OO Language Comparison

Last Updated: 2018-02-25
by David Cooper
Curtin University

This is a quick reference for COMP2003 Object Oriented Software Engineering. It *does not* cover the unit material per se. Rather, it is designed to help you apply concepts presented in the lectures to various different OO languages.

There is an enormous amount of detail that is *not shown* here, for all languages. Also, the four languages chosen – Java, C#, C++ and Python – are certainly not the only OO languages (or even necessarily the best ones), but they are among the most popular at the time of writing.

In the HTML version of his document, the code snippets below contain hyperlinks to other online references. There is, of course, a wealth of other tutorial and reference material available if you search for it!

If you find any errors or deficiencies in these notes (keeping in mind the intentionally limited scope), please email david.cooper@curtin.edu.au (or whoever is coordinating COMP2003).

	Java	C#	C++	Python

Console I/O Example	***Simple.java***
	
	`import java.util.Scanner;`
	
	`public class Simple`
	`{`
	`    public static void main(String[] args)`
	`    {`
	`        String name;`
	`        int x;`
	`        double y;`
	`        Scanner sc = new Scanner(System.in);`
	
	`        System.out.print("Enter your name: ");`
	`        name = sc.nextLine();`
	
	`        System.out.print("Enter an integer: ");`
	`        x = sc.nextInt();`
	
	`        System.out.print("Enter a real number: ");`
	`        y = sc.nextDouble();`
	
	`        System.out.println("Hello " + name);`
	`        System.out.println("Product is " + ((double)x * y));`
	`    }`
	`}`
		simple.cs
		
		`using System;`
		
		`public class Simple`
		`{`
		`    public static void Main()`
		`    {`
		`        string name;`
		`        int x;`
		`        double y;`
		
		`        Console.Write("Enter your name: ");`
		`        name = Console.ReadLine();`
		
		`        Console.Write("Enter an integer: ");`
		`        x = Int32.Parse(Console.ReadLine());`
		
		`        Console.Write("Enter a real number: ");`
		`        y = Double.Parse(Console.ReadLine());`
		
		`        Console.WriteLine("Hello " + name);`
		`        Console.WriteLine("Product is " + ((double)x * y));`
		`    }`
		`}`
			simple.cpp
			
			`#include <string>`
			`#include <iostream>`
			
			`int main()`
			`{`
			`    std::string name;`
			`    int x;`
			`    double y;`
			
			`    std::cout << "Enter your name: ";`
			`    std::getline(std::cin, name);`
			
			`    std::cout << "Enter an integer: ";`
			`    std::cin >> x;`
			
			`    std::cout << "Enter a real number: ";`
			`    std::cin >> y;`
			
			`    std::cout << "Hello " << name << std::endl;`
			`    std::cout << "Product is "`
			`                << ((double)x * y) << std::endl;`
			`    return 0;`
			`}`
				simple.py
				
				`#!/usr/bin/python3`
				
				`name = input("Enter your name: ")`
				`x = int(input("Enter an integer:"))`
				`y = float(input("Enter a real number: "))`
				
				`print("Hello " + name)`
				`print("Product is " + str(float(x) * y))`
				
				`# Notes:`
				`# In Python 2, use 'raw_input()'.`

Compiling (with selected compilers)	***With JDK*** `[user@pc]$ javac Simple.java`	***With Mono*** `[user@pc]$ mcs simple.cs`	***With GCC*** `[user@pc]$ g++ -c simple.cpp`	***With CPython*** Compiled automatically when run.
Linking	Linked automatically when run.	Linked automatically when run.	`[user@pc]$ g++ simple.o -o simple`	Linked automatically when run.
Running	`[user@pc]$ java Simple`	`[user@pc]$ mono simple.exe`	`[user@pc]$ ./simple`	`[user@pc]$ python3 simple.py`

1. Basic Syntax and Data Types

Comments	`// Single-line comment`
	
	`/* Multi-`
	`    line`
	`    comment */`
		`// Single-line comment`
		
		`/* Multi-`
		`    line`
		`    comment */`
			`// Single-line comment`
			
			`/* Multi-`
			`    line`
			`    comment */`
				`# Single-line comment`
				
				`""" Multi-`
				`    line`
				`    comment/string """`

Variable Declarations	Same as C.
	
	`int myInt;`
	`float myFloat;`
	`double myDouble;`
	`char myChar;`
		Same as C.
		
		`int myInt;`
		`float myFloat;`
		`double myDouble;`
		`char myChar;`
			Same as C.
			
			`int myInt;`
			`float myFloat;`
			`double myDouble;`
			`char myChar;`
				Declarations not required.

	`boolean myBool;`
	`String myStr;`
		`bool myBool;`
		`string myStr;`
			`bool myBool;`
			`std::string myStr;`
				Declarations not required.

Converting between integers and reals	Same as C.
	
	`myInt = (int)myDouble;`
	`myDouble = (double)myInt;`
		Same as C.
		
		`myInt = (int)myDouble;`
		`myDouble = (double)myInt;`
			Same as C.
			
			`myInt = (int)myDouble;`
			`myDouble = (double)myInt;`
				`myInt = int(myFloat)`
				`myFloat = float(myInt)`

Converting from strings	`try`
	`{`
	`    myInt = Integer.parseInt(myStr);`
	`    myDouble = Double.parseDouble(myDouble);`
		`try`
		`{`
		`    myInt = Int32.Parse(myDouble);`
		`    myDouble = Double.Parse(myInt);`
			`#include <sstream>`
			`...`
			`std::stringstream(myStr) >> myInt;`
			`std::stringstream(myStr) >> myDouble;`
				`try:`
				`    myInt = int(myStr)`
				`    myFloat = float(myStr)`
				`except ValueError as e:`

	Java	C#	C++	Python

	`}`
	`catch(NumberFormatException e) { ... }`
		`}`
		`catch(FormatException e) { ... }`
		`catch(OverflowException e) { ... }`
		
		`// --- ALTERNATIVELY ---`
		
		`if(Int32.TryParse(myStr, out myInt) &&`
		`   Double.TryParse(myStr, out myDouble))`
		`{`
		`    // Success; myInt and myDouble have values`
		`}`
		`else`
		`{`
		`    // Error`
		`}`
			`if(std::ios::fail())`
			`{`
			`    std::ios::clear();`
			`    // Error`
			`}`
			`else`
			`{`
			`    // Success; myInt and myDouble have values`
			`}`
			
			Alternatively, `std::stringstream` can be made to throw an exception, by first calling its `exceptions()` method.
				`    ...`

Converting to strings	Various approaches; for example:
	
	`myStr = Integer.toString(myInt);`
	`myStr = String.valueOf(myInt);`
	`myStr = "" + myInt;`
		`myStr = myInt.ToString();`	`#include <sstream>`
			`...`
			`std::stringstream ss;`
			`ss << myInt;`
			`myStr = ss.str();`
				`myStr = str(myInt)`

2. Control Statements

If	Same as C.
	
	`if(condition1)`
	`{`
	`    ...`
	`}`
	`else if(condition2)`
	`{`
	`    ...`
	`}`
	`else`
	`{`
	`    ...`
	`}`
		Same as C.
		
