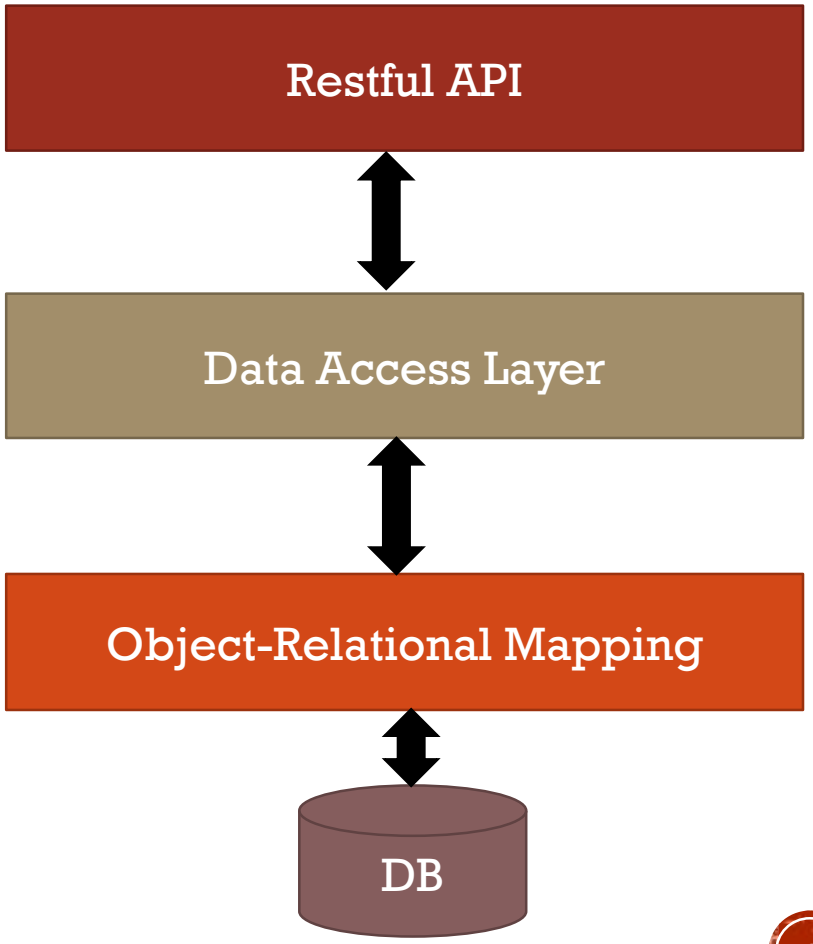
RAWDATA SECTION 2

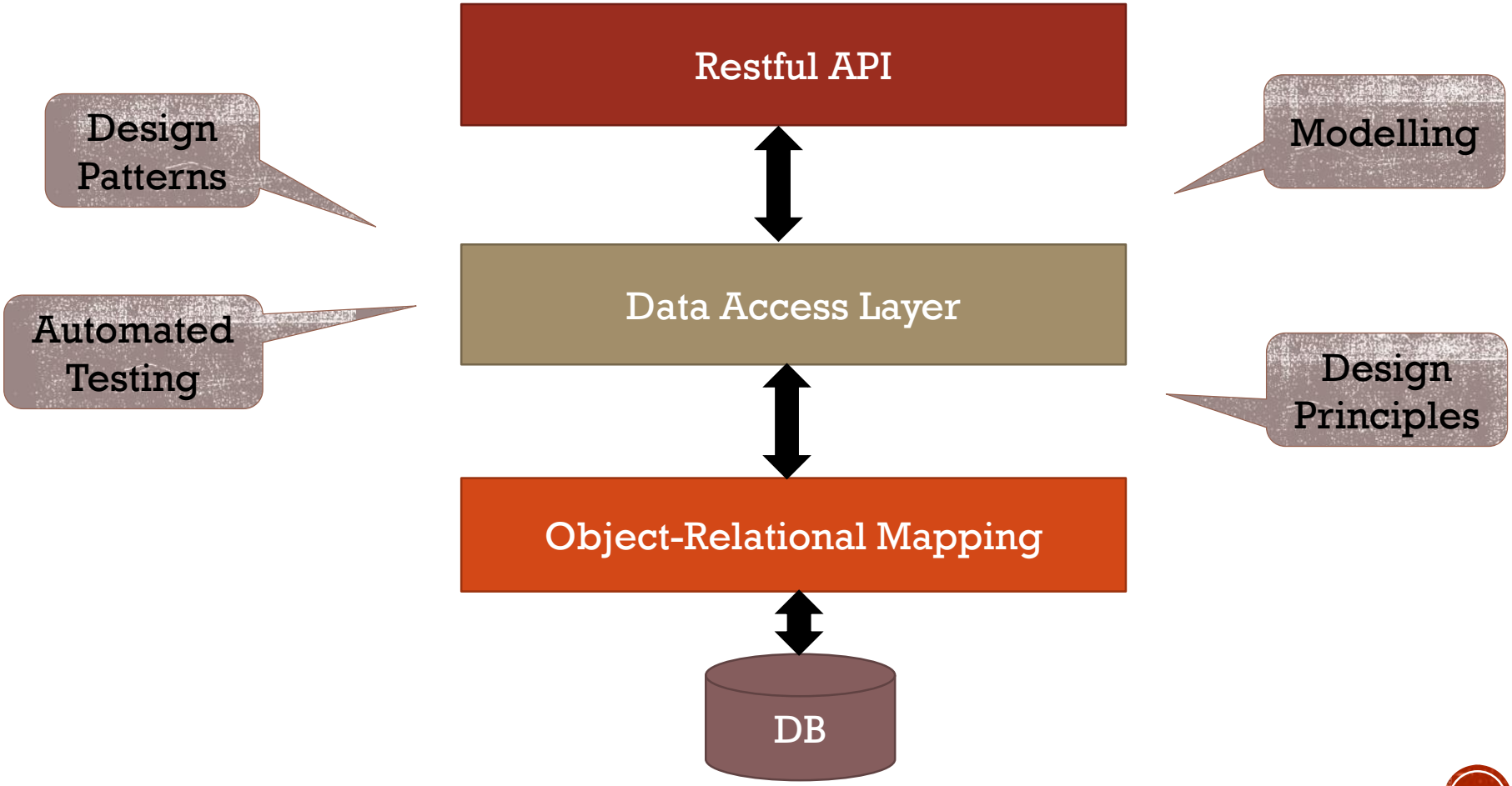
Troels Andreasen
& Henrik Bulskov

WHAT TO DO IN SECTION 2?

- C#
  - Lambda
  - Linq
  - Entity Framework
- Network
  - HTTP protocol
  - Sockets
- ASP.NET
  - Restful Web services



