DOTNEXT

Runtime code generation techniques in real life scenarios

Raffaele Rialdi
Senior Software Architect
Microsoft MVP

@raffaeler
https://github.com/raffaeler
http://iamraf.net

Abstract

- The runtime code generation is a powerful practice that many developers are still reluctant to use. A typical, useful and simple point to start exploring the Expression Trees library is building a predicate (filter) or a math calculation. But Expression Trees is not the only library available.
- More recently the .NET world gained the ability to generate code using the compiler itself via APIs exposed by the Roslyn/CodeAnalisys libraries which allows, among many other things, parsing, visiting and generating the source code.
- During the session we will analyze practical use-cases, sometimes improving dramatically the performance of the app which will bring to the final dilemma of debugging the generated code, a task that is fundamental in real-life projects.

Who am I?

- Raffaele Rialdi, Senior Software Architect in Vevy Europe – Italy
  @raffaeler also known as "Raf"
- Consultant in many industries
  Manufacturing, racing, healthcare, financial, …
- Speaker and Trainer around the globe (development and security)
  Italy, Romania, Bulgaria, Russia (CodeFest @ Novosibirsk), USA, …
- And proud member of the great Microsoft MVP family since 2003

Writing code, that will generate the code at the right time

- Why avoiding Reflection (in hot paths)?
  - Slow because of reading ECMA-335 metadata and building Reflection artifacts
  - The code that can't make speculations is harder, slower and power consuming

- When should we generate the code?
  - As soon as we have the information to reduce the code complexity

- What I possibly don't know at compile time?
  - Linq predicates, formulas, Types loaded from plugins, transformation functions (projections), user choices, interop code, DTOs, data modeled via DSL, …

Why generating code at runtime?

- After all, we have reflection … BUT
  - Reflection is slow because of the need to read and interpret the metadata
  - Reflection objects are not automatically cached
  - Creating a generic algorithm is hard, slower and consume CPU/power

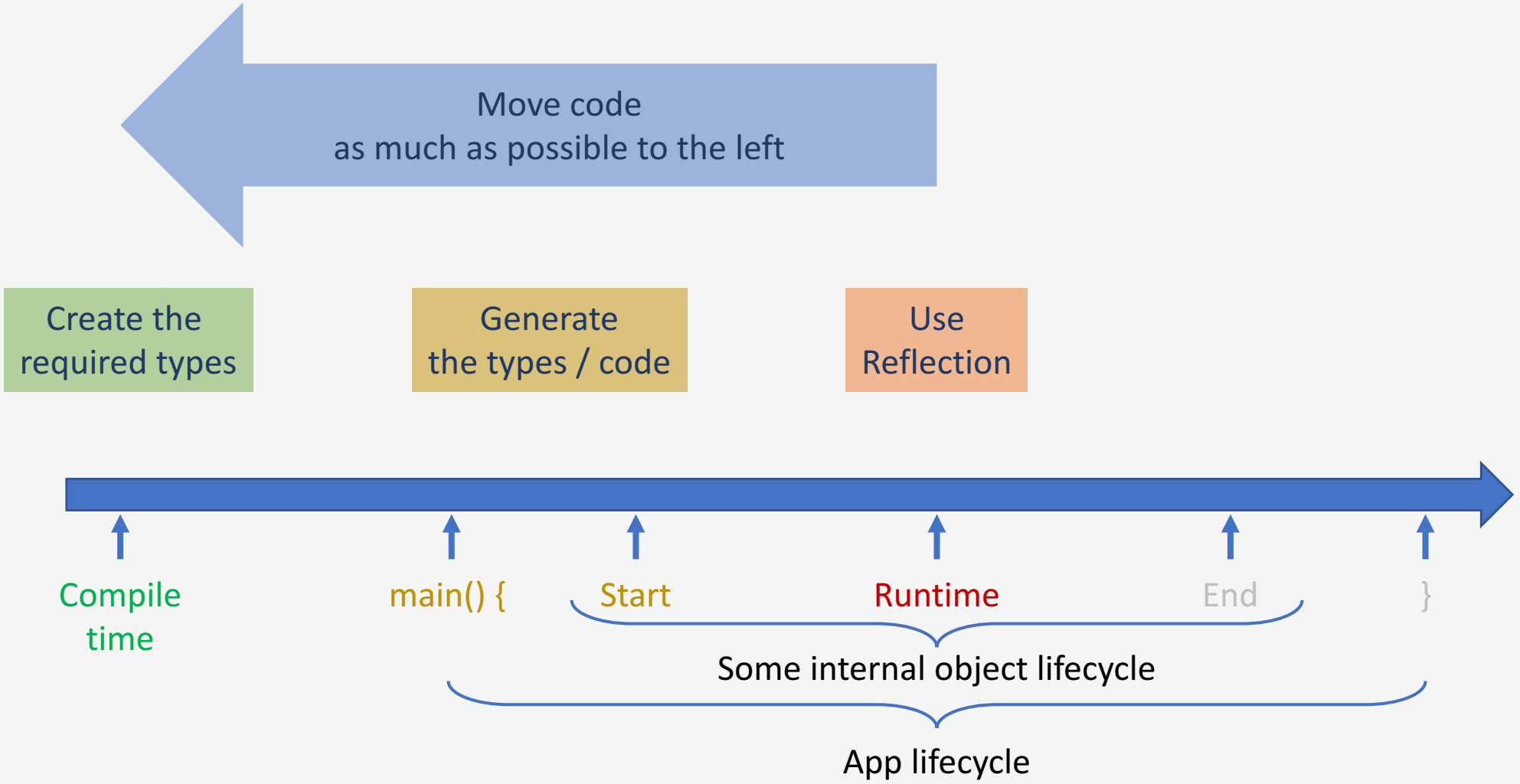
- We can write the code, which will write the code at the right time
  - At a certain point at runtime, we will have enough info to reduce the algorithm complexity