		`if(condition1)`
		`{`
		`    ...`
		`}`
		`else if(condition2)`
		`{`
		`    ...`
		`}`
		`else`
		`{`
		`    ...`
		`}`
			Same as C.
			
			`if(condition1)`
			`{`
			`    ...`
			`}`
			`else if(condition2)`
			`{`
			`    ...`
			`}`
			`else`
			`{`
			`    ...`
			`}`
				`if condition1:`
				`    ...`
				
				`elif condition2:`
				`    ...`
				
				`else:`
				`    ...`

Switch	C-style, but Java 7 also permits `switch` to operate on strings.
	
	`switch(n)`
	`{`
	`    case 1: case 2:  // Identical cases allowed`
	`        ...`
	`        break;`
	
	`    case 3:`
	`        ...`
	`    case 4:          // Fall-through allowed`
	`        ...`
	`        break;`
	
	`    default:`
	`        ...`
	`}`
		C-style, but with `switch` (a) permitted to operate on strings, and (b) forbidden from "falling through" from one non-empty `case` to the next.
		
		`switch(n)`
		`{`
		`    case 1: case 2:  // Identical cases allowed`
		`        ...`
		`        break;`
		
		`    case 3:`
		`        ...`
		`    case 4:    // Compile error (fall through not allowed)`
		`        ...`
		`        break;`
		
		`    default:`
		`        ...`
		`}`
		
		The `goto case` construct is permitted in place of `break`, but you will probably regret it!
			Same as C (with only integer or enumerable types allowed).
			
			`switch(n)`
			`{`
			`    case 1: case 2:  // Identical cases allowed`
			`        ...`
			`        break;`
			
			`    case 3:`
			`        ...`
			`    case 4:          // Fall-through allowed`
			`        ...`
			`        break;`
			
			`    default:`
			`        ...`
			`}`
				Python has no direct `switch` equivalent. Use an `if-elif` statement (or, if you want to be really fancy, a dictionary of lambda functions).

While	Same as C (except the condition must be strictly boolean).
	
	`while(condition)`
	`{`
	`    ...`
	`}`
		Same as C (except the condition must be strictly boolean).
		
		`while(condition)`
		`{`
		`    ...`
		`}`
			Same as C.
			
			`while(condition)`
			`{`
			`    ...`
			`}`
				`while condition:`
				`    ...`

Do-While	Same as C (except the condition must be strictly boolean).
	
	`do`
	`{`
	`    ...`
	`} while(condition);`
		Same as C (except the condition must be strictly boolean).
		
		`do`
		`{`
		`    ...`
		`} while(condition);`
			Same as C.
			
			`do`
			`{`
			`    ...`
			`} while(condition);`
				Python has no direct `do-while` equivalent. Use a `while` loop, and either find a way to adjust the exit condition, or perform the first "iteration" before the loop itself,

	Java	C#	C++	Python

For	Same as C99.
	`for(int i = 5; i < 15; i++)`
	`{`
	`    ...`
	`}`
		Same as C99.
		`for(int i = 5; i < 15; i++)`
		`{`
		`    ...`
		`}`
			Same as C99.
			`for(int i = 5; i < 15; i++)`
			`{`
			`    ...`
			`}`
				`for i in range(5, 15):`
				`    ...`

For-Each	`for(MyClass obj : listOfObjects)`
	`{`
	`    ...`
	`}`
		`foreach(MyClass obj in listOfObjects)`
		`{`
		`    ...`
		`}`
			Since C++11:
			`for(MyClass* obj : listOfObjects)`
			`{`
			`    // Use 'MyClass&' instead if`
			`    // listOfObjects contains raw objects.`
			`    ...`
			`}`
			C++03 and earlier have no direct for-each equivalent. However, the following idiom, using a somewhat mangled standard for-loop, is common:
			`std::vector<MyClass*>::iterator it;`
			`// Replace 'vector' with another container`
			`// type if appropriate.`
			`for(it = listOfObjects.begin();`
			`    it != listOfObjects.end(); it++)`
			`{`
			`    MyClass* obj = *it;`
			`    // Use 'MyClass&' instead if`
			`    // listOfObjects contains raw objects.`
			`    ...`
			`}`
				`for obj in listOfObjects:`
				`    ...`

Break (typical usage)	Same as C.
	`boolean flag = false;`
	`while(whileCondition)`
	`{`
	`    ...`
	`    if(breakCondition)`
	`    {`
	`        ...`
	`        flag = true;`
	`        break;`
	`    }`
	`    ...`
	`}`
	`if(!flag)`
	`{`
	`    ... // Loop iterated to the end`
	`}`
	Applies to all loop types.
		Same as C.
		`bool flag = false;`
		`while(whileCondition)`
		`{`
		`    ...`
		`    if(breakCondition)`
		`    {`
		`        ...`
		`        flag = true;`
		`        break;`
		`    }`
		`    ...`
		`}`
		`if(!flag)`
		`{`
		`    ... // Loop iterated to the end`
		`}`
		Applies to all loop types.
			Same as C.
			`bool flag = false;`
			`while(whileCondition)`
			`{`
			`    ...`
			`    if(breakCondition)`
			`    {`
			`        ...`
			`        flag = true;`
			`        break;`
			`    }`
			`    ...`
			`}`
			`if(!flag)`
			`{`
			`    ... // Loop iterated to the end`
			`}`
			Applies to all loop types.
				`while whileCondition:`
				`    ...`
				`    if breakCondition:`
				`        ...`
				`        break`
				`    ...`
				`else:`
				`    ... # Loop iterated to the end`
				Applies to all loop types. The `else` clause is connected to the *loop* here, not the `if`.

Yield	Java has no direct `yield` equivalent.	C# `yield` statement:
		`using System.Collections.Generic;`
		`...`
		`public IEnumerable<int> getMultiples(int n, int count)`
		`{`
		`    for(int i = 0; i < count; i++)`
		`    {`
		`        yield return i * n;`
		`    }`
		`}`
		`...`
		`foreach(int i in getMultiples(3, 8))`
		`{`
		`    ... // i = 0, 3, 6, 9, 12, 15, 18, 21`
		`}`
			C++ has no direct `yield` equivalent.	Python `yield` statement:
				`def geMultiples(n, count):`
				`    for i in range(count):`
				`        yield i * n`
				`...`
				`for i in getMultiples(3, 8):`
				`    ... // i = 0, 3, 6, 9, 12, 15, 18, 21`

3. Exception Handling and Cleaning Up

Tutorials	Java exceptions.	C# exceptions.	C++ exceptions.	Python exceptions.
Throw/Raise	`throw new MyException("message");`	`throw new MyException("message");`	`throw MyException("message");`	`raise MyException("message")`

	Java	C#	C++	Python

Declaring checked exceptions	Java requires you to declare any “checked” exceptions your methods and constructors might throw. Checked exceptions are those that inherit from `Exception` but not from its subclass `RuntimeException`.
	
	`public void method() throws MyException, OtherException`
	`{`
	`    ...`
	`}`
		C# has no checked exceptions, and so does not require any such declaration.	C++ has no checked exceptions.
			
			Since C++11, the `noexcept` specifier *sort-of* declares that a function/method does not throw any exceptions:
			
			`void method() noexcept;`
			
			However, this guarantee is *not* implemented by actually checking whether exceptions are thrown and refusing to compile. Rather, if an exception *is* thrown, `noexcept` causes the method/function to abort the program! Also note that C++ has other error-handling mechanisms too, so `noexcept` does not cover a lot of what might go wrong.
			
