Performance of Imperative C# Code

Bart J.F. De Smet
bartde@outlook.com

pluralsight
hardcore dev and IT training

Array Initializers

- **Data known at compile time**
  - Constants, e.g. Math.PI
  - String literals, e.g. “Hello, world!”
  - Composite data structures, e.g. arrays
- **No need to compute at runtime**
  - Embed result in assembly as bit sequence
  - Map data into memory
  - Provide strongly typed, safe accessor
- **Example**

```
class Foo {
   static readonly int[] s_primes = { 2, 3, 5, 7 };
}
```

Array Initializers

```
.class Foo
{
  .field private static initonly int32[] s_primes

  .method private static void rtspecialname .cctor()
  {
    ldc.i4.4
    newarr [mscorlib]System.Int32
    dup
    ldtoken field valuetype '<PrivateImpl>/__Arr16'::$$mtd
    call void [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers
        ::InitializeArray(
            class [mscorlib]System.Array,
            valuetype [mscorlib]System.RuntimeFieldHandle)
    stsfld int32[] Foo::s_primes
    ret
  }
}
```

Compiler-generated type (<>)

In lieu of 4 stelem

Array Initializers

```
.class Foo {
  .method private static void rtspecialname .cctor() {
    …
    ldtoken field valuetype ‘<PrivateImpl>/__Arr16’::$$mtd
    …
  }
}

.class <PrivateImpl> {
  .class value nested sealed __Arr16 {
    .pack 1
    .size 16
  }
  .field static assembly valuetype ‘<PrivateImpl>/_Arr16’ $$mtd
    at I_00002050
}

.data cil I_00002050 = bytearray ( 02 00 00 00 03 00 00 00 … )
```

Array Initializers

```
C:\Demo> dumpbin.exe /all ra.dll
Microsoft (R) COFF/PE Dumper Version 12.00.20617.1
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file ra.dll
…
SECTION HEADER #1
   .text name
     484 virtual size
    2000 virtual address (10002000 to 10002483)
         …
RAW DATA #1
  …
  10002050: 02 00 00 00 03 00 00 00 05 00 00 00 07 00 00 00
  10002060: 5E 1A 8D 04 00 00 01 25 D0 02 00 00 04 28 04 00
  …
```

Switch Statements

- **Switch in C#**
  - Unordered set of cases
  - Optional default case
  - Cases have to be literals or compile time constants
  - No implicit fall-through
  - Governing types:
    - Integral types (int, long, enums, etc.)
    - bool, char, string
    - Nullable variants of these
    - User-defined implicit conversion allowed
- **Switch in IL**
  - Jump instruction
  - One unsigned integer operand on the stack
  - Arguments consist of:
    - The number of jump targets
    - An offset for each target

Switch Statements

- **Simple switch statement**
  - Using integral numbers
  - Consecutive, zero-based cases

```
static string Switch(int x)
{
  switch (x)
  {
    case 0:  return “zero”;
    case 1:  return “one”;
    case 2:  return “two”;
    default: return null;
  }
}
```

```
.method static string Switch(int32 x)
{
  IL_00:  ldarg.0
  IL_03:  switch (IL_16, IL_1c, IL_22)
  IL_14:  br.s IL_28
  IL_16:  ldstr “zero”
  IL_1b:  ret
  IL_1c:  ldstr “one”
  IL_21:  ret
  IL_22:  ldstr “two”
  IL_27:  ret
  IL_28:  ldnull
  IL_29:  ret
}
```

Switch Statements

- **Simple switch statement**
  - Fall-through from switch using branch to “default” code
  - Encoding of instructions using offsets

```
.method static string Switch(int32 x)
{
  IL_00:  /* 02                                */ ldarg.0
  IL_03:  /* 45 0300 0200 0800 0E00 */ switch (IL_16, IL_1c, IL_22)
  IL_14:  /* 2B 12                             */ br.s IL_28
  IL_16:  /* 72 (70)000001                     */ ldstr “zero”
  IL_1b:  /* 2A                                */ ret
  IL_1c:  /* 72 (70)00000B                     */ ldstr “one”
  IL_21:  /* 2A                                */ ret
  IL_22:  /* 72 (70)000013                     */ ldstr “two”
  IL_27:  /* 2A                                */ ret
  IL_28:  /* 14                                */ ldnull
  IL_29:  /* 2A                                */ ret
}
```

