Java and C# in depth
Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: framework overview and in-the-small features
Java and C# in depth
Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: framework overview
What’s in a name

Internal name of initial project: Cool (C-like Object Oriented Language)
  - Ruled out by the trademark lawyers

Chief C# architect at Microsoft: Anders Hejlsberg
  - Previously on Turbo Pascal & Delphi

Grounded in the .NET platform and CLI (Common Language Infrastructure)

“An imitation of Java”
  - According to Java’s Bill Gosling

Version 1.0: 2001

Latest version: 5.0 (August 2012)
C# platform goals (from ECMA standard)

- Simple, general-purpose, object-oriented
- Correct and robust
  - strong type checking, array bounds checking, detecting usage of uninitialized variables, automated memory management, ...
- Component- and reusability-oriented
- Programmer-portable
  - easy for developers coming from C/C++ and from other .NET languages
- No direct competition with C in terms of performance
- Introduction of selected functional programming features
  - Main motivation: dealing with data conveniently
CLI: Common Language Infrastructure

C# code → Compiler → Common Language Infrastructure → Common Intermediate Language → Common Language Runtime → 010011001010111010101100110

VB.NET code → Compiler → Common Language Infrastructure → Common Intermediate Language → Common Language Runtime → 010011001010111010101100110

J# code → Compiler → Common Language Infrastructure → Common Intermediate Language → Common Language Runtime → 010011001010111010101100110

.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).

The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
CIL and Assemblies

- C# compilation produces CIL (Common Intermediate Language) code
- Instruction set similar to Java bytecode
  - “object-oriented stack-based assembly code”
- CIL code is organized in assemblies
  - for Windows platforms: .exe and .dll
- Executed by a Virtual Machine (VM)
  - .NET on Windows platforms
  - Mono for Linux/Unix
- Code generation usually with a JIT compiler
  - AOT (Ahead-Of-Time) option also available
Security

1. Of the language:
   - Restricted: no pointers, no explicit memory de-allocation, checked type casts, enforced array bounds

2. Of the runtime: CAS (Code Access Security)
   - Code group
     - Associate evidences with permission types
     - Associations vary according to environment-dependent policies
   - Evidence
     - Any information associated with an assembly
       - E.g., digital signature of publisher, URL, an hash identifying the version, etc.

3. Verification and validation
   - Series of checks that make sure that the code doesn’t do anything clearly unsafe
     - Checks can be quite conservative: safe code may be rejected
Code generation: CLR

- CLR can denote two things:
  - the runtime component of CLI
  - Microsoft’s implementation of it for Windows platforms

- A JIT compiler converts CLI bytecode into native code just before running it
  - classes and methods are compiled dynamically just when they are needed

- Alternatively, a AOT (Ahead-Of-Time) compiler translates the whole application in native code
  - NGEN (Native Image Generator) in Microsoft’s CLR
  - not necessarily overall faster than JIT: certain dynamic optimization can be done only with JIT
CLR: more features

- Exception handling

- Memory management (garbage collection)

- Threads and concurrency

- Usually includes set of libraries: FCL (Framework Class Libraries)

- Has other languages running on top of it
  - VB.NET
  - J# (transitional language from Java to C#)
  - IronPython, IronRuby, IronScheme
  - ...

Java and C# in depth
Command-line C#

- Compile
  
  \[
  \begin{align*}
  \text{csc} & \quad a\_file.cs \quad // \quad \text{Microsoft} \; .NET \\
  \text{mcs} & \quad a\_file.cs \quad // \quad \text{Mono} \; .NET
  \end{align*}
  \]

- Execute
  
  \[
  \begin{align*}
  a\_file.exe \\
  ./a\_file.exe
  \end{align*}
  \]

- Generate XML documentation
  
  \[
  \begin{align*}
  \text{csc} & \quad /doc:docu.xml \; a\_file.cs \\
  \text{mcs} & \quad -doc:docu.xml \; a\_file.cs
  \end{align*}
  \]