			C++03 and earlier provided a superficially Java-like `throw(...)` declaration, but it was so badly misconceived as to defeat the point of exception handling, and it's now deprecated.
				Python has no checked exceptions, and so does not require any such declaration.

Try-Catch	`try`
	`{`
	`    ...`
	`}`
	`catch(MyException e)`
	`{`
	`    ...`
	`}`
	`catch(ExceptionType2 e)`
	`{`
	`    ...`
	`}`
		`try`
		`{`
		`    ...`
		`}`
		`catch(MyException e)`
		`{`
		`    ...`
		`}`
		`catch(ExceptionType2 e)`
		`{`
		`    ...`
		`}`
		
		C# 6.0 also supports “exception filters”, where you can specify a boolean condition for a `catch` block:
		
		`...`
		`catch(MyException e) when(e.getValue() == 42)`
		`{`
		`    ...`
		`}`
			`try`
			`{`
			`    ...`
			`}`
			`catch(MyException& e)`
			`{`
			`    ...`
			`}`
			`catch(ExceptionType2& e)`
			`{`
			`    ...`
			`}`
			
			Notice the use of “&”. It is not required per se, but otherwise exception objects are passed by value.
				`try:`
				`    ...`
				`except MyException as e:`
				`    ...`
				`except ExceptionType2 as e:`
				`    ...`
				`else:`
				`    ...`
				
				The `else` clause is optional. It is executed after `try` if no exceptions are raised. (This includes *any* exceptions, not just those dealt with by the `except` block(s).)

Chaining exceptions	Pass the 1st exception to the 2nd's constructor (same as C#):
	
	`...`
	`catch(ExceptionType1 e)`
	`{`
	`    throw new ExceptionType2("new message", e);`
	`}`
		Pass the 1st exception to the 2nd's constructor (same as Java):
		
		`...`
		`catch(ExceptionType1 e)`
		`{`
		`    throw new ExceptionType2("new message", e);`
		`}`
			You can throw exceptions from inside `catch` blocks, but C++ has no conventional way of informing the 2nd exception about the 1st.	As of Python 3, using a `from` clause:
				
				`...`
				`except ExceptionType1 as e:`
				`    raise ExceptionType2("new message") from e`
				
				In Python 2, you can still raise exceptions from inside `except` blocks, but there is no conventional way of informing the 2nd exception about the 1st.

Try-Finally	Same as C#.
	
	`try`
	`{`
	`    ...`
	`}`
	`// Optional 'catch' blocks here`
	`finally`
	`{`
	`    ...`
	`}`
		Same as Java.
		
		`try`
		`{`
		`    ...`
		`}`
		`// Optional 'catch' blocks here`
		`finally`
		`{`
		`    ...`
		`}`
			C++ has no direct `finally` equivalent. Instead, any clean-up actions related to a particular object should be done in the relevant destructor.	`try:`
				`    ...`
				
				`# Optional 'except' block(s) here`
				`# Optional 'else' block here`
				
				`finally:`
				`    ...`

Cleaning up resources	Java 7 onwards has a “`try`-with-resources” statement – an extension to the standard `try` statement.
	
	`try(MyResource res = new MyResource(...),`
	`    OtherResource res2 = new OtherResource(...))`
	`{`
	`    ... // res and res2 can be used here.`
	`}`
	`// Optional 'catch' block(s) here`
	`// Optional 'finally' block here`
	
	Resource classes must implement `AutoCloseable`, and their `close()` methods will automatically be called at the end of the `try` statement, regardless of how it ends.
	
	In Java 6 and earlier, a `try-finally` is required instead:
	
	`MyResource res = null;`
	`OtherResource res2 = null;`
	`try`
	`{`
		C#'s `using` statement is independent of its `try` statement.
		
		`using(MyResource res = new MyResource(...),`
		`      OtherResource res2 = new OtherResource(...))`
		`{`
		`    ... // res and res2 can be used here.`
		`}`
		
		Resource classes must implement `IDisposable`, and their `Dispose()` methods will automatically be called at the end of the `using` statement, regardless of how it ends.
			C++ achieves the same effect without an explicit statement. An object's destructor, if defined, is guaranteed to be called when the object goes out of scope. (However, be aware that if a *pointer* to an object goes out of scope, the *object* itself still exists, so C++ takes no action.)	Python's `with` statement is independent of its `try` statement.
				
				`with MyResource(...) as res, \`
				`     OtherResource(...) as res2:`
				`    ... # res and res2 can be used here.`

```
    res = new MyResource(...);
    res2 = new OtherResource(...);
    ...
}
finally
{
    if(res != null) res.close();
    if(res2 != null) res2.close();
}
```

However, this is not ideal. If an exception occurs in `try`, and then if *another* exception occurs in `finally`, the second will "take over" and block you from seeing the first, and yet the first is generally much more important. You can engineer a set of nested `try-catch-finally` blocks to get around this, but at great cost to readability.