Switch Statements

- **Non-zero offsets**
  - Sort cases to find closest to zero
  - Rebase to zero using add or sub

Rebasing

```
static string Switch(int x)
{
  switch (x)
  {
    case 10:  return “ten”;
    case 11:  return “eleven”;
    default:  return null;
  }
}
```

```
.method static string Switch(int32 x)
{
  IL_00:  ldarg.0
  IL_03:  ldc.i4.s 10
  IL_05:  sub
  IL_06:  switch (IL_15, IL_1b)
  IL_13:  br.s IL_21
  IL_15:  ldstr “ten”
  IL_1a:  ret
  IL_1b:  ldstr “eleven”
  IL_20:  ret
  IL_21:  ldnull
  IL_22:  ret
}
```

Switch Statements

- **Gaps between cases**
  - Fill gaps with jumps to default case
  - Only if the gaps aren't too big

```
static string Switch(int x)
{
  switch (x)
  {
    case 0:  return "zero";
    case 2:  return "two";
    case 4:  return "four";
    case 1:
    case 3:
    default: return null;
  }
}
```

```
.method static string Switch(int32 x) {
  IL_00:  ldarg.0
  IL_03:  switch (IL_1e, IL_30, IL_24,
                  IL_30, IL_2a)
  IL_1c:  br.s IL_30
  IL_1e:  ldstr "zero"
  IL_23:  ret
  IL_24:  ldstr "two"
  IL_29:  ret
  IL_2a:  ldstr "four"
  IL_2f:  ret
  IL_30:  ldnull
  IL_31:  ret
}
```

Switch Statements

- **Gaps between cases**
  - Too big gaps?
  - Can always compile using if…else… branching

```
static string Switch(int x)
{
  switch (x)
  {
    case 0:  return “zero”;
    case 3:  return “three”;
    default: return null;
  }
}
```

```
.method static string Switch(int32 x)
{
  IL_00:  ldarg.0
  IL_01:  stloc.0
  IL_02:  ldloc.0
  IL_03:  ldc.i4.0
  IL_04:  beq.s     IL_0c   // case 0
  IL_06:  ldloc.0
  IL_07:  ldc.i4.3
  IL_08:  beq.s     IL_12   // case 3
  IL_0a:  br.s      IL_18   // default
  IL_0c:  ldstr “zero”
  IL_11:  ret
  …
```

Switch Statements

- **Sparse labels**
  - Create clusters, build search tree
  - Try to minimize the number of evaluations

```
static string Switch(int x)
{
  switch (x)
  {
    case 10:  return “0x0A”;
    case 11:  return “0x0B”;
    case 12:  return “0x0C”;
    case 20:  return “0x14”;
    case 21:  return “0x15”;
    case 22:  return “0x16”;
    default:  return null;
  }
}
```

```
.method static string Switch(int32 x)
{
  IL_00:  ldarg.0
  IL_01:  stloc.0
  IL_02:  ldloc.0
  IL_03:  ldc.i4.s 10
  IL_05:  sub
  IL_06:  switch (IL_2e, IL_34, IL_3a)
  IL_17:  ldloc.0
  IL_18:  ldc.i4.s 20
  IL_1a:  sub
  IL_1b:  switch (IL_40, IL_46, IL_4c)
  IL_2c:  br.s IL_52
  …
```

Rebasing

Rebasing

Switch Statements

- **Switching on strings**
  - Start with if…else… compilation strategy
  - For small number of labels

```
static int Switch(string s)
{
  switch (s)
  {
    case “zero”:  return 0;
    case “one”:   return 1;
    case “two”:   return 2;
    default:      return -1;
  }
}
```

```
.method static int32 Switch(string s)
{
  IL_00:  ldarg.0
  IL_01:  dup
  IL_02:  stloc.0
  IL_03:  brfalse.s IL_34 // case null
  IL_05:  ldloc.0
  IL_06:  ldstr “zero”
  IL_0b:  call bool [mscorlib]System
           .String::op_Equality(string,
                                string)
  IL_10:  brtrue.s IL_2e
  …
```

