C# Comparisons

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

May 27, 2017

Overview I

1. Compilation and Execution
   - C++ Process
   - Java Process
   - C# Process
2. Engineering Concepts
   - Inheritance
     - Method overriding
     - Summary
   - Design Components
     - Structs
     - Unsafe C#
     - Memory Management
     - Generics/Templates
     - Nested classes/Enums
   - Collections
     - Sorting
     - Delegates
     - Lambdas

Overview II

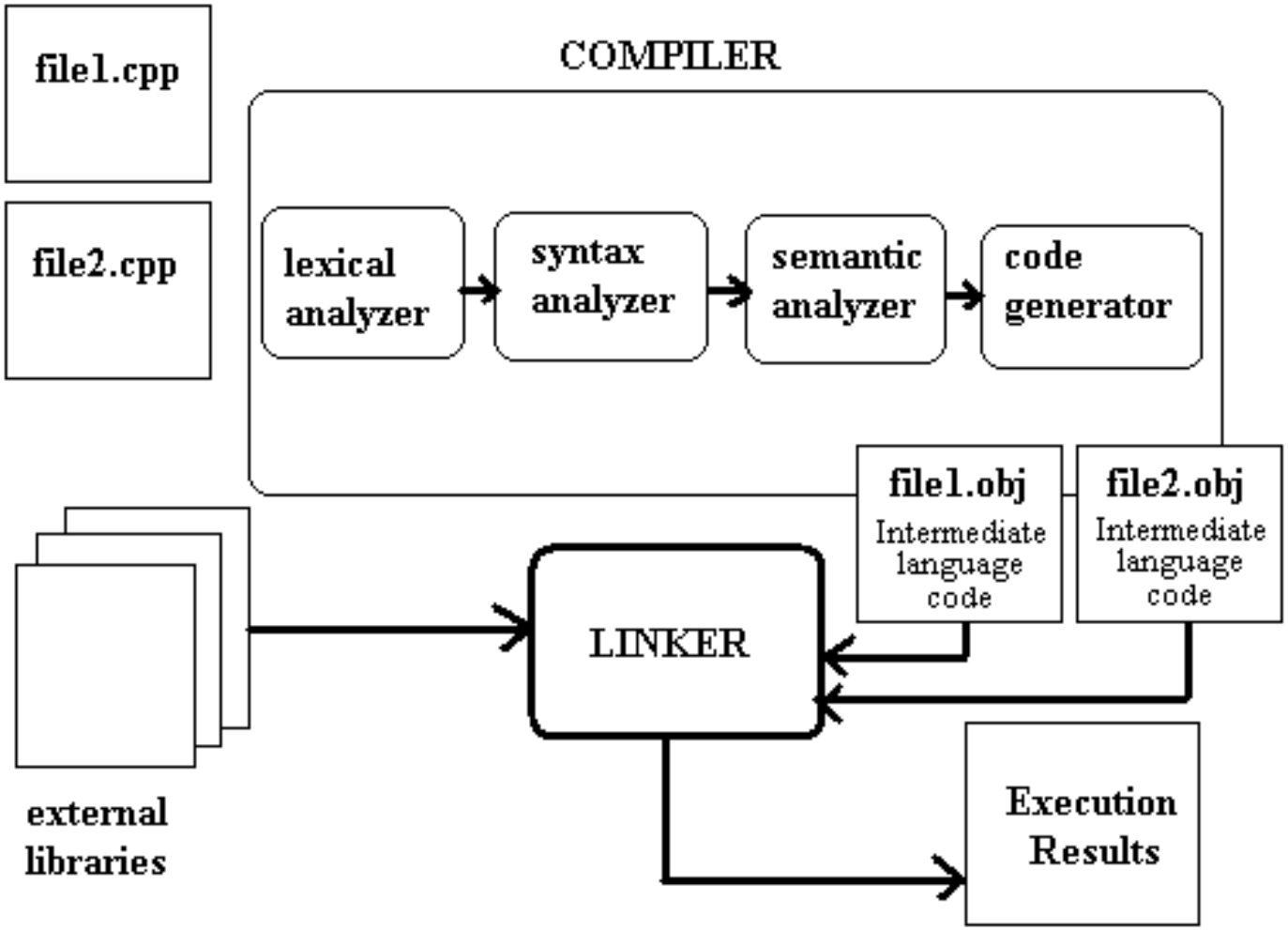
3 Differences Between Languages

Compilation and Execution

- C# is compiled to an intermediate language (IL) which then runs in the Common Language Runtime (CLR)
- The principle is that a range of compilers can convert code from a range of languages into IL and integrate them.
- Whereas Java creates a native code with Just in time compilers. IL code is always natively compiled before running.