SYSTEM DEVELOPMENT



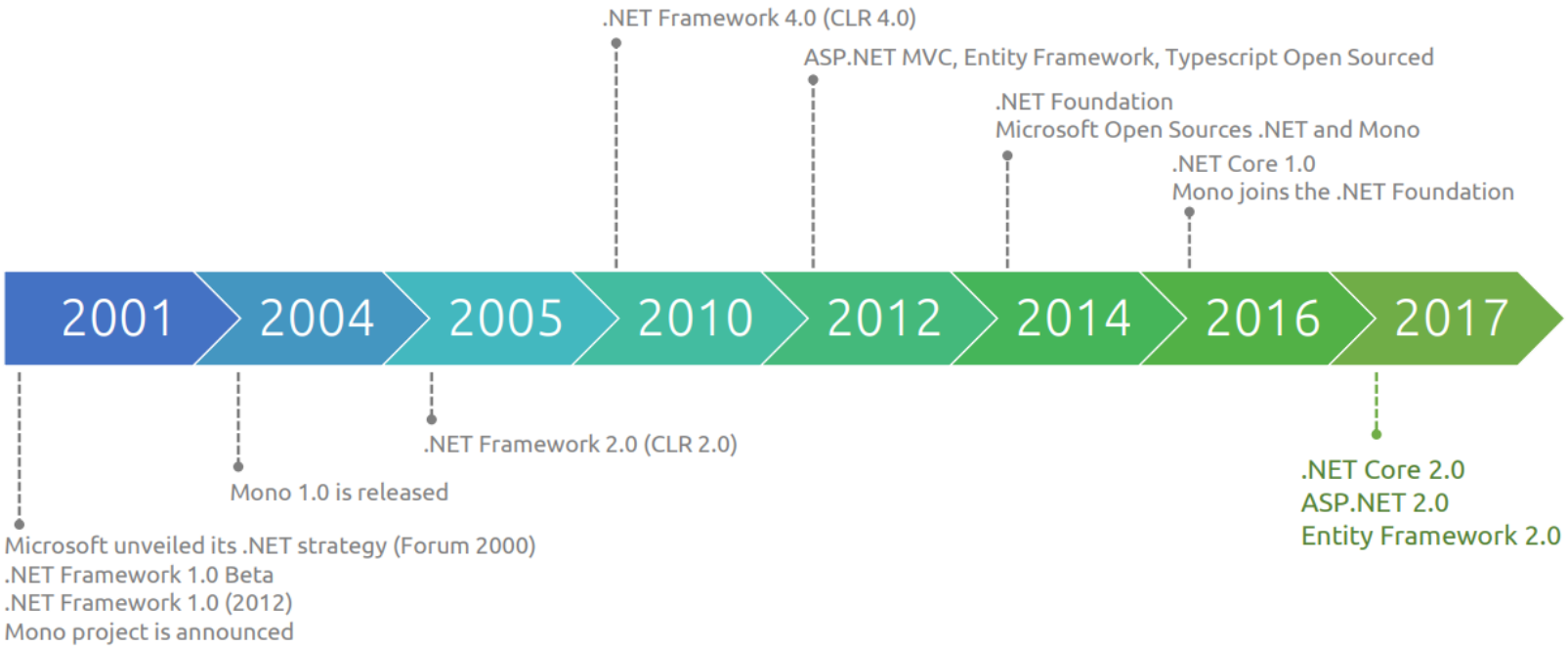
DEVELOPMENT TOOLS

- Visual Studio
- Visual Studio for Mac
- Visual Studio Code

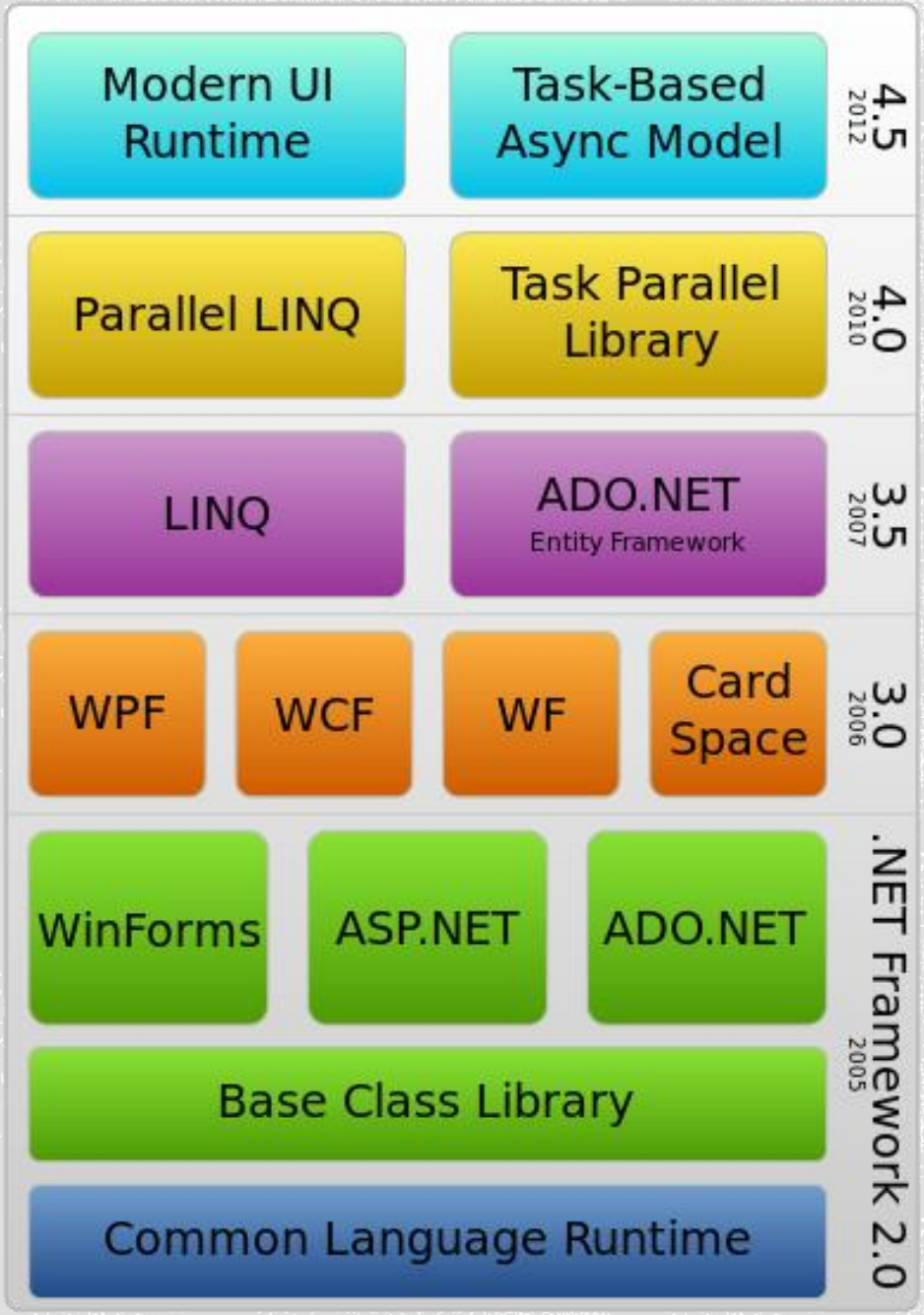
VISUAL STUDIO 2017

- Download:
  - https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx
  - For the enterprice version on Windows go to Søren Døygaard to get a Imagin account
- Additional Tools(only Visual Studio):
  - Resharper (https://www.jetbrains.com/resharper/) You can get a student license if you register with your RUC mail. https://www.jetbrains.com/student/