4. Type Model Overview

	Java	C#	C++	Python
Value types	Java has a fixed set of "primitive types": `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. These types do not use references, and cannot be set to `null`.	C# has "value types", including: • Integer types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` and `char`; • Real types `float`, `double`, and `decimal`; • `bool`; • C-style enumerations (using `enum`); • Structs (using `struct`), which share much in common with classes, but which cannot inherit from other classes/structs. Value types can be made "nullable" (allowing `null` to be a valid value) by appending "`?`" to the type: `int? x = null;`	All C++ datatypes are value types, except when explicitly using "&" in the type. There is no difference, in terms of storage and referencing, between a struct and a class. Only pointer types may be set to `NULL`.	Python does not have value types.
Reference types	All Java objects (including arrays) are accessed via reference variables, and all references work the same way. Any reference can be set to `null`. References can have a class, interface, or array type.	In C#, all instances of classes (but not structs) are accessed via reference variables. The `string` class has its own special reference type (which makes `s1 == s2` equivalent to `s1.Equals(s2)`, but only for strings). There is also a `dynamic` reference type used to bypass static type checking. In addition, C# allows method parameters to be declared `ref` or `out`, which creates another level of referencing.	C++ has several mechanisms to create reference types: • C-style pointers. While a pointer itself is a value (a number representing a memory address), and what it points to is also a value, the combination of the two can be thought of as a reference type. • Lvalue references using "&". • Rvalue references using "&&" (since C++11). • Smart pointers (since C++11), including `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`.	All Python objects and values are accessed via reference variables, including even `int` and `float`. Any reference can be assigned to any type, and to "`None`" (which itself is technically just another object). (Standard types like `int` are immutable, which makes them feel like value types. The code "`x = 42`" isn't overwriting a numerical value, but rather making a reference variable point to the *object* "42", of class `int`. But the effect is much the same in the end.)
Garbage collection	Yes.	Yes.	No. All memory allocated via `new` (or `new[]`) must eventually be deallocated via `delete` (or `delete[]`). It is vitally important that you keep track of which object "owns" which other object, so that one can `delete` the other when the time comes.	Yes.
Type parameters	Java (version 5 and later) supports generics, where type parameters are checked at compile time and then *erased* during compilation. Type parameters can represent any reference type, but not primitive types.	C# supports generics, where type parameters are checked at compile time and retained ("*reified*") at runtime. Type parameters can represent any type.	C++ supports templates, where type parameters are *not* immediately checked. Rather, each type parameter is replaced by a type at compile time, and the compiler checks that the end result makes sense. It creates a separate copy of the entire class/method/function being made, for each type it is requested to handle.	Being dynamically-typed, Python has no use for type parameters.
Object class	All Java objects (including arrays) inherit from the `Object` class, directly or indirectly. However, primitive types have no supertype.	All C# types (both reference and value) inherit from the `Object` class, directly or indirectly. Value types in particular implicitly inherit from `ValueType`, which itself inherits from `Object`.	C++ has no "Object" class equivalent. If a class/struct does not specify a superclass, then it does not have one. However, C++'s templates provide a way of dealing with values of any type, as long as the actual type can be determined by the compiler.	As of Python 3, all types (including `int` and `float`) inherit from the `object` class, directly or indirectly. Prior to Python 3, there were two kinds of class: • "New-style", the only kind in Python 3; • "Old-style", the default kind in Python 2.x and the only kind prior to Python 2.1. Old-style classes do not inherit from `object`.
Inheritance model	Java supports single-inheritance with interfaces. A class inherits from one superclass, and can implement any number of interfaces. An interface can inherit from any number of other interfaces.	C# supports single-inheritance with interfaces. A class inherits from one superclass, and can implement any number of interfaces. A struct cannot inherit from another struct or class (other than `ValueType`), but it can implement any number of interfaces. An interface can inherit from any number of other interfaces.	C++ supports full multiple inheritance. A class/struct can inherit from zero-or-more other classes/structs. You can effectively create an interface by declaring a class/struct with only abstract ("pure virtual") methods.	Python supports full multiple inheritance. A class can inherit from one-or-more other classes. Interfaces are not really required (due to duck typing), but you can create one by declaring a class with only stubbed methods.
Type inference	In Java, a variable, parameter or field must always have its full type specified. The compiler will not attempt to infer its type. However, Java *does* perform type inference in two situations: • Since Java 7, the "diamond operator" `<>` can be used to avoid writing out type arguments more than once: `Map<String,LocalDateTime> myMap = new HashMap<>();`	Since C# 3, the `var` keyword can be used in place of a datatype for a local variable. The compiler will infer the type from the initialisation value. However, this doesn't apply to parameters or fields. `var myStr = new Dictionary<string,DateTime>();`	Since C++11, the `auto` keyword can be used in place of a datatype for a local variable. The compiler will infer the type from the initialisation value. However, this doesn't apply to parameters or fields. `auto myStr = "Hello world";`	Being dynamically-typed, Python does not perform type inference.

- Since Java 8, the compiler infers what interface a lamba/method reference must implement based on what it's being assigned to.

5. Classes, Interfaces, and Inheritance

Basic example

Person.java

```
public class Person
{
    private String name;
    private int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String getName()
    {
        return name;
    }

    public int getAge()
    {
        return age;
    }

    public void setAge(int age)
    {
        this.age = age;
    }
}
```

person.cs

```
// As of C# 3.0, using properties.
public class Person
{
    public string Name { get; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        this.Name = name;
        this.Age = age;
    }
}
```

Note: you could use ordinary fields instead of properties, and it would look the same as Java.

person.h (the header file)

```
#ifndef PERSON_H
#define PERSON_H

#include <string>

class Person
{
    private:
        std::string name;
        int age;

    public:
        Person(std::string name, int age);
        ~Person();

        std::string getName() const;
        int getAge() const;
        void setAge(int age);

}; // Note semicolon!
#endif
```

person.cpp

```
#include "person.h"

Person::Person(std::string name, int age)
{
    this.name = name;
    this.age = age;
}

Person::~Person()
{
}

std::string Person::getName() const
{
    return name;
}

int Person::getAge() const
{
    return age;
}

void Person::setAge(int age)
{
    this.age = age;
}
```

person.py

```
# As of Python 2.6, using properties.
class Person:
    def __init__(self, name, age):
        assert isinstance(name, str) # Optional
        assert isinstance(age, int)  # Optional
        self._name = name
        self._age = age

    @property
    def name(self):
        return self._name

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age)
        self._age = age
```

Alternatively:

```
# Without properties -- more like Java.
class Person:
    def __init__(self, name, age):
        assert isinstance(name, str) # Optional
        assert isinstance(age, int)  # Optional
        self._name = name
        self._age = age

    def getName(self):
        return self._name

    def getAge(self):
        return self._age

    def setAge(self, age)
        self._age = age
```

Static fields and methods

```
public class Person
{
    // Constant
    public static final int DEFAULT_AGE = 42;

    // Non-constant (should be private)
    private static int nPeople = 0;

    // Static method
    public static Person loadPerson(String filename)
    {
        nPeople++;
        ...
    }
    ...
}
```

```
public class Person
{
    // Constants are implicitly static
    public const int DEFAULT_AGE = 42;

    private static int nPeople = 0;

    public static Person LoadPerson(string filename)
    {
        nPeople++;
        ...
    }
    ...
}
```

Header file

```
class Person
{
    private:
        // Non-const static fields must be initialised separately.
        static int nPeople;

    public:
        // True constants can be initialised right here.
        const static int DEFAULT_AGE = 42;
        static Person* loadPerson(std::string filename);
};
```

Implementation file

Static fields are defined directly within the `class` block. (Compare this to non-static fields, which are defined within the constructor `__init__()`.)

```
class Person:
    DEFAULT_AGE = 42
    nPeople = 0

    @staticmethod
    def loadPerson(filename):
        Person.nPeople += 1
        ...
```

<table>
<tr><th></th><th>Java</th><th>C#</th><th>C++</th><th>Python</th></tr>
<tr><td></td><td></td><td></td><td>

```
// Define and initialise a previously-declared static field.
int Person::nPeople = 0;

Person* Person::loadPerson(std::string filename)
{
    nPeople++;
    ...
}
```

</td><td></td></tr>
<tr><td>

Accessing static fields and methods

</td><td>

```
System.out.print(Person.DEFAULT_AGE);
Person newPerson = Person.loadPerson("datafile.txt");
```

</td><td>

```
Console.Write(Person.DEFAULT_AGE);
Person newPerson = Person.loadPerson("datafile.txt");
```

</td><td>

C++ uses "`::`" to access static members.

```
std::cout << Person::DEFAULT_AGE;
Person* newPerson = Person::loadPerson("datafile.txt");
```

</td><td>

```
print(Person.DEFAULT_AGE)
newPerson = Person.loadPerson("datafile.txt")
```

</td></tr>
<tr><td>

Single inheritance and constructors

</td><td>

```
// Superclass
public class Product
{
    public Product(...) {...}
    ...
}

// Subclass
public class ClothingProduct extends Product
{
    public ClothingProduct()
    {
        // Call superclass constructor
        super(...);
        ...
    }
    ...
}
```

</td><td>

```
// Superclass (aka the "base" class)
public class Product
{
    public Product(...) {...}
}

// Subclass (aka the "derived" class)
public class ClothingProduct : Product
{
    public ClothingProduct()
        : base(...) // Call superclass constructor
    {
        ...
    }
    ...
}
```

</td><td>

Header file(s)

```
// Superclass (aka the "base" class)
class Product
{
    public:
        Product(...);
        ...
};

// Subclass (aka the "derived" class)
class ClothingProduct : public Product
{
    public:
        ClothingProduct();
        ...
};
```

You must write "`public`" before the superclass name, because the default is a rather unhelpful C++ concept called "private inheritance" (which isn't really inheritance at all).