- What I possibly still don't know at compile time?
  - Linq filters/predicates, computational formulas, Types loaded from a plugin, data transformation function (projections), user choices, interop code, DTOs, data modeled via DSL, …

Code "Introspection"

- It is a precious feature available in .NET as well as in Java
  - C++ ISO committee is willing to add it in the specifications
- Leverage metadata to understand the Types 'shape'
- Offered as an API in .NET as "Reflection" (System.Reflection)
  - Another API is available in a "raw" form in "System.Reflection.Metadata"
- Reflection does not offer any "cache" mechanism
  - Reading a Type metadata can be very expensive
  - It is still fundamental during the code generation phase

Code Generation: complexity vs performance



Options to generate code

- Legacy / old-school options
  - T4 templates
  - CodeDom
  - IL Emit
  - Composing textual code by hand

- The most interesting
  - Expression Trees
  - Roslyn (.NET Compiler Platform)
  - Mono.Cecil (inject / modify IL)

Why are these libraries less attractive?

- T4 templates
  - Requires a 3rd party runtime, difficult to debug
- CodeDom
  - Old API still working but does not cover all the language features
- IL Emit
  - Total freedom but complex structures are much harder to build
- Composing textual code by hand
  - Very prone to syntax errors

CodeGen with the Expressions

Generating method calls
Example: msg => Console.WriteLine(msg);

```
// using reflection to get the exact overload
var type = typeof(Console);
var methodInfo = type.GetMethod("WriteLine", new Type[] { typeof(string) });

// generate the call method
var message = Expression.Parameter(typeof(string), "msg");
var call = Expression.Call(null, methodInfo, message);

Expression<Action<string>> lambda =
    Expression.Lambda<Action<string>>(call, message);

// compile
Action<string> delegate1 = lambda.Compile();

// invoke
delegate1.Invoke("hello, world");
```

typed lambda

Finding the exact overload in Reflection

- There are NO APIs to distinguish these two overloads

```
IQueryable<TSource> Where<TSource>(this IQueryable<TSource> source,
                                   Expression<Func<TSource, bool>> predicate)

IQueryable<TSource> Where<TSource>(this IQueryable<TSource> source,
                                   Expression<Func<TSource, int, bool>> predicate)
```

