# C# 4.0

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

# Objectives

- Optional parameters
- Named Parameters
- Dynamic types for interop
- Dynamic types for parsing
- Co- and Contra- Generic Variance

# The Focus of C# 4.0

- **Each version of C# has a primary focus**
  - C# 2.0 exposed generics and functional constructs
  - C# 3.0 Programmer productivity, LINQ support and rounded out functional constructs
- **C# 4.0 focus is interop**
  - With COM components
  - With dynamic langauges

# Optional Parameters

- **Historically C# has not supported optional parameters**
  - Made COM interop cumbersome: Type.Missing
- **C# 4 introduces optional parameters**
  - Implementor declares default value if parameter not passed
  - Optional parameters must come after required ones

```
double CalculateTax(double amount, double rate = 15.0)
{
    return amount * rate / 100;
}
```

```
double tax = CalculateTax(2000);
```

Rate defaults
to 15.0

# Named Parameters

- **Named parameters allow caller to specify parameters by name rather than by position**
  - Useful when multiple optional parameters and not specifying all of them

```
double CalculateInterest(double initialAmount,
                         double interestRatePercentage = 5.0,
                         int term = 1)
{
    return initialAmount *
           Math.Pow(1 + (interestRatePercentage / 100),
                    term);
}
```

```
double compoundAmount = CalculateInterest(10000, term: 20);
```

# Dynamic Typing

- **C# has traditionally been a statically typed language**
  - All types known at compile time
- **Some languages are dynamically typed**
  - Javascript, Ruby, Python, etc
  - Types resolved at runtime
- **C# 4.0 introduces dynamic typing to C#**
  - Variable types evaluated at runtime
  - Type members not fixed
- **Pivots around dynamic keyword**
  - Dynamic variables assume type of object they are referencing

```
dynamic dyn = 42;
Console.WriteLine(dyn + 7);
dyn = "Hello";
Console.WriteLine(dyn.ToUpper());
```

Prints 49

Prints “HELLO”

# Invoking Methods via Dynamic Variables

- **Methods on dynamic variables must be resolved at runtime**
  - On normal types performs a runtime lookup via reflection
  - Uses functionality in the Dynamic Language Runtime (DLR) for call site caching
  - Dynamic languages present their own functionality through the DLR
- **Slower than normal invocation**
- **No intellisense**

# Dynamic Typing and Interop

- **Dynamic typing makes COM interop simpler**
  - Often COM types returned look like object in RCW
  - Interface prior to 4.0 clumsy
  - Dynamic typing and optional params make code more natural

```
Application app = new Application();
Workbook wb = app.Workbooks.Add(Type.Missing);
Worksheet ws = (Worksheet)wb.Worksheets[1];
Range r = ws.get_Range("A1", Type.Missing);
r.Value2 = 10;
```

```
Application app = new Application();
Workbook wb = app.Workbooks.Add();
dynamic ws = wb.Worksheets[1];
Range r = ws.Range("A1");
r.Value = 10;
```

# Leveraging Dynamic Typing

- **Can create types that support dynamic functionality**
  - Dynamically add methods and properties
- **Powerful model for parsing hierarchical dynamic data**
  - XML
  - FileSystem
  - Registry
- **Derive class from DynamicObject**

# Adding Members to Dynamic Types

- **DynamicObject has virtual members for unknown methods**
  - TryGetMember
  - TrySetMember
  - TryInvokeMember

```
class SimpleDyn : DynamicObject
{
  public override bool TryGetMember
                   (GetMemberBinder binder, out object result)
  {
    if (binder.Name == "MagicNumber")
    {
       result = 42;
       return true;
    }
    return base.TryGetMember(binder, out result);
  }
}
```

# Invoking DynamicObject Type

- **Use a dynamic reference**
  - DynamicObject members invoked at runtime

```
dynamic d = new SimpleDyn();

int i = d.MagicNumber;
```

```
public override bool TryGetMember
             (GetMemberBinder binder, out object result){
  if (binder.Name == "MagicNumber"){
     result = 42;
     return true;
  }
  return false;
}
```

# Generic Variance

- **Generic variance means being able to use derived or base classes in place of the actual generic types**
- **Since introduction generics have not supported variance**
  - No way for compiler to check that use of  derived or base is safe

```
List<string> names = new List<string>(){ ... };

List<object> objs = names;
```

✗

Through this reference
the code could add
integers to the list

# Some Forms of Variance are Safe

- **There are constructs for which variance is safe**
  - If the generic construct **only returns the type** then can assign to a base version in its place – known as co-variance
  - If the generic construct only takes the type inbound then can use a derived version in its place – known as contra-variance

```
List<string> names = new List<string>(){ ... };

IEnumerable<object> objs = names;
```

IEnumerable<T> is a readonly interface only returning T in the Current property so this construct is safe

# How Does the Compiler Know?

- **How does the compiler know that a generic is safe for co- or contra- variance?**
  - Does not try to infer from type members
- **Type author annotates the construct to specify variance support**
  - Type parameters annotated with **out for co-variance** and **in for contra-variance**

```
public interface IEnumerable<out T> : IEnumerable { ... }
```

```
public interface IComparable<in T> { ... }
```

```
public delegate TResult Func<in T, out TResult>(T arg);
```

# The Compiler Keeps You Honest

- **Compiler error if you specify variance and construct does not match**

```
interface ICombinable<in T>
{
    void Add(T item);
    T Result { get; }
}
```

```
error CS1961: Invalid variance: The type parameter 'T' must
be covariantly valid on 'Variance.ICombinable<T>.Result'.
'T' is contravariant.
```

# C# 5 async,await

- **Simplifies the writing of continuations**
  - Orchestrates concurrency
  - Compiler builds state machine

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
   calcButton.IsEnabled = false;
   Task<double> piResult = CalcPiAsync(1000000000);

    // If piResult not ready returns, allowing UI to continue
    await piResult;
   // piResult now available continues to run on UI thread

    calcButton.IsEnabled = true;
    this.pi.Text = piResult.Result.ToString();
 }
```

# Summary

- **C# 4.0 introduces extra support for interop**
  - Optional and named parameters
  - Dynamic typing
- **Dynamic typing can be useful for parsing data**
- **Variance support makes using generics more flexible**