Implementation file(s)

```
// Superclass constructor
Product::Product(...) {...}

// Subclass constructor
ClothingProduct::ClothingProduct()
    : Product(...) // Call superclass constructor
{
    ...
}
```

</td><td>

```
# Superclass (aka the "base" class)
class Product:
    def __init__(self, ...):
        ...

    ...

# Subclass (aka the "derived" class)
class ClothingProduct(Product):
    def __init__(self):
        # Call superclass constructor
        super().__init__(...)
        ...

    ...
```

</td></tr>
<tr><td>

Abstract classes and method overriding

</td><td>

A subclass *can* override any superclass method *except* those declared `private`, `static` and/or `final`. A non-abstract subclass *must* override any superclass method declared `abstract`.

```
public abstract class Superclass
{
    // Must be overridden.
    public abstract void methodA();
    protected abstract void methodB();

    // Can be overridden.
    public void methodC() {...}
    protected void methodD() {...}

    // Cannot be overridden.
    private void methodE() {...}
    public final void methodF() {...}
    public static void methodG() {...}
}

public class Subclass extends Superclass
{
    @Override
    public void methodA() {...}

    @Override
    protected void methodB() {...}

    @Override
    public void methodC() {...}
```

</td><td>

A subclass *can* override any superclass method declared `virtual`, and a non-abstract subclass *must* override any declared `abstract`.

```
public abstract class Superclass
{
    // Must be overridden.
    public abstract void MethodA();
    protected abstract void MethodB();

    // Can be overridden.
    public virtual void MethodC() {...}
    protected virtual void MethodD() {...}

    // Cannot be overridden.
    private void MethodE() {...}
    public void MethodF() {...}
    public static void MethodG() {...}
}

public class Subclass : Superclass
{
    public override void MethodA() {...}
    protected override void MethodB() {...}
    public override void MethodC() {...}
}
```

</td><td>

A subclass *can* override any superclass virtual method, and *must* override any have "`= 0`" at the end of their declaration (signifying "pure virtual").

"Virtual-ness" itself is inherited. If you override a method declared `virtual`, the overriding method itself is automatically virtual too.

Overriding is independent of access level. You can have a private virtual method, but it's not a good idea.

```
class Superclass
{
    // Must be overridden (pure-virtual).
    virtual void methodA() = 0;

    // Can be overridden (but the superclass has its own
    // definition in Superclass.cpp).
    virtual void methodB();

    // Cannot be overridden.
    void methodC();
};

class Subclass : public Superclass
{
    void methodA() override;
    void methodB() final override; // Can't be further overridden.

    // If needed, 'const' must come before override/final:
    // void methodA() const override;
```

</td><td>

All methods are overridable, but there is no built-in language construct equivalent to "abstract", "virtual", "override" or "final". However, a simple approach is to have your superclass method raise an exception if it's supposed to be overridden.

```
class Superclass:
    # Must be overridden
    def methodA(self):
        raise NotImplementedError

    # May be overridden
    def methodB(self):
        ...

class Subclass(Superclass):
    def methodA(self):
        ...

    def methodB(self):
        ...
```

That being said, the `abc` ("abstract base class") module adds on the "`@abstractmethod`" construct. The effect is still to raise an exception during runtime, not to prevent compilation, though the exception is raised a bit earlier (during object construction, rather than when the method is called).

```
from abc import ABC, abstractmethod

class Superclass(ABC):
```

</td></tr>
</table>

	Java	C#	C++	Python

	`}`		`}`
			
			The keywords `virtual`, `override` and `final` only appear in the header file's class declaration, not in the implementation file.
			
			Only C++11 onwards has `override` and `final`. However, they're both optional, and overriding can happen without the `override` keyword. It's just there to help the compiler check what you're doing.
				`    # Must be overridden`
				`    @abstractmethod`
				`    def methodA(self): pass`
				
				`class Subclass(Superclass):`
				`    def methodA(self):`
				`        ...`

Interface implementation	`public interface EventObserver`
	`{`
	`    // Implicitly 'abstract' and 'public'`
	`    void alert(AlertEvent event);`
	`}`
	
	`public class AlarmSystem implements EventObserver`
	`{`
	`    @Override`
	`    public void alert(AlertEvent event)`
	`    {`
	`        ...`
	`    }`
	`    ...`
	`}`
		We *don't* write "`override`" when implementing an interface method. However, we can (optionally) prefix the method name with the interface name.
		
		`public interface EventObserver`
		`{`
		`    // Implicitly 'abstract' and 'public'`
		`    void Alert(AlertEvent event);`
		`}`
		
		`public class AlarmSystem : EventObserver`
		`{`
		`    // Writing "EventObserver." is optional here.`
		`    public void EventObserver.Alert(AlertEvent event)`
		`    {`
		`        ...`
		`    }`
		`    ...`
		`}`
			Technically C++ does not have interfaces, but you can use a class to do the same thing.
			
			`class EventObserver`
			`{`
			`    public:`
			`        virtual void alert(AlertEvent event) = 0;`
			`};`
			
			`public class AlarmSystem : public EventObserver`
			`{`
			`    public:`
			`        void alert(AlertEvent event) override;`
			`        ...`
			`}`
				Technically Python does not have interfaces, but there are two viable options:
				
				• Simply *describe* the interface in your documentation, and just remember to give your implementing classes the correct set of methods ("duck typing"), OR
				• Use a class instead; for instance:
				
				`from abc import ABC, abstractmethod`
				
				`class EventObserver(ABC):`
				`    @abstractmethod`
				`    def alert(event): pass`
				
				`class AlarmSystem(EventObserver):`
				`    def alert(event):`
				`        ...`
				`    ...`

6. Containers (Arrays, Lists, Sets and Maps)				

Arrays	Java's arrays can contain any primitive or reference type.
	
	`// Declare and create an array of size n.`
	`MyClass[] myArray = new MyClass[n];`
	
	`for(int i = 0; i < myArray.length; i++)`
	`{`
	`    myArray[i] = new MyClass();`
	`    System.out.println(myArray[i]);`
	`}`
	
	`for(MyClass obj : myArray)`
	`{`
	`    System.out.println(obj);`
	`}`
		C#'s arrays can contain any value or reference type.
		
		`MyClass[] myArray = new MyClass[n];`
		
		`for(int i = 0; i < myArray.Length; i++)`
		`{`
		`    myArray[i] = new MyClass();`
		`    Console.WriteLine(myArray[i]);`
		`}`
		
		`foreach(MyClass obj in myArray)`
		`{`
		`    Console.WriteLine(obj);`
		`}`
			C++'s arrays can contain any type, and can be allocated either on the stack or on the heap (the latter by means of `new[]` and `delete[]`).
			
			(You can technically also use `malloc()` and `free()` from C, but this is a bit archaic in C++.)
			