- Solution:

```
public static MethodInfo Where2 = GenericMethodOf(_ =>
  Queryable.Where<int>(default(IQueryable<int>), default(Expression<Func<int, bool>>)));
```

```
private static MethodInfo GenericMethodOf<T>(Expression<Func<object, T>> expression) =>
  GenericMethodOf(expression as Expression);

private static MethodInfo GenericMethodOf(Expression expression) =>
  ((expression as LambdaExpression).Body as MethodCallExpression)
  .Method.GetGenericMethodDefinition();
```

Generating predicates
Example: x => x > -10

```
public Expression<Func<int, bool>> CreatePredicate()
{
  var left = Expression.Parameter(typeof(int), "x");
  var right = Expression.MakeUnary(ExpressionType.Negate,
                Expression.Constant(10), typeof(int));

  var f = Expression.MakeBinary(ExpressionType.GreaterThan, left, right);

  return Expression.Lambda<Func<int, bool>>(f, left);
}
```

Expression visitors

- Many use cases:
  - Modify a Linq predicate / query
  - Block some user code using a whitelist of callable methods
  - Replace the data source of a query
  - Replace variables with constants to simplify the query

```
public class WhereExtractor<T> : ExpressionVisitor
{
    protected override Expression VisitMethodCall(MethodCallExpression node)
    {
        if (node.Method.Name == "Where")
        {
                ...
```

Creating, visualizing and debugging Expression Trees

Generate code with Roslyn

Roslyn (.NET Compiler Platform)

- The compiler exposing APIs! (code generation is just a small part)
  - Compile code, semantic API, symbol API, formatting code, colorize code
  - Intellisense and code completion too!
- Cover the whole set of language features
- The API is not as type safe as the Expression Trees
  - Syntax Nodes can be composed to produce code that does not compile
- Powerful compilation APIs (assemblies can be saved in memory)
- Parser APIs convert text to syntax nodes with full fidelity
- Code visitor to change code instead of generating it from scratch

Parsing and visiting

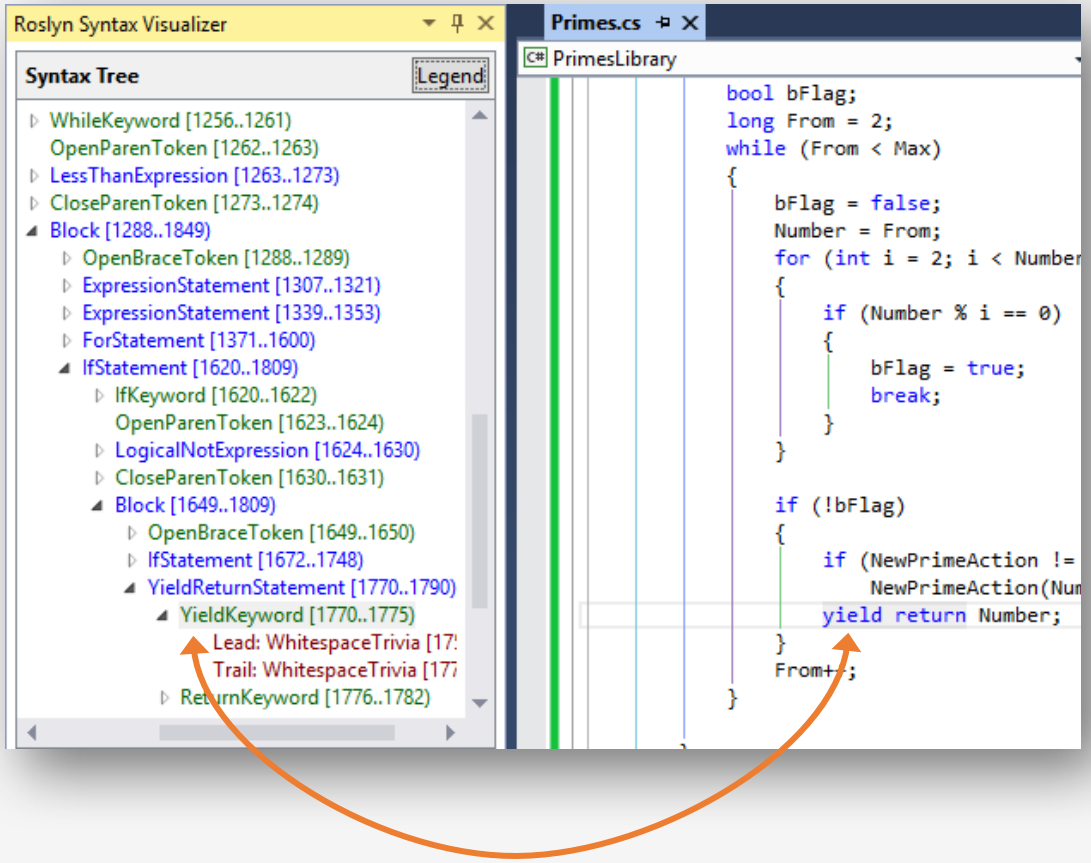
- You can start parsing an existing source code (text) and then …
  - Use a Linq-style syntax to find and replace the desired nodes
  - Use a CSharpSyntaxWalker derived class to find nodes and make changes