A BRIEF INTRODUCTION



HISTORY OF A FRAMEWORK

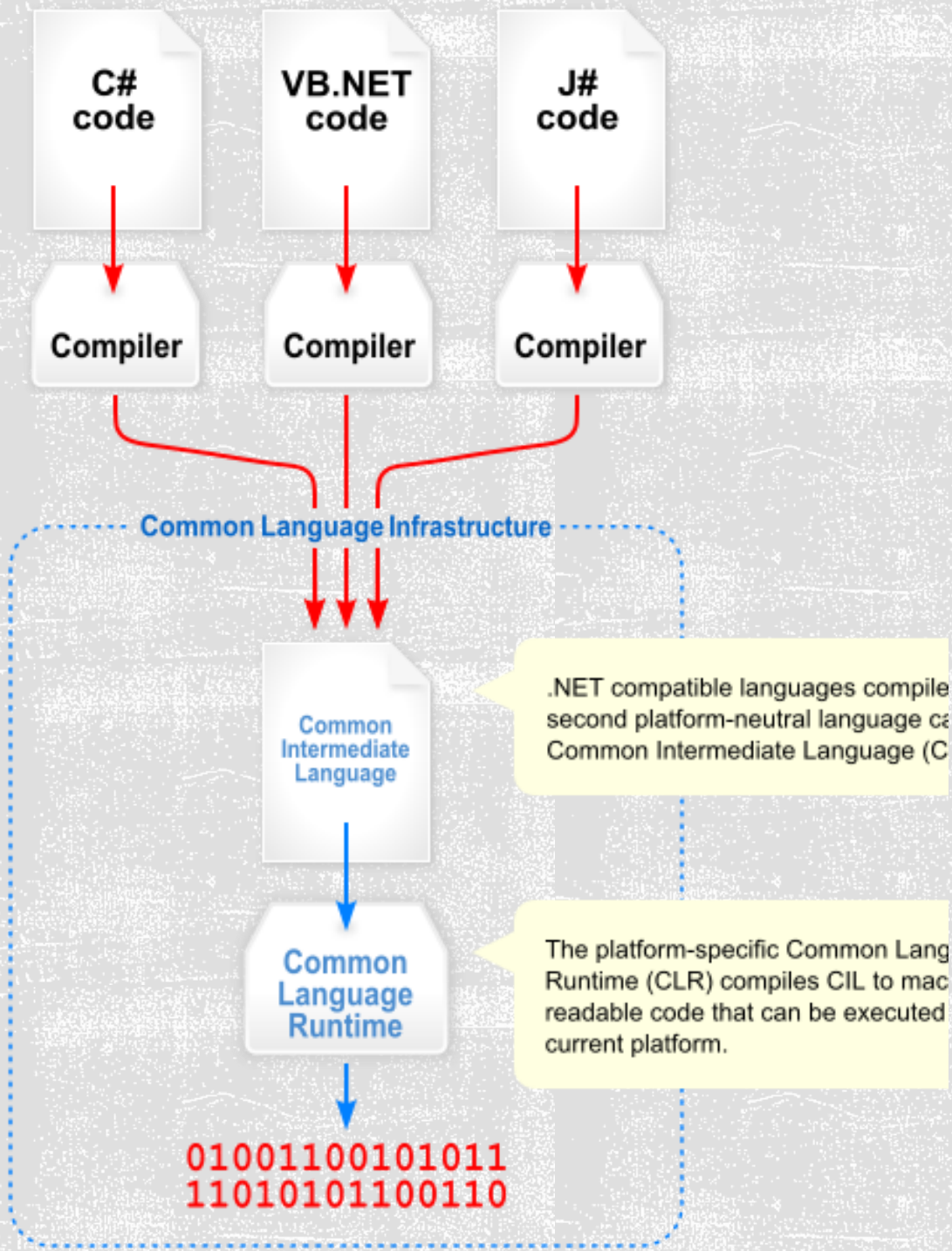


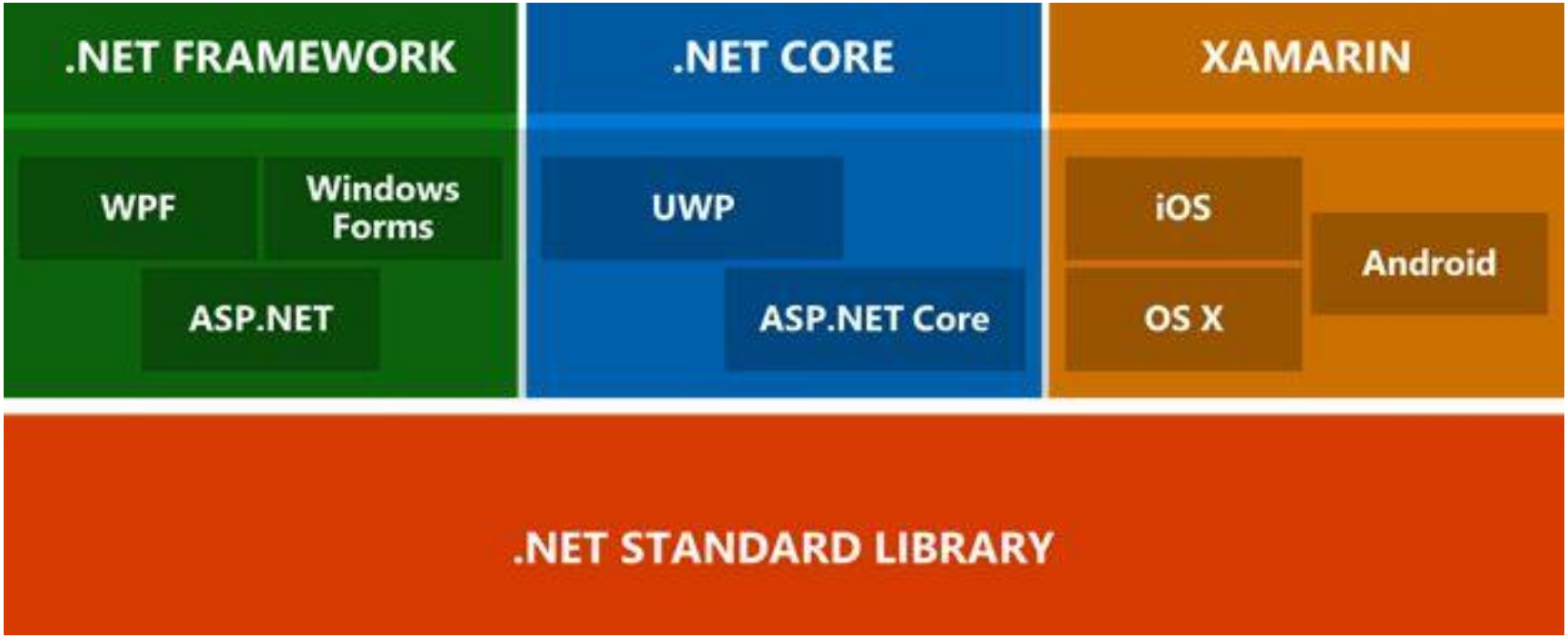