- Compile all .cs files in the current directory and pack them in a DLL
  
  \[
  \begin{align*}
  \text{csc} & \quad /target:library \; /out:a\_library.dll \; *\,.cs \\
  \text{mcs} & \quad -target:library \; -out:a\_library.dll \; *\,.cs
  \end{align*}
  \]
Java and C# in depth
Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: in-the-small language features
Encoding and formatting

- Uses unicode as encoding system: [www.unicode.org](http://www.unicode.org)

- Free format
  - Blanks, tabs, new lines, form feeds are only used to keep tokens separate

- Comments
  - Single line: //Single line comment
  - Multiple lines: /* non-nested, multi-line comment */
  - Comment for XML documentation system:
    /** multi line */ // single-line line */
Identifiers

- Maximum length: 255 characters

- Can start with _ or @ or a letter

- Cannot start with a digit or a symbol other than _ or @

- Cannot include / or –

- @ can appear only in the first position

- Cannot be a keyword
Attributes are something else in C#

The counterparts to Java’s annotations
Meant to provide additional declarative information about program entities, which can be retrieved at run-time.

Typical usages:

- **Debugging information**
  e.g.: line number in the source where a method is called
  \[\text{[CallerLineNumber]}\]

- **Information for code analysis/compilation**
  e.g.: to compile certain code only in debugging mode
  \[\text{[Conditional ("DEBUG")]}\]

- **Compiler flags**
  e.g.: to generate a warning during compilation
  \[\text{[Obsolete ("You’d better use class X instead")]}\]
### Keywords

<table>
<thead>
<tr>
<th>abstract</th>
<th>as</th>
<th>base</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>by</td>
<td>byte</td>
<td>case</td>
</tr>
<tr>
<td>catch</td>
<td>char</td>
<td>checked</td>
<td>class</td>
</tr>
<tr>
<td>const</td>
<td>continue</td>
<td>decimal</td>
<td>default</td>
</tr>
<tr>
<td>delegate</td>
<td>do</td>
<td>double</td>
<td>descending</td>
</tr>
<tr>
<td>explicit</td>
<td>event</td>
<td>extern</td>
<td>else</td>
</tr>
<tr>
<td>enum</td>
<td>false</td>
<td>finally</td>
<td>fixed</td>
</tr>
<tr>
<td>float</td>
<td>for</td>
<td>foreach</td>
<td>from</td>
</tr>
<tr>
<td>goto</td>
<td>group</td>
<td>if</td>
<td>implicit</td>
</tr>
<tr>
<td>in</td>
<td>int</td>
<td>interface</td>
<td>internal</td>
</tr>
<tr>
<td>into</td>
<td>is</td>
<td>lock</td>
<td>long</td>
</tr>
<tr>
<td>new</td>
<td>null</td>
<td>namespace</td>
<td>object</td>
</tr>
<tr>
<td>operator</td>
<td>out</td>
<td>override</td>
<td>orderby</td>
</tr>
<tr>
<td>params</td>
<td>private</td>
<td>protected</td>
<td>public</td>
</tr>
<tr>
<td>readonly</td>
<td>ref</td>
<td>return</td>
<td>switch</td>
</tr>
<tr>
<td>struct</td>
<td>sbyte</td>
<td>sealed</td>
<td>short</td>
</tr>
<tr>
<td>sizeof</td>
<td>stackalloc</td>
<td>static</td>
<td>string</td>
</tr>
<tr>
<td>select</td>
<td>this</td>
<td>throw</td>
<td>true</td>
</tr>
<tr>
<td>try</td>
<td>typeof</td>
<td>uint</td>
<td>ulong</td>
</tr>
<tr>
<td>unchecked</td>
<td>unsafe</td>
<td>ushort</td>
<td>using</td>
</tr>
<tr>
<td>var</td>
<td>virtual</td>
<td>volatile</td>
<td>void</td>
</tr>
<tr>
<td>while</td>
<td>where</td>
<td>yield</td>
<td></td>
</tr>
</tbody>
</table>