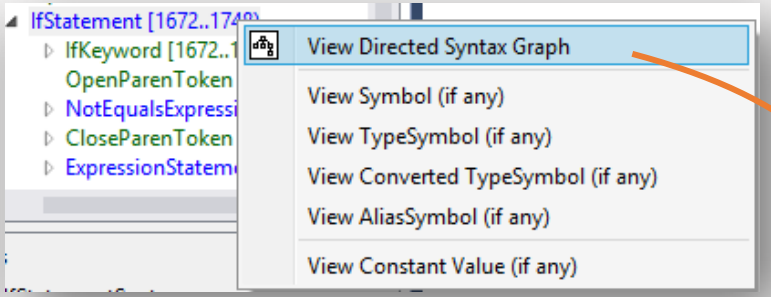
```
var root = CSharpSyntaxTree.ParseText(@"
using System;
using System.Collections.Generic;
namespace RafNamespace{
class RafClass{
void main(){Console.WriteLine(""Hello, world"");
}}}
").GetRoot();

// Format the code "nicely" (like Visual Studio does)
var wspace = new AdhocWorkspace();
var formatted = Formatter.Format(root.ToString(), wspace, wspace.Options);
return formatted.ToString();
```

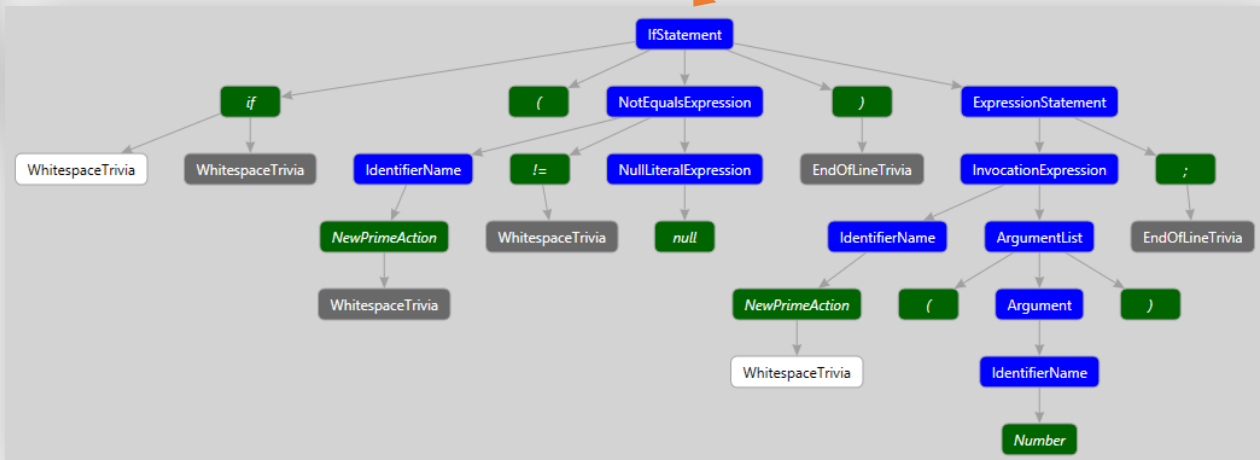
SyntaxGenerator in action

- Is a Roslyn class providing high-level services to generate SyntaxNodes
- Frequently used methods
  - Get a syntax node for a type
    - var node = SyntaxFactory.ParseTypeName("string")
  - Create a node from a literal
    - var node = generator.IdentifierName(literalName)
  - Declaring a class
    - var node = generator.ClassDeclaration(name, genericParameters, accessibility, …);
  - Create a new object
    - var node = generator.ObjectCreationExpression(type, arguments);
  - «adjust» the space among nodes with SyntaxNode extension method
    - node.NormalizeWhitespace();

Visual Studio Syntax Tree tool







Building a class with the SyntaxGenerator

```
var nodes = new List<SyntaxNode>();
nodes.AddRange(Usings.Select(n =>
_generator.NamespaceImportDeclaration(n).NormalizeWhitespace()));

nodes.Add(_generator.NamespaceDeclaration(Namespace,
    _generator.ClassDeclaration(Name, null, Accessibility.Public,
           DeclarationModifiers.Partial, null, null,
           Properties.Select(kvp =>
              CreateProperty(kvp.Key, kvp.Value)))).NormalizeWhitespace());

var root = _generator.CompilationUnit(nodes).NormalizeWhitespace();


// optional step (not needed in this case)
var formatted = Formatter.Format(root, _workspace, _workspace.Options);
var sourceCode = formatted.ToString();
```

Creating a trivial property

```
public string Name { get; set; }
```

trivial

```
private SyntaxNode CreateProperty(string name, SyntaxNode type)