Switch Statements

- **Switching on strings**
  - What about large numbers of labels? (6 or more today)
  - Lazily initialize a Dictionary<string, int> with switch values

```
static int Switch(string s)
{
  switch (s)
  {
    case “zero”:  return 0;
    case “one”:   return 1;
    case “two”:   return 2;
    case “three”: return 3;
    case “four”:  return 4;
    case “five”:  return 5;
    default:      return -1;
  }
}
```

```
static int Switch(string s) {
  if (<>_d == null) {
    <>_d = new Dictionary<string, int>
    {
      { “zero”, 0 },
      …
    };
  }

  int <>_x;
  if (<>_d.TryGetValue(s, out <>_x)) {
    switch (<>_x) {
      case 0: return 0;
      …
```

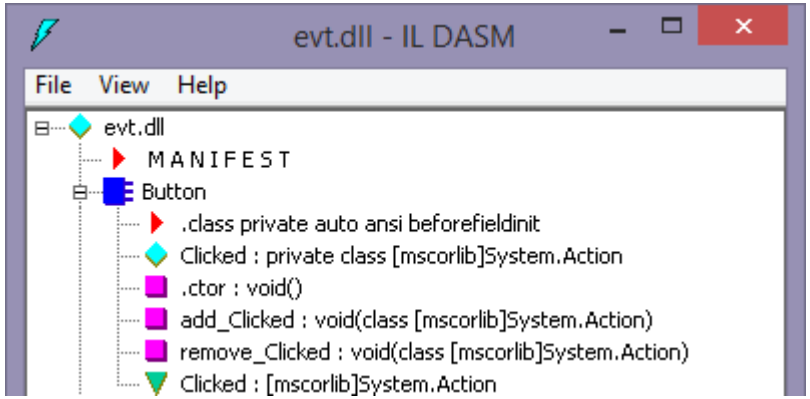
Volatile field

Switch Statements

- **Strings**
  - Case-sensitive, default comparer
  - May differ in other languages
- **Enums**
  - Friendly names for integral values
  - Compiled using their value
    - Same compilation strategy
    - Beware of versioning issues
- **Fall-through**
  - Empty case bodies
    - Same branch target for each case
  - Explicit goto
    - “goto case” and “goto default”
    - Become branch instructions

Events

- **C#'s view of events**
  - More than delegate-typed properties
  - Raising event requires private access
  - Attaching and detaching event handlers can have non-private access
- **CLR's view of events**
  - Metadata citizens, just like properties
  - Point to methods for add, remove, raise
- **Compiler-generated code**
  - Default code pattern
    - Delegate stored in field
    - Add and remove accessors
  - Customization pattern
    - Specify add and remove accessors
    - Manage underlying delegate storage

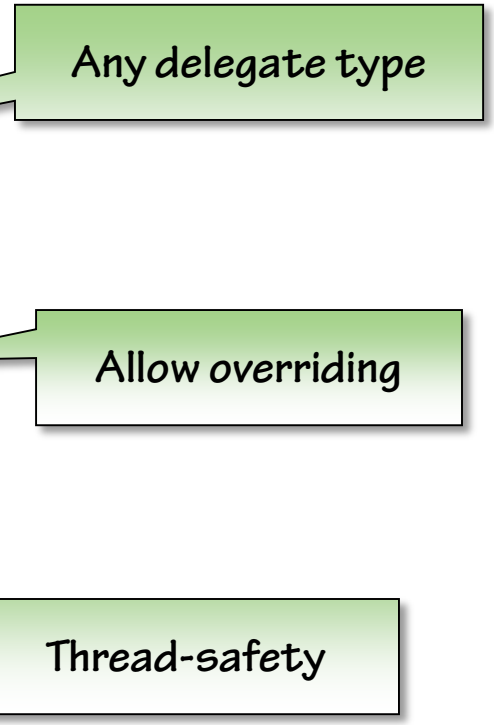


Events

- **Raising events**
  - Delegate invocation syntax
  - Typical On* method design pattern

```
class Button
{
  public event Action Clicked;

  protected virtual void OnClicked()
  {
    var clicked = Clicked;
    if (clicked != null)
    {
      clicked();
    }
  }
}
```