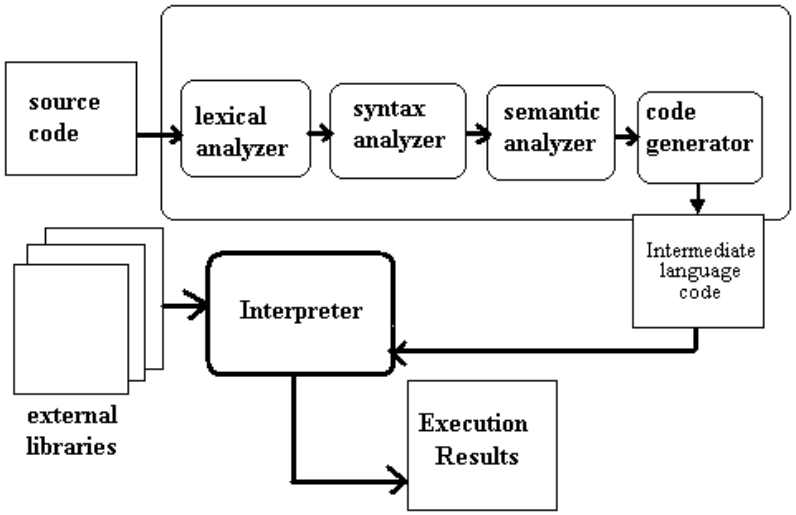
C++ Process

- C++ compiles source code into object files.
- Object files are then linked together with external libraries into machine dependant executables (via assembly language).



Java Process

- Uses a Hybrid compiler-interpreter method.
- Compiles source nto byte code.
- Byte code is portable.
- When a class (byte code file) is executed, the JVM compiles and executes it.
- Known as Just In Time compilation.
- Main method can change without recompiling the bytecode, since everything is compiled on the fly.



C# Process

- C# is compiled into IL equivalent to bytecode, but unit of compilation is in assembly (DLL file).
- This means if the main method changes then the whole project has to be recompiled.
- .NET then converts the assembly code into executables.
- The CLR is an environment in which the .NET applications that have been compiled to IL can be run.
- Does not interpret. It forms an executable.

Inheritance

```
class Class1 {}
interface IMyInterface{}
class Class 2: Class1, IMyInterface {}
```

- Uses c++ style to signify inheritance using a `:`.
- It is not obvious which class is inherited and which are implemented interfaces.
- By convention interfaces being with an I.
- Only single inheritance is allowed.
- All objects inherit from base class object.
- The operator `is` is the equivalent to `instanceof` in java.
- Made unextendable with the keyword `sealed`.

Method Overriding

- By default, methods cannot be overridden, they must be declared as `virtual`.
- Methods can be shadowed (Method called the same name, but not overridden, static type determines method called).
- Virtual methods can be overridden or shadowed.
- To override then the keyword `override` must be used, otherwise it is shadowed.

```
public class Mammal {
 virtual public void move(){// Overridden or shadowed
  Console.WriteLine("Move like a mammal in C#");}
 virtual public void jump(){ // Overridden or shadowed
  Console.WriteLine("Jump like an mammal in C#");  }
 public void dance(){ // Shadowed only
  Console.WriteLine("dance like a mammalin C#"); }
}
```

```
public class Dog : Mammal{
  override public void move(){ // Overrides method
     Console.WriteLine("Move like a dog in C#");}
   public void jump(){ // Shadows - Static type decides on method called
     Console.WriteLine("Jump like a dog in C#");}
   public void dance(){ // Shadows - Static type decides on method called
     Console.WriteLine("Dance like a dog in C#");}
}
```

Method Overriding - Cont

```
class Cat : Mammal
    {
        override public void jump(){
            Console.WriteLine("Jump like a Cat");}
          public void move(){
            Console.WriteLine("Move like a Cat");} }
```

```
static void Main(string[] args)
{
    Dog rover= new Dog();
    Cat mog = new Cat();
    Mammal mammal= new Mammal();
    Mammal myPet = rover; // Static - Mammal, Dynamic - Dog

    mammal.move();
        rover.move();
        mog.move();
        myPet.move();
        myPet=mog;
        myPet.move();
```

```
// Output:
// Move like an animal
// Move like a dog
// Move like a cat

// Type of My Pet = Dog
// Move like a dog

// Type of My Pet = Cat
// Move like a Mammal
}
```

Summary

- Interfaces and Abstract methods are the same as in Java.
- Don't shadow methods, bad form.
- In Java/C# all objects inherit from a common base class (`Object/object`), in C++, they don't.
- C++ allows multiple inheritance Java/C# Do not.
- By default, methods in C++ and C# cannot be overridden. Must be explicitly declared as `virtual`. Java can always be overridden unless stated as `final`.
- C++ and C# allow method shadowing... for some reason... who knows why?

Structs

- C# allows C like `structs`.
- These are value types and not reference types like classes are.
- The name of the struct variable is directly associated with the data and allocated on the stack rather than the heap.
- No inheritance.
- Useful for large, structured immutable data.
- Main benefit is that they do not need a reference to point to them and the memory is allocated at compile time which can be more efficient.

```
struct point {
        byte x;
        byte y;
}
```

```
class point2 {
        byte x;
        byte y;
}
```

```
Point[] myPoints = new Point[1000000];// 2 Million bytes allocated on stack
Point2[] mP2 = new Point2[1000000]; // 1 Million bytes allocated on stack the 2 million
    on Heap
```

Unsafe C#

- C# hides most of its memory management like Java.
- Possible to do C like pointer manipulation in C# if you are in an `unsafe` block or method.
- Must be compiled with the unsafe flag set.
- Bypasses all memory checking and management done by the CLR.
- Can have `unsafe structs` too.
- Again, bad form, stick to C.