{
    var propertyDeclaration = _generator.PropertyDeclaration(name, type,
        Accessibility.Public, DeclarationModifiers.None);

    var getAccessor = _generator.GetAccessor(
       propertyDeclaration, DeclarationKind.GetAccessor);
    var simpleGetAccessor = _generator.WithStatements(getAccessor, null);
    propertyDeclaration = _generator.ReplaceNode(
       propertyDeclaration, getAccessor, simpleGetAccessor);

    var setAccessor = _generator.GetAccessor(
       propertyDeclaration, DeclarationKind.SetAccessor);
    var simpleSetAccessor = _generator.WithStatements(setAccessor, null);
    propertyDeclaration = _generator.ReplaceNode(
       propertyDeclaration, setAccessor, simpleSetAccessor);

    return propertyDeclaration.NormalizeWhitespace();
}
```

SyntaxGenerator in action

Going down to IL

What about generating IL code?

- In my experience it fits better for "patching" / modifying existing code
- Mono.Cecil is your friend: Decompile, Add and Remove IL code

Searching IL calls

```
var processor = method.Body.GetILProcessor();
var callsOpcodes = processor.Body.Instructions
    .Where(i => i.OpCode == OpCodes.Call ||
        i.OpCode == OpCodes.Callvirt)
        .ToList();
```

Creating new instructions

```
var premsg = processor.Create(OpCodes.Ldstr, $"{before}");
var postmsg = processor.Create(OpCodes.Ldstr, $"{after}");
var externalCall = processor.Create(OpCodes.Call, method);
```

Injection

```
processor.InsertBefore(current, premsg);
processor.InsertBefore(current, externalCall);
```

A vision to the future: PicoLibraries

- This is not (yet) a thing, just an idea and some code around
- My vision is creating building blocks to be orchestrated by generated code
- PicoLibraries are designed to avoid as much of boilerplate code as possible
- A set of PicoLibraries is a DSL that can be assembled by a designer or by an automatic Artificial Intelligence process
- It is a way to promote autonomous systems

