post execution conditions.
- As the programmer, you will *always* know what is going to happen when you do anything.
- Behaviors can also define conditions they expect their callers to fulfil (This value must be instantiated, this character can't be 'z')
- Forces a self documenting and highly modularised nature onto any code written in the language.

How does Eiffel do DBC?

- Eiffel lets you set a series of Boolean expressions for both pre and post conditions, that are evaluated before and after execution.
- This allows you to expressly test for validity as the program executes.
- If something goes wrong, Eiffel itself will generate exceptions describing the failure to fulfil the contract.
- Can be handled through a “rescue” clause!
- Exceptions work (once thrown) much like Java’s.

Eiffel Inheritance

- Eiffel allows single, repeated, and multiple inheritance.
- You can rename superclasses (alias them) to avoid name clashes (far nicer than C++'s method)
- Invariant clauses apply to any child, and classes cannot be combined if their invariants are contradictory.
- Polymorphism is also constrained by DBC, you cannot create new functionality that violates previous contracts! (works for both static and dynamic polymorphism!)

Deferred Classes

- Deferred classes and behaviors basically allow for abstract classes in Eiffel.
- Act kind of like contracts in their own right.
- Declaring a routine as deferred will indicate that there is no default implementation of that routine. This enforces a contract on the children that states that the routine cannot be used until it is implemented
- If the whole class is deferred, the class is effectively abstract.

Eiffel Compiler?

- Eiffel can be compiled or interpreted, depending on your compiler/interpreter.
- Microsoft had a compiler at one point, no longer.
- The standard Eiffel compiler was closed source and extremely expensive.
- Other free compilers did not keep up with feature creep.
- This was simply because people were not using it.

What do classes contain?

- Very similar in structure to Java or C# with class attributes, behaviors, constructors, and destructors declared in the class scope.
- Has an indexing feature that allows you to extend the class scope, allowing you to write classes in parts like in C++
- Can deprecate features of your code without destroying them (!!!) which throws soft exceptions warning of their expiry.
- Assertions and contract components.

I Am the Very Model of a Modern Major-GENERAL

- In C++, everything inherits from…. Er…
- In Java, everything inherits from Object
- In Eiffel, everything inherits from GENERAL
- GENERAL descends to PLATFORM, which contains platform specific info
- You can explicitly remove PLATFORM.
- In Eiffel, you also have NONE, which inherits from *everything*.

What do you mean NONE?

- Yes NONE
- It's a class that inherits from everything
- PROVIDED, no naming conflicts occur.
  - Has no instance. Ever.
  - Has no heirs. Ever.
  - Cannot be exported to anything.
  - Cannot be inherited by anything.
- Why? Well it's basically "NULL"

Cool things in Eiffel

- Routines
  - Behaviors, basically
  - Also interface and definition in *one*!
- Scopes
  - Defined exclusively by routine body, assertions, and rescue clauses
- Synonyms
  - Can be used to add new aliases for old functions, allowing you to implement new functions with old names in new classes.

More cool things

- Information Hiding
  - Two ways:
    - Specify in contracts the exact clients that can use a component
    - Restrictions that can be changed by inherited component
- Components of Classes can be of three types
  - Generally Available
  - Selectively Available
  - Secret

Components (or features)

- Components (features) are defined by:
  - signature (type of arguments and results)
  - implementation
  - specification (preconditions and postconditions)
- Component redeclaration is done using two approaches
  - redefinition - modify the inherited component's behaviour or specification
  - effecting - used for deferred components - define the implementation for a feature that did not have in the parent
    - Effecting involves one of two processes:
      - joining - combining/merging two or more features that were inherited as deferred
      - undefining - the inherited implementation of a component is discarded and the component is essentially a new feature
  - redeclaration of components cannot reduce the effectivness of a component

Typing in Eiffel

- Typing in Eiffel is static. Types are also classes, even basic ones.
- References do exist, but exist in “reference types” which are real objects that only contain references.
- Expanded types exist too, but create the objects they have references too so you don’t have to set them first.
- Basic types: bool, char, int, real, double
- Special types: Bit, array, string
- Anchor types (defined using LIKE) automatically redefine themselves in all descendant classes.

Control structures and design

- Semicolon ends lines
- Control structures use “END” to end themselves, but don’t use “BEGIN” to start
- Case statement is the “INSPECT” statement
  - Uses “WHEN x THEN y” format
- Loop is similar to the ADA loop structure in design.
- Also has a “Debug” flag that lets you set code that only executes if the debug flag is on.

Attibutes in Eiffel

- Objects can have two types of attributes
- Variable:
  - It's a variable… like normal
- Constant:
  - Is constant. Two kinds, manifest constant or unique constant.
  - Manifests are set by the programmer at time of writing.
  - Uniques are set once and then never change again.

Cloning Objects

- Eiffel has specific copy constructors inherent to every class
- Deep copy and Shallow Copy
  - Deep will copy every single object in the data structure, an object and every object it contains
  - Shallow just copies the object, and may or may not keep references to the rest of the structure based on whether the object copied is a reference class or not.
- This also is matched by shallow or deep equality
  - Shallow: Are the fields here the same?
  - Deep: Is every object this contains the same?

So Eiffel In A Nutshell

- Strong OO language designed to work with DBC.
- Designed for experienced programmers
- Allows for nearly all OO features.
- Highly readable, writeable.
- Honestly one of the most nicely designed languages out there.
- At the time, people thought it represented the future of programming.