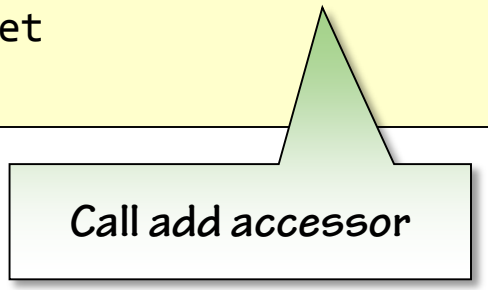
Events

- **Adding and removing handlers**
  - += and -= delegate syntax
  - Calls add and remove accessors

```
static void Demo(Button b)
{
  b.Clicked += Clicked;
}

static void Clicked()
{
  Console.WriteLine("Click");
}
```

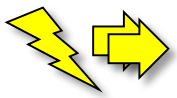
```
.method static void Demo(class Button b)
{
  ldarg.0
  ldnull
  ldftn  void Evt::Clicked()
  newobj instance void
   System.Action::.ctor(object, native int)
  callvirt instance void
   Button::add_Clicked(class System.Action)
  ret
}
```



Events

- **Inside add and remove**
  - + operator on delegates
    - `static Delegate Combine(Delegate a, Delegate b)`
  - - operator on delegates
    - `static Delegate Remove(Delegate a, Delegate b)`

- **Thread safety**
  - Non-atomic helper method calls
    - Read current delegate object
    - Call Combine or Remove
    - Write new delegate object
  - Compiler uses Interlocked helpers
    - `static T CompareExchange<T>(ref T location, T value, T comparand)`
      - Read current value from field
      - Combine or remove delegate
      - Try to swap in to field
      - Repeat until succeeded



```
void add_Clicked(Action a) {
  Action  old = _clicked;
  Action @new = (Action)
    Delegate.Combine(old, a);
  _clicked = @new;
}
```

Events

```
class Button
{
  private Action _clicked;

  public event Action Clicked
  {
    add
    {
      Action old, @new;
      do {
        old  = _clicked;
        @new = old + value; // calls Delegate.Combine
      } while (Interlocked.CompareExchange(ref _clicked, @new, old) != old);
    }

    remove { … }
  }
}
```

Events

- **Building a custom event manager**
  - Sparse usage of events
    - Many events exposed
    - Little event handlers attached
    - Delegate fields take space
  - Examples in the .NET Framework
    - Windows Forms
    - Windows Presentation Foundation (WPF)

- **Design points**
  - Beware of multi-threading implications
    - Single message loop to the rescue?
    - Careful add, remove, raise code
  - Provide easy declaration of events

Events

- **Windows Runtime (WinRT) support in .NET 4.5, C# 5.0**
  - System.Runtime.InteropServices.WindowsRuntime
    - **EventRegistrationToken**
      - Closer to Win32 handles
      - Add accessor returns a registration token
      - Remove accessor accepts a registration token
    - **EventRegistrationTokenTable<T>**
      - Stores mapping between delegates and tokens
      - AddEventHandler, used by compiler
      - RemoveEventHandler, used by compiler

```
.event TextChangedEventHandler TextChanged {
  .addon instance valuetype
   [Windows.Foundation]Windows.Foundation.EventRegistrationToken
   TextBox::add_TextChanged(class TextChangedEventHandler)
 .removeon instance void TextBox::remove_TextChanged(valuetype
   [Windows.Foundation]Windows.Foundation.EventRegistrationToken)
}
```

Summary

- **Imperative code**
  - C#'s roots are imperative
  - CLR and IL provide a VM for imperative code
- **Array initializers**
  - Compile-time data stored in PE file
  - Usage of CompilerHelpers
- **Switch statements**
  - Primitive construct in the CLR using "switch" instruction
  - Tricks to leverage this, e.g. rebasing
  - Efficient lookups for strings
- **Events**
  - Wrappers over delegates
  - Thread-safety is key