To sum up

- Use code generation to free the "hot paths"
- Get the most out Roslyn and Expressions by mixing them
  - Roslyn is the choice to **create** new Types or **modify** existing source code
  - Expressions are good to enforce the **validation** of the assembled Expressions
- Leverage IL only when you need to intervene after the compilation

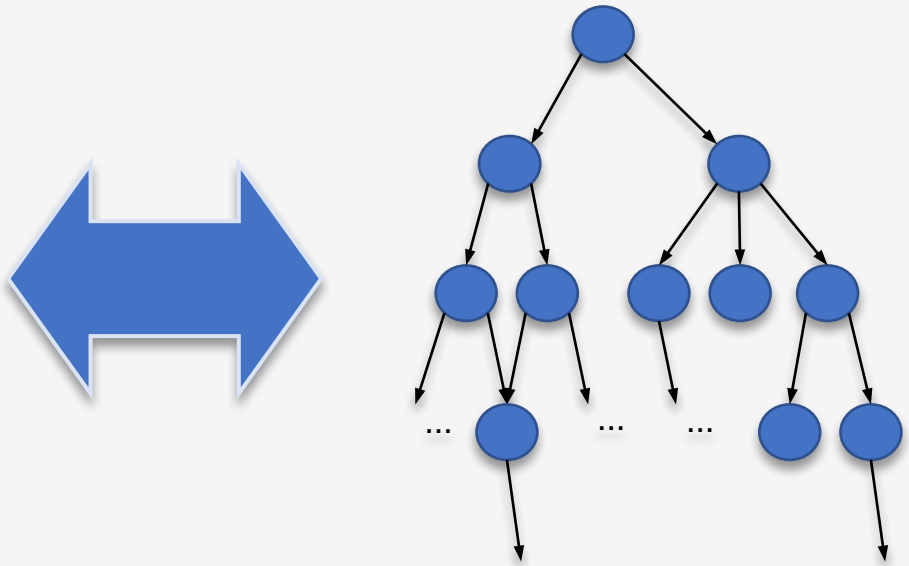
Questions @ booth #1

```
var compilation = CSharpCompilation.Create(
assemblyName, trees, references,
new CSharpCompilationOptions(
OutputKind.DynamicallyLinkedLibrary));
```

Thank you!

Parsing the sources

```
public override string GetTextForToken(SyntaxToken syntaxToken, bool isStartOfLine)
{
    var text = System.Net.WebUtility.HtmlEncode(syntaxToken.Text);
    var kind = syntaxToken.CSharpKind();
    var parent = syntaxToken.Parent;

    var prefix = string.Empty;
    if (isStartOfLine)
        prefix = "<div>";

    string format;
    if (_formats.TryGetValue(kind, out format))
        return string.Format(format, text);

    if (syntaxToken.IsKeyword())
    {
        if (!string.IsNullOrEmpty(_formatKeyword))
            return prefix + string.Format(_formatKeyword, text);
        else
            return prefix + text;
    }

    if (kind == SyntaxKind.IdentifierToken)
    {
        var parentKind = parent.CSharpKind();
        return prefix + GetTextForIdentifierToken(text, parentKind, parent);
    }
    return prefix + text;
}
```



- Transformation from text to nodes is reversible
  - Full fidelity (comments and alignments as well)
- Every syntax tree is immutable and thread-safe
  - Allow multiple consumers to access concurrently a single tree

From sources to syntax tree

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(source);
SyntaxNode node = tree.GetRoot();
Debug.Assert(node.CSharpKind() == SyntaxKind.CompilationUnit);
var root = (CompilationUnitSyntax)node;
```

- Syntax API is represented with these classes:
  - SyntaxTree is the binary form of the source with full fidelity
  - SyntaxNode represents declarations, statements, clauses and expressions
  - SyntaxToken identifies the special language tokens like keywords, identifiers, operators or punctuation
    - They never have children
  - SyntaxTrivia everything not impacting on the generated IL code such as whitespaces, comments, etc.