The .NET Framework Stack

.NET FRAMEWORK

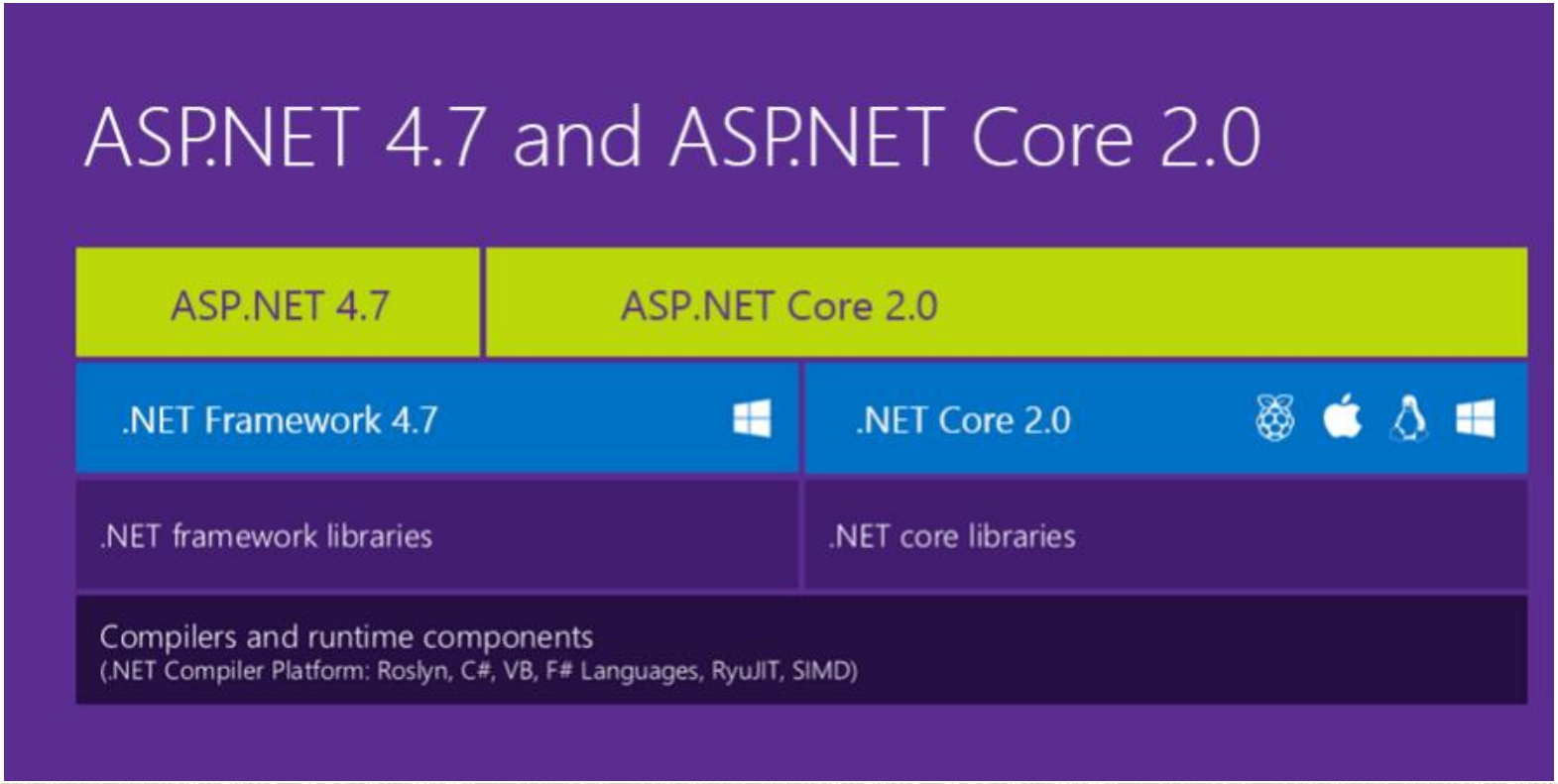
- Versions
  - 3.0 (2006)
  - 3.5 (2007) VS2008
  - 4.0 (2010) VS2010
  - 4.5 (2012) VS2012
  - 4.5.1-2 (2013) VS2013
  - 4.6 (2015) VS2015
  - 4.7 (2017) VS2017