			`// Declare and create arrays of size n:`
			`MyClass* stackArray[n];`
			`MyClass** heapArray = new MyClass*[n];`
			
			`// Note: C++ has no "array.length" equivalent.`
			`for(int i = 0; i < n; i++)`
			`{`
			`    stackArray[i] = new MyClass();`
			`    heapArray[i] = new MyClass();`
			`    std::cout << stackArray[i]`
			`              << heapArray[i] << std::endl;`
			`}`
			
			`// Heap-based arrays must be deleted later on:`
			`delete[] heapArray;`
			
			`// Don't forget to delete the MyClass objects`
			`// too (before you lose track of them!).`
			
			C++11 introduces the `std::array` class, which wraps around an array and provides a similar interface to other container types.
				Python does not have a general-purpose array construct. Use a `list` or `tuple` instead.
				
				The Python API does have an `array` class, which can represent arrays of selected integer and float types (corresponding to the standard C types). However, this class can *only* store those specific primitive types. It is intended only for performance-critical code.

Container API	The Java "Collections Framework" (part of the standard Java API) contains various interrelated interfaces and classes that represent lists, sets and maps, all in the `java.util` package. These containers can *only* contain reference types, not primitives. To work around this, Java provides the wrapper classes `Integer`, `Long`, `Short`, `Byte`, `Float`, `Double`, `Character` and `Boolean`. The compiler automatically "boxes" primitive values into wrapper objects as needed. (The creation and garbage-collection of wrapper objects incurs some performance overhead, which may become noticeable in performance critical code. In such cases, arrays or 3rd-party libraries may be more appropriate.)	C# uses .NET's `System.Collections.Generic` namespace, which contains various interrelated interfaces and classes that represent lists, sets and maps. They can contain any reference or value type (i.e. anything).	The C++ "Standard Template Library" (part of the standard API) contains various list, set and map implementations. These are all within the `std` namespace, but in separate standard header files. C++ containers can contain anything. However, storing pointers to objects in sets, or as map keys, is complicated and requires some additional definitions.	Basic lists, sets and maps are built into the Python language itself, but there are also more specialised forms available in the standard `collections` module. Python's containers can contain anything.

Lists	The classes `ArrayList` and `LinkedList` both implement a common `List` interface.
	
	`import java.util.List;`
	`import java.util.ArrayList;`
		The class `List` represents an array-based list, and implements the `IList` interface.
		
		There is also a `LinkedList` class, which (curiously) *does not* implement `IList` and has a somewhat different set of methods.
			The `vector` class implements an array-based list, while `list` implements a linked-list. There is no common superclass, but they share essentially the same set of methods.
			
			The methods `begin()` and `end()` retrieve iterators that initially point
				The `list` class represents an array-based list. It is one of several "sequence types", with others being:
				
				• `tuple` – an immutable list used to bundle together a fixed group of values, often of different types and often temporarily.

	Java	C#	C++	Python

	`import java.util.LinkedList;`
	`...`
	
	`List<MyClass> list1 = new ArrayList<>();`
	`List<MyClass> list2 = new LinkedList<>();`
	
	`// Adding`
	`list1.add(new MyClass());     // Append`
	`list1.add(5, new MyClass());  // Insert`
	`list1.addAll(list2);          // Append multiple`
	
	`// Removing`
	`list1.remove(5);`
	`list1.remove(obj);`
	`list1.clear(); // Remove everything`
	
	`// Querying`
	`MyClass obj = list1.get(5);`
	`boolean objInList = list1.contains(obj);`
	`int index = list1.indexOf(obj);`
	`int size = list1.size();`
		`using System.Collections.Generic;`
		`...`
		
		`IList<MyClass> list1 = new List<MyClass>();`
		
		`// Adding`
		`list1.Add(new MyClass());`
		`list1.Insert(5, new MyClass());`
		
		`// Removing`
		`list1.RemoveAt(5);`
		`list1.Remove(obj);`
		`list1.Clear();`
		
		`// Querying`
		`MyClass obj = list1[5];`
		`boolean inList = list1.Contains(obj);`
		`int index = list1.IndexOf(obj);`
		`int size = list1.Count; // Property, not a method.`
			to the beginning and end of the list, respectively. These are often used simply to *represent* positions in the list.
			
			`#include <vector>`
			`#include <list>`
			`...`
			
			`std::vector<MyClass*> list1; // Array-based`
			`std::list<MyClass*> list2;   // Linked-list-based`
			
			`// Adding`
			`list1.push_back(new MyClass()); // Append`
			`list1.insert(5, new MyClass()); // Insert`
			`list1.insert(list1.end(), list2.begin(), list2.end());`
			
			`// Removing`
			`// Note: if your list contains pointers, this does not deallocate`
			`// the objects. You must 'delete' them separately.`
			`list1.erase(list1.begin() + 5); // Remove at index`
			`list1.erase(std::find(list1.begin(), list1.end(), obj));`
			`list1.erase(list1.begin(), list1.end()); // Clear`
			
			`// Querying`
			`MyClass* objPtr = list1[5];`
			`list1::iterator location =`
			`    std::find(list1.begin(), list1.end(), obj);`
			`bool inList = (location != list1.end());`
			`int index = (location - list1.begin()); // Vectors only.`
			`size_type size = list1.size();`
				- `range` – a virtual sequence of numbers defined by start, stop, and step values. (The elements in the range are not stored, but calculated when needed.)
				
				`list1 = []   # OR: list1 = list()`
				`list2 = {"tree", "dog", "house"}`
				
				`# Adding`
				`list1.append(MyClass())`
				`list1.insert(5, MyClass())`
				`list1.extend(list2)`
				
				`# Removing`
				`del list1[5]`
				`list1.remove(obj)`
				`list1.clear()`
				
				`# Querying`
				`obj = list1[5]`
				`inList = obj in list1`
				`index = list1.index(obj)`
				`size = len(list1)`

Sets	The classes `HashSet` and `TreeSet` both implement a common `Set` interface.
	
	`import java.util.Set;`
	`import java.util.HashSet;`
	`import java.util.TreeSet;`
	`...`
	
	`Set<MyClass> set1 = new HashSet<>();`
	`Set<MyClass> set2 = new TreeSet<>();`
	
	`// Basic mutators`
	`set1.add(new MyClass());`
	`set1.remove(obj);`
	`set1.clear();`
	
	`// Set union`
	`set1.addAll(set2);`
	
	`// Set intersection`
	`set1.retainAll(set2);`
	
	`// Asymmetric difference (set1 - set2)`
	`set1.removeAll(set2);`
	
	`// Symmetric difference (union - intersection)`
	`// [DIY!]`
	
	`// Querying`
	`boolean isSuperset = set1.containsAll(set2);`
	`boolean inSet = set1.contains(obj);`
	`int size = set1.size();`
		The classes `HashSet` and `SortedSet` (tree-based) both implement a common `ISet` interface.
		