Java and C# in depth
Operators

- Primary: ., (), [], x++, x--, new, typeof, checked, unchecked
- Unary: +, -, !, ~, ++x, --x, (aType)x
- Multiplicative: *, /, %
- Additive: +, -
- Shift: <<, >>
- Relational: <, >, <=, >=, is, as
- Equality: ==, !=
- Logical (precedence left to right): &, ^, |, &&, ||
- Conditional: condition ? (expr1):(expr2)
- Assignment: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- Precedence: from top to bottom
- Tip: don’t rely too much on precedence rules: use parentheses
```csharp
int i = 2147483647 + 10;  // compiler error
int ten = 10
int j = 2147483647 + ten; /* no compiler error.
Result: -2147483639. Overflow checking can be enabled by compiler options, environment configuration or the checked keyword. */
Console.WriteLine(checked(2147483647 + ten));
// OverflowException
Console.WriteLine(unchecked(2147483647 + 10));
// no compiler error. Result: -2147483639
```
Type system: value types

- Basic value types
  - sbyte, short, int, long, byte, ushort, uint, ulong, decimal, float, double, bool, char
  - struct
  - enum

- Nullable types for value types
  ```csharp
  int? n = null; ...
  if (n != null){int m = n.Value}
  ```

  ```csharp
  int p = n ?? 7 //null coalescing operator: //if n != null p = n, otherwise p = 7
  ```
Type system: reference types

- [] (arrays)
- class
- interface
- delegate
- event

Pointers
- restricted to blocks marked **unsafe**
- **unsafe** blocks can be executed only with certain permissions enabled
Widening conversions with precision loss

```csharp
float g(int x) {
    return x;
}
...
int i = 1234567890;
float f = g(i);
Console.WriteLine(i - (int)f)
// output: -46
...
```
Boxing and unboxing

- Variables of value types are stored on the stack
- Variables of reference types are stored on the heap

- **Boxing** transforms a value type into a reference of type `object` and is implicit
  
  ```java
  int i = 2;   object o = i;
  ```

- **Unboxing** transforms a reference of type `object` into a value type and requires a cast
  
  ```java
  object o = 3;   int i = (int)o;
  ```
Control flow: conditional branch

Same syntax as in C/C++/Java

```java
if (booleanExpr)
{
    // do something
}
else  // else is optional
{
    // do something else
}
```
Control flow: loops

```java
while (booleanExpr)
{
    // execute body
    // until booleanExpr becomes false
}

do
{
    // execute body (at least once)
    // until booleanExpr becomes false
}
while (booleanExpr);
```
Control flow: for loop

for (int i=0; i < n; i++)
{
    // execute loop body n times
}

// equivalent to the following
int i=0;
while (i < n)
{
    // execute loop body n times
    i++;
}
Control flow: **foreach** loop

```csharp
foreach (variable in collection)
{
    // loop body
}
```

- **collection** is an array or an object of a class that implements **IEnumerable**
- Executes the loop body for every element of the collection, assigned iteratively to **variable**
Control flow: `switch` selector

```
switch (Expr) {
    case value: instructions;
        break;
    case value: instructions;
        break;
    // ...
    default: instructions;
        break;
}
```

- `Expr` can be an integer or `string` expression
- `break` is required after each non-empty block
  - Including the `default` block
  - Fall through forbidden unless an `instructions` block is empty
Breaking the control flow: **break** and **continue**

**break;**
- Within a loop or a switch
- Exit the loop or switch

**continue;**
- Within a loop
- Skip the remainder of the current iteration and continue with the next iteration
Breaking the control flow: **goto**

**Label: instruction**
- Identifies an instruction (possibly compound, such as a loop)

**goto** **Label**;
- Anywhere
- Transfer control directly to the labeled statement

**goto** **case** **value**;
**goto** **default**;
- Within a **switch** (replacing standard **break** terminator)
- Transfer control to the corresponding **case** or to the **default**