.NET FRAMEWORK





ENTER .NET CORE



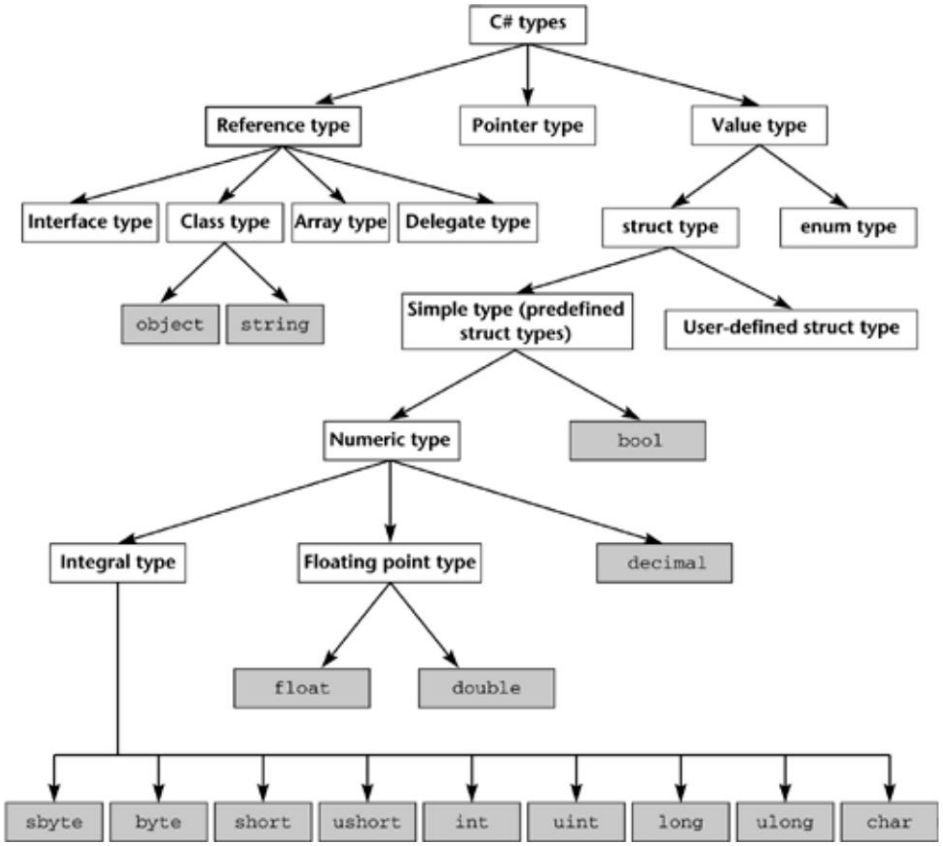
ASP.NET CORE HIGH-LEVEL OVERVIEW

LANGUAGE BASICS

C# NAMING CONVENTIONS

- Composed names
  - currentLayout, CurrentLayout
- Variable and private fields
  - vehicle or **_vehicle, _leftElement**
- Methods
  - CurrentVehicle(), Size()
- Properties
  - Pi, Name, Size
- Classes
  - MyClass, List
- Interfaces
  - IException, IObserver

http://msdn.microsoft.com/en-us/library/ms229002.aspx


C# types
Reference type
Pointer type
Value type
Interface type
Class type
Array type
Delegate type
struct type
enum type
object
string
Simple type (predefined struct types)
User-defined struct type
Numeric type
bool
Integral type
Floating point type
decimal
float
double
sbyte
byte
short
ushort
int
uint
long
ulong
char

C# TYPES

REFERENCE TYPE EQUALITY

- System.Object.ReferenceEquality:

  Person p1 = new Person("Joe");

  Person p2 = new Person("Joe");

  Person p3 = p2;

  Object.ReferenceEquality(p1, p2) = false

  Object.ReferenceEquality(p2, p3) = true;

- In C# the == operator is "equal" to reference equality. (Can be overridden)
- Value Equality (for reference types)

  p1.Equals(p2) = true; (Can be overridden)

VALUE TYPE EQUALITY

- Equals the same as for reference types
- Object.ReferenceEquality will always return false for value types
- == operator is overridden so it does value equality

NULLABLE TYPES

- Reference types can represent a nonexistent value with a null reference. Value types, however, cannot ordinarily represent null values. For example:

```
string s = null;        // OK, Reference Type
int i = null;           // Compile Error, Value Type cannot be null
```

- Nullab

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);           // True


int? i = null;
Console.WriteLine (i == null);              // True
```

NULL OPERATORS

- Null-coalescing operator ??

```
string s1 = null;
string s2 = s1 ?? "nothing";   // s2 evaluates to "nothing"
```

- Null-conditional operator .?

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString();  // No error; s instead evaluates to null
```

The last line is equivalent to:

```
string s = (sb == null ? null : sb.ToString());
```


COMPILE TIME INFERRED TYPES

once the type is assigned it cannot be changed. E.g this gives a compile error:

```
var v1 =13;
v1= "Hello" ; // error
v1 = 23.6;  // error
```

```
var identifier = expression
```

var is a keyword, not a type

```
var age = 25;           ==> inferred type int
var flag = false;       ==> inferred type bool


Object[] objects = { 3,"hello",5.6 };
foreach(var val in objects)
{
    Console.Out.WriteLine(val); //Compiles and runs OK
}
```

RUNTIME INFERRED TYPES

```
dynamic identifier
```

dynamic is a type not a keyword like var

```
class Phone{
   public readonly String name;
   public readonly int phone;
   public Phone(String name, int phone){...}
}
```

```
dynamic d1 = 31;
int i1 = d1 * 2;                        ==> cast (int)d1 at runtime
bool b1 = d1;                           ==> Compiles OK; throws run-time exception
dynamic p1 = new Phone("Dario", 123);   ==> Compiles and runs OK
String name = p1.name;                  ==> Field access checked at runtime
int n2 = p1.age;                        ==> Compiles OK; throws run-time exception
```