		`using System.Collections.Generic;`
		`...`
		
		`ISet<MyClass> set1 = new HashSet<MyClass>();`
		`ISet<MyClass> set2 = new SortedSet<MyClass>();`
		
		`// Basic mutators`
		`set1.Add(new MyClass());`
		`set1.Remove(obj);`
		`set1.Clear();`
		
		`// Set union`
		`set1.UnionWith(set2);`
		
		`// Set intersection`
		`set1.IntersectWith(set2);`
		
		`// Asymmetric difference (set1 - set2)`
		`set1.ExceptWith(set2);`
		
		`// Symmetric difference (union - intersection)`
		`set1.SymmetricExceptWith(set2);`
		
		`// Querying`
		`bool isSuperset = set1.IsSupersetOf(set2);`
		`boolean inSet = set1.Contains(obj);`
		`int size = set1.Count; // Property, not a method.`
			The `std::set` class represents an ordered, tree-based set. Since C++11, the `std::unordered_set` class represents a hash-based set. There is no common superclass. Notes:
			
			- Storing standard types (`int`, `std::string`, etc.) is straightforward, in both types of sets.
			- Storing your own class in an `std::unordered_set` requires you to define a functor to perform hashing.
			- Storing *pointers*-to-objects, in either type of set, requires you to define functors to perform pointer dereferencing when comparing/hashing.
			- *Not* storing pointers means the class being stored cannot have subclasses.
			
			In short, it's easy to store strings and ints in sets. It's reasonably easy (no harder than in other langauges) to store your own classes in `std::set` specifically, provided nothing inherits from them.
			
			`#include <set>`
			`#include <unordered_set>`
			`#include <algorithm> // For union, intersection & difference.`
			
			`// Sets of objects`
			`std::set<MyClass> set1;`
			`std::set<MyClass> set2;`
			`std::unordered_set<MyClass> set3;`
			
			`// Basic mutators`
			`set1.insert(MyClass());`
			`set1.erase(obj);`
			`set1.clear();`
			
			`// Set union (only works with std::set)`
			`std::set<MyClass> unionSet;`
			`std::set_union(set1.begin(), set1.end(),`
			`               set2.begin(), set2.end(),`
			`               unionSet.begin());`
			
			`// std::set_intersection(), std::set_difference() and`
			`// std::symmetric_difference() are used the same way.`
			
			`// Querying`
			`set1::iterator location = set1.find(obj);`
			`bool inSet = (location != set1.end());`
			`size_type size = set1.size();`
				The `set` class represents a hash-based set. There is also an immutable set implementation called `frozenset`.
				
				`set1 = set()`
				`set2 = {"tree", "dog", "house"}`
				`# 'set1' and 'set2' are of the same type.`
				`# Unfortunately, '{}' is NOT a set.`
				
				`# Basic mutators`
				`set1.add(MyClass())`
				`set1.remove(obj)`
				`set1.clear()`
				
				`# In-place set operations (modifying set1)`
				`set1.update(set2) # Union`
				`set1.intersection_update(set2)`
				`set1.difference_update(set2)`
				`set1.symmetric_difference_update(set2)`
				
				`# Operations that create new sets`
				`unionSet     = set1.union(set2)`
				`intersectSet = set1.intersection(set2)`
				`diffSet      = set1.difference(set2)`
				`symDiffSet   = set1.symmetric_difference(set2)`
				
				`# Equivalent symbolic forms`
				`unionSet     = set1 | set2`
				`intersectSet = set1 & set2`
				`diffSet      = set1 - set2`
				`symDiffSet   = set1 ^ set2`
				
				`# Querying`
				`isSuperset = set1.issuperset(set2)`
				`inSet = obj in set1`
				`size = len(set1)`

Iterating over lists and sets	`for(MyClass obj : list1)`
	`{`
		`foreach(MyClass obj in list1)`
		`{`
			`// Since C++11. If you have raw objects in your list/set:`
			`for(MyClass& obj : list1)`
				`for obj in list1:`
				`    ...`

	Java	C#	C++	Python

	`    ...`
	`}`
		`    ...`
		`}`
			`{`
			`    ...`
			`}`
			
			`// If you have pointers:`
			`for(MyClass* obj : list1)`
			`{`
			`    ...`
			`}`	

Maps	The classes `HashMap` and `TreeMap` both implement a common `Map` interface.
	
	`import java.util.Map;`
	`import java.util.HashMap;`
	`import java.util.TreeMap;`
	`...`
	
	`Map<KeyClass,ValueClass> map1 = new HashMap<>();`
	`Map<KeyClass,ValueClass> map2 = new TreeMap<>();`
	
	`// Mutators`
	`map1.put(key, val); // Add/overwrite key-value`
	`map1.remove(key);   // Remove key-value`
	`map1.clear();`
	
	`// Querying`
	`ValueClass val = map1.get(key); // Returns null if key not in map.`
	`boolean keyInMap = map1.containsKey(key);`
	`boolean valueInMap = map1.containsValue(val);`
	`int size = map1.size();`
		The classes `Dictionary` and `SortedDictionary` (tree-based) both implement a common `IDictionary` interface.
		
		`using System.Collections.Generic;`
		`...`
		
		`IDictionary<KeyClass,ValueClass> dict1 =`
		`    new Dictionary<KeyClass,ValueClass>();`
		`IDictionary<KeyClass,ValueClass> map2 =`
		`    new SortedDictionary<KeyClass,ValueClass>();`
		
		`// Basic mutators`
		`dict1[key] = val;`
		`dict1.Add(key, val); // Throws ArgumentException if key is`
		`                     // already in the dictionary.`
		`dict1.Remove(key); // Remove key-value`
		`dict1.Clear();`
		
		`// Querying`
		`ValueClass val = dict1[key]; // Throws KeyNotFoundException if`
		`                             // key not in dictionary.`
		`bool keyInMap = dict1.ContainsKey(key);`
		`bool valueInMap = dict1.ContainsValue(val);`
		`int size = dict1.Count; // Property, not a method.`
			The `std::map` class represents an ordered, tree-based map. Since C++11, the `std::unordered_map` class represents a hash-based map. There is no common superclass. Notes for map *keys* (same principle as for sets):
			
			• Using standard types (`int`, `std::string`, etc.) as keys is straightforward, in both types of maps.
			• Using your own class of keys in an `std::unordered_map` requires you to define a functor to perform hashing.
			• Using *pointers*-to-objects as keys, in either type of map, requires you to define functors to perform pointer dereferencing when comparing/hashing.
			• *Not* using pointers means your key class cannot have subclasses.
			
			In short, it's easy to use strings and ints as map keys. It's reasonably easy (no harder than in other langauges) to use your own classes as keys in `std::map` specifically, provided nothing inherits from them.
			
			`#include <map>`
			`#include <unordered_map>`
			`...`
			
			`// Define two maps; with value-typed objects as keys, and`
			`// pointers to objects as values.`
			`std::map<KeyClass,ValueClass*> map1;`
			`std::unordered_map>KeyClass,ValueClass*> map2;`
			
			`// Basic mutators`
			`map1[key] = val; // Add/overwrite key-value`
			`map1.erase(key); // Remove key-value. Note: if your map values`
			`                 // are pointers, this does not deallocate the`
			`                 // objects. You must 'delete' them separately.`
			`map1.clear();`
			
			`// Querying`
			`ValueClass* val = map1[key]; // Adds key-NULL to map if key is`
			`                             // not already there.`
			`ValueClass* val = map1.at(key); // Throws std::out_of_range if`
			`                                // key not in map.`
			`bool keyInMap = (map1.find(key) != map1.end());`
			`size = map1.size();`
				The `dict` ("dictionary") class represents a hash-based map.
				