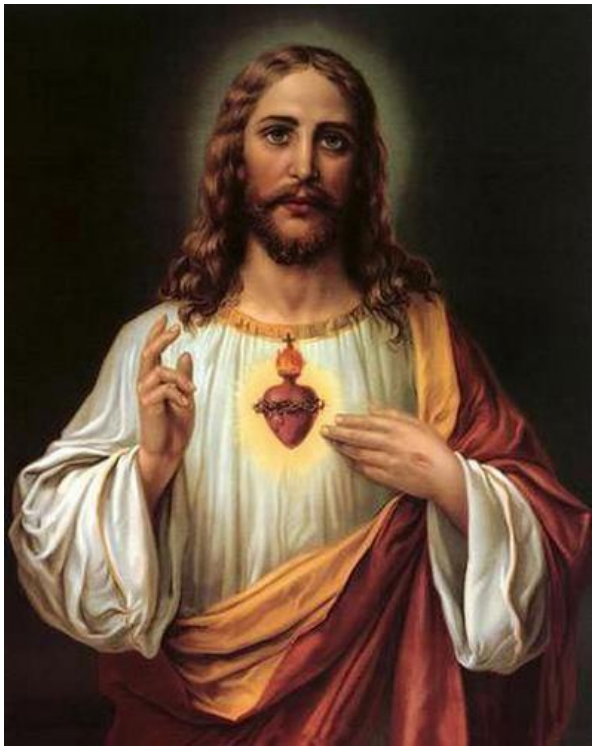
Why did Eiffel Die?

- the language in gaining in popularity - however the impact it will have on the programming community is still difficult to estimate

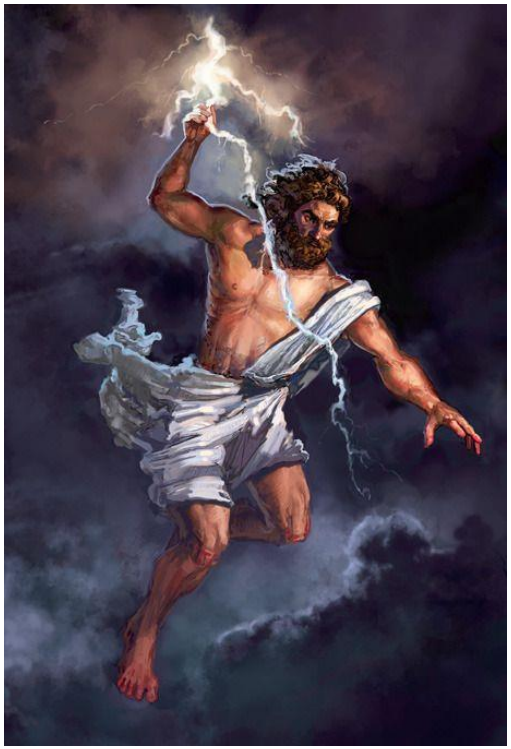
C#

ALL HAIL MIGHTY WINDOWS

Looks Familiar



Looks real familiar….



But oh, the Hemsworth

That’s what C# is.

C#? What is it?

- Hey Stefan, go on a rant about why Windows is awesome please.

So what is it?

- Language built to act as the language of the .NET framework and virtual machine.
- Strongly typed
- Similar to C++
- Almost the same as Java
- Incredible mind bending power… but only on Windows based devices.
  - PC, Xbox, Surfaces, Hololens, Windows Phone I guess…

Have you used java?

- Then you can probably use C#.
- Similar syntax and declarations to Java
  - Almost half way between C++ and Java actually
- No delete operator (because garbage collector)
- But you *can* call the garbage collector into existence.
- New means things more like the constructor in Java than in C++

C# Syntax

- Uses a main function (in a class somewhere) just like Java.
- C# statements have ; to end their lines, but don't end their objects with ;, just like Java
- C# allows methods to be overloaded… just like Java.
- Has in and out (ref or out) in function calls, *and* a pass by value default mode if you so wish.

C# Control structures

- Have you used Java?
- It uses Java control structures.
- Has a specific foreach loop though, ripped from Visual Basic

More things in C# - Basics

- Two data types:
  - Data types, not objects (int, char, whatever)
  - Reference data types (strings and objects, also Int, Char, etc…)
- In C#, everything can be treated as an object (even non objects, cause they get encapsulated in their object parents Power Rangers style through a thing called Boxing)
- All classes inherit from System.Object
- Arrays can be 1D, infD, or “jagged” (variable D)

C# basics some more

- C# introduces more OO compatible features:
  - Read only class members!
  - Sealed classes (you can't see what's in them but you can use them)
  - Aliasing (for inheritance and changing function names)
  - Properties *as a language construct*.

The Real Power Of C#

- In .NET, everything is a library.
- Therefore, C# can import anything as a library.
- Code reuse is practically infinite.
- Direct access to OS services as objects
  - DirectX, Window manager, Event loops…
- As Windows goes on, C# becomes exponentially more expressive
  - (in windows)
- Can be used for *literally anything*.
  - *Is still a regular, fairly simple to pick up language!!!*

C# in a nutshell

- The be all end all Windows engineering language.
- Strong support from Microsoft and literally anyone who writes code that works on Windows.
- However:
  - It's still a Microsoft project
  - I mean it's open source *now* but…
  - And it's an engineering language like C++
  - But seriously, if you like Java, this is like Java but with god mode.

The End