ENUMERATION

```
enum-modifiers enum t base-clause
{
   enum-member-list
}
```

```
public enum Day {Mon,uint  Wed, Thu, Fri, Sat, Sun}

public enum Month: unit {Jan=1, Feb, Mar, Jun, Jul, Aug, Sep, Oct, Nov, Dec}

public enum Color: uint {Red = 0xFF0000, Green = 0x00FF00, Blue = 0x0000FF}



Vehicle car = new Vehicle();
car.SetColor(Color.Red);

Console.Out.WriteLine(Day.Mon);        ==> Mon
```

CONDITIONAL STATEMENTS

```
if (condition)
   true branch
```

```
if (condition)
   true branch
else
   false branch
```

```
switch (expression) {
   case constant 1: branch 1
   case constant 2: branch 2
   ...
   default: default branch
}
```

Fall-through is not allowed, but goto ☺

LOOP STATEMENTS

```
for (initialization; condition; step)
   body
```

```
while (condition)
   body
```

```
foreach (type identified in expression)
   body
```

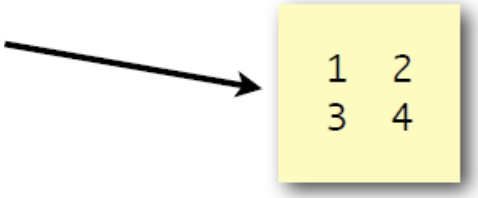
```
do
   body
while (condition)
```

```
int[] values = {1,2,3,4};

foreach (int value in values){
   Console.Out.WriteLine(value);
}
```

QUICK PICK- ARRAY

```
int[] intArray = new int[4];

double[,] doubleArray = new double[4,5];

int[,] array1 = {{1,2},{3,4}};

int value1 = intArray[0];

double value2 = doubleArray[0,1];

Console.Out.WriteLine(array1[1,1]);
```

default values are assigned automatically



Arrays are 0 based, thus the console print out would be the number 4.

AGENDA

- Class declarations
- User defined operators
- Inheritance
- Extension methods
- Partial classes

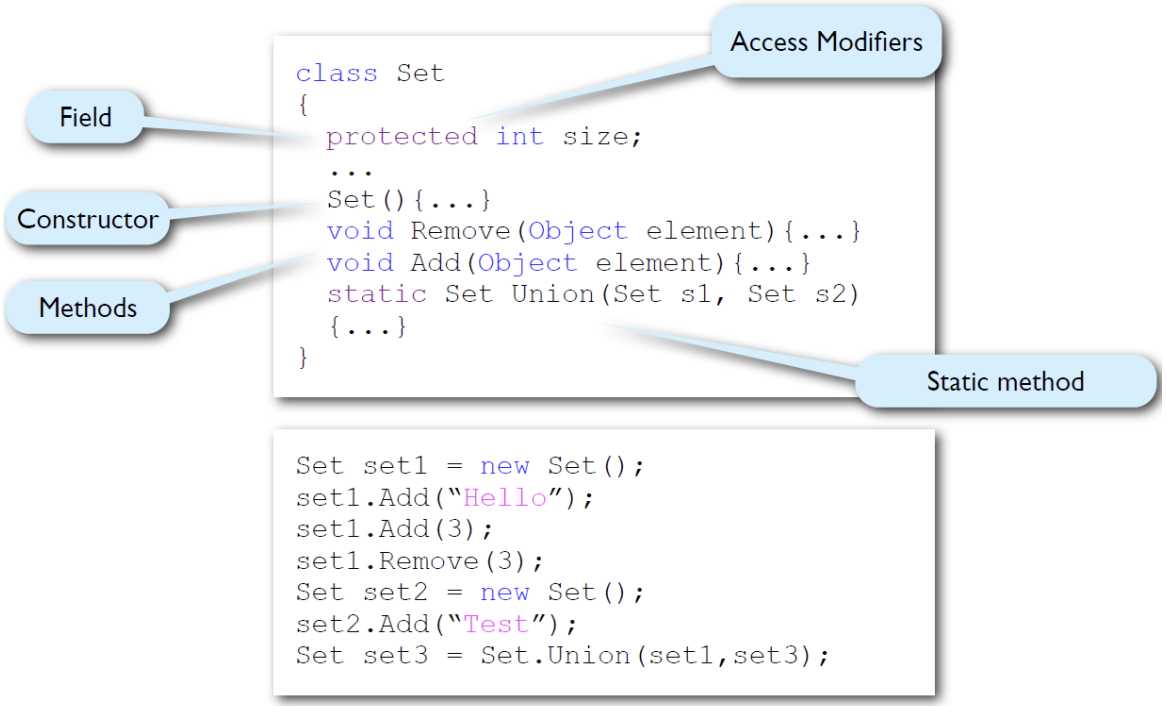
CLASS VS STRUCT