				`dict1 = {}  # OR: dict1 = dict()`
				`dict2 = {key1: value1, key2: value2, key3: value3}`
				
				`# Mutators`
				`dict1[key] = val  # Add/overwrite key-value`
				`del dict1[key]    # Remove key-value`
				`dict1.clear()`
				
				`# Querying`
				`val = dict1[key]`
				`keyInMap = key in dict1`
				`size = len(dict1)`

Iterating over maps	`// Iterate over keys`
	`for(KeyClass key : map1.keySet())`
	`{`
	`    ...`
	`}`
	
	`// Iterate over values`
	`for(ValueClass val : map1.values())`
	`{`
	`    ...`
	`}`
	
	`// Iterate over key-value pairs`
	`for(Map.Entry<KeyClass,ValueClass> kv : map1.entrySet())`
	`{`
	`    KeyClass key = kv.getKey();`
	`    ValueClass val = kv.getValue();`
	`    ...`
	`}`
		`// Iterate over keys`
		`foreach(KeyClass key in dict1.Keys)`
		`{`
		`    ...`
		`}`
		
		`// Iterate over values`
		`foreach(ValueClass val in dict1.Values)`
		`{`
		`    ...`
		`}`
		
		`// Iterate over key-value pairs`
		`foreach(KeyValuePair<KeyClass,ValueClass> kv in dict1)`
		`{`
		`    KeyClass key = kv.Key;`
		`    ValueClass val = kv.Value;`
		`    ...`
		`}`
			`// Iterate over key-value pairs`
			`for(auto& kv : map1)`
			`{`
			`    KeyClass& key = kv.first;`
			`    ValueClass* val = kv.second;`
			`    // OR, if your map values are *not* pointers:`
			`    ValueClass& val = kv.second;`
			`    ...`
			`}`
			
			There's no specific way of getting only the keys or only the values.
				`# Iterate over keys`
				`for key in dict1.keys():`
				`    ...`
				
				`# Iterate over values`
				`for val in dict1.values():`
				`    ...`
				
				`# Iterate over keys-value pairs`
				`for key, val in dict1.items():`
				`    ...`

7. Object Comparability (for Sets and Maps)

Equality (for hash-based	`public class Person`
	`{`
		`public class Person`
		`{`
			In C++ we use == to check for equality. For any given class, we must create a method-like construct called `operator==()` that defines the	In Python, `x == y` is equivalent to `x.__eq__(y)`.

<table>
<tr><th></th><th>Java</th><th>C#</th><th>C++</th><th>Python</th></tr>
<tr>
<td>

sets/maps)

</td>
<td>

```java
    private String name;
    private int age;
    ...

    @Override
    public boolean equals(Object other)
    {
        boolean result = false;
        if(other instanceof Person)
        {
            Person pOther = (Person)other;
            result = name.equals(pOther.name) &&
                     age == pOther.age;
        }
        return result;
    }
}
```

</td>
<td>

```csharp
    public string Name { get; }
    public int Age { get; set; }
    ...

    public override bool Equals(object other)
    {
        bool result = false;
        if(other is Person)
        {
            Person pOther = (Person)other;
            result = Name.Equals(pOther.Name) &&
                     Age == pOther.Age;
        }
        return result;
    }
}
```

This uses properties to represent “Name” and “Age”. You could use ordinary fields and the `Equals()` method would be the same.

</td>
<td>

meaning of `obj1 == obj2`.

Header file

```cpp
class Person
{
    private:
        std::string name;
        int age;

    public:
        ...
        bool operator==(const Person& other) const;
};
```

Implementation file

```cpp
bool Person::operator==(const Person& other) const
{
    return name == other.name && age == other.age;
}
```

</td>
<td>

```python
class Person:
    def __init__(self):
        self.name = ...
        self.age = ...

    ...

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age
```

</td>
</tr>
<tr>
<td>

Hashing (for hash-based sets/maps)

</td>
<td>

The `Objects` class (not to be confused with `Object`) has a utility method for calculating hash codes.

```java
import java.util.Objects;

public class Person
{
    private String name;
    private int age;
    ...

    @Override
    public int hashCode()
    {
        return Objects.hash(name, age);
    }
}
```

</td>
<td>

```csharp
public class Person
{
    public string Name { get; }
    public int Age { get; set; }
    ...

    public override int GetHashCode()
    {
        return Name.GetHashCode() * 37 + Age;
        // Algorithms vary. Search online for advice on
        // general-purpose GetHashCode() implementations.
    }
}
```

</td>
<td>

TBD. It's complicated!

</td>
<td>

The built-in `hash()` function calculates a hash code for any standard type, including lists. So, you can just pass it a list of your class fields.

```python
class Person:
    def __init__(self):
        self.name = ...
        self.age = ...

    ...

    def __hash__(self):
        return hash([self.name, self.age])
```

</td>
</tr>
<tr>
<td>

Ordering (for tree-based sets/maps)

</td>
<td>

Implement the `Comparable` interface and define a `compareTo()` method (which takes the same type as the class it's in). This must return:

- Zero, if the objects are equal,
- A negative integer, if the current object is should come before the argument (in sorted order), OR
- A positive integer, if the current object is should come after the argument (in sorted order).

```java
public class Person implements Comparable<Person>
{
    private String name;
    private int age;
    ...

    @Override
    public int compareTo(Person other)
    {
        // Assuming 'null' isn't important.
        int result = name.compareTo(other.name);
        if(result == 0) // Names are the same, compare ages
        {
            result = age - other.age;
        }
        return result;
    }
}
```

</td>
<td>

Implement the `IComparable` interface and define a `CompareTo()` method (which takes an object). This must return:

```csharp
public class Person : IComparable
{
    public string Name { get; }
    public int Age { get; set; }
    ...

    public int IComparable.CompareTo(object other)
    {
        if(!(other is Person))
        {
            throw new ArgumentException();
        }

        // Assuming 'null' isn't important.
        Person otherP = (Person)other;
        int result = Name.CompareTo(otherP.Name);
        if(result == 0)
        {
            result = Age - otherP.Age;
        }
        return result;
    }
}
```

</td>
<td>

C++ uses the “<” operator to determine which object goes first, in sorted order. For any given class, we must create a method-like construct called `operator<()` that defines the meaning of `obj1 < obj2`.

Header file

```cpp
class Person
{
    private:
        std::string name;
        int age;

    public:
        ...
        bool operator<(const Person& other) const;
};
```

Implementation file

```cpp
bool Person::operator<(const Person& other) const
{
    // Fortunately, the < operator is already defined for
    // std::string.

    return name < other.name ||
        (name == other.name && age < other.age);
}
```

</td>
<td>

In Python, `x < y` is equivalent to `x.__lt__(y)`. The `functools` module can auto-generate methods for the other relational operators too (`__ne__()`, `__gt__()`, etc.), once you've defined two of them yourself.

```python
from functools import total_ordering

@total_ordering
class Person:
    def __init__(self):
        self.name = ...
        self.age = ...

    ...

    def __eq__(self): # As above
        return self.name == other.name and self.age == other.age

    def __lt__(self, other):
        # Bundle the fields into two tuples. This will compare each
        # pair of elements in turn.
        return (self.name, self.age) < (other.name, other.age);
```

</td>
</tr>
</table>