Memory Management

- Constructors are the same as in Java.
- Can also have destructors in C#. Same syntax as C++.
- Semantics are like Java finalizers, but more useful.

Generics/Templates

- C++ uses Code specialization
- Java uses Type erasure
- C# performs code specialization for primitives (at second compilation phase).
- C# performs Type erasure for reference types.
- Syntactically, the same as Java, however you can have primitives and hence struct generics.

```
public struct Point<T> {
 public T x;
 public T y;
 public Point(T a, T b){
  x = a;
  y = b;
 }
}
```

```
using System.Collections.Generic.List;
static void genericTest(){

 List<string> myList = List<string>();
 List<Point<int>> points=new List<Point<int>>();
 myList.Add("Up the Arsenal");
 points.Add(new Point<int>(1, 2));
}
```

Nested Classes/Enums

- Nested classes:
  - C# has the equivalent to Java static nested classes.
  - No equivalent for local inner classes, anonymous classes or inner classes.
- Enums:
  - C# enums are simply masked integers like in C/C++.

Collections

- `System.Collections` or `System.Collections.Generic`
- Lists:
  - `ArrayList`
  - `LinkedList`
  - `SortedList`
  - `BitList`
  - `Stack and Queue`
- Set:
  - `HashSet`
  - `SortedSet`
- Maps:
  - `Dictionary Classes`
  - `HashTable`
  - **No** `TreeMap`
- Iterating:
  - Any class that implements the `IEnumerable` interface can be using in a `foreach` context.

```
IEnumerator<string> iterator = myList.GetEnumerator();
while (iterator.Current != null) // Gets current element
{
        Console.WriteLine(iterator.Current);
        iterator.MoveNext(); // Moves to next element
}
```

Sorting

- `Collections` has a default sort, to use it on a class, the class must implement `IComparable`.
- Java uses Functors to sort by another method than the default, C# uses Delegates.
- Delegates are a form of method pointer and hence can be used instead of Functors.

Delegates I

1. Declare `delegate` with a specific signature:

```
public delegate int CompareStudent(Student s);
public delegate int StaticCompare (Student a, Student b);
```

2. Write methods that could be stores in a delegate:

```
public int compNos(Student s)
{
  if (nos > s.nos)
    return 1;
  if (nos < s.nos)
    return -1;
  return 0;
}
```

```
public int compName(Student s)
{
  return name.CompareTo(s.name);
}
```

3. Now possible to declare a `delegate` and store a pointer to different functions:

Delegates II

```
// Declare two delegate references
Student.CompareStudent objectMethodPointer;
Student.StaticCompareStudent staticMethodPointer;

// Assign methods to them
objectMethodPointer = bob.CompNos;
staticMethodPointer =Student.CompNos;

// Call methods via delegate
int x = objectMethodPointer.Invoke(alice);
int y = staticMethodPointer.Invoke(bob, alice);

objectMethodPointer = alice.CompName;
staticMethodPointer = Student.CompName;
x = objectMethodPointer.Invoke(bob);
y = staticMethodPointer.Invoke(bob, alice);
```

- Often passed to methods to act as a form of selection.
- Often created at the method call in a similar way to anonymous inner classes:

```
myClass.findBest(
new Student.StaticCompareStudent(Student.CompNos)
);
```

- They are a form of anonymous function.

```
int square(int x) {return x*x;}
// Is the same as:
x=>x*x; // Lambda form
);
```

- Lambdas can be stored as `delegates`:

```
delegate int del(int i);
static void Main(string[] args) {
      del myDelegate = x => x * x;
      int j = myDelegate(5); //j = 25
}
```

Differences Between Languages

- **Inheritance:** In Java/C# all objects inherit from a common base class (`Object/object`).
- **Inheritance:** C++ allows Multiple inheritance, Java and C# do not,
- **Inheritance:** By default, methods in C++ and C# cannot be overridden, they must be explicitly declared as `virtual`. In Java they can always be overridden unless `final`.
- **Nested Classes:** Java has four types, C#/C++ only has static nested classes.
- **Enum:** Java enums are instances of anonymous inner classes, C++/C# are wrappers for integers.
- **Structs:** C++/C# allow structs (User defined primitives).
- **Generics/Templates:** Java uses Type erasure, C++ uses Code specialization, C# uses Type erasure for objects and Code specialization for primitives.
- **Threads:** Java/C# have built in threads, C++ does not.
- **Passing methods:** Java uses functors, C++ uses function pointers and C# uses Delegates
- **Operator overloading:** Allowed in C#/C++, not allowed in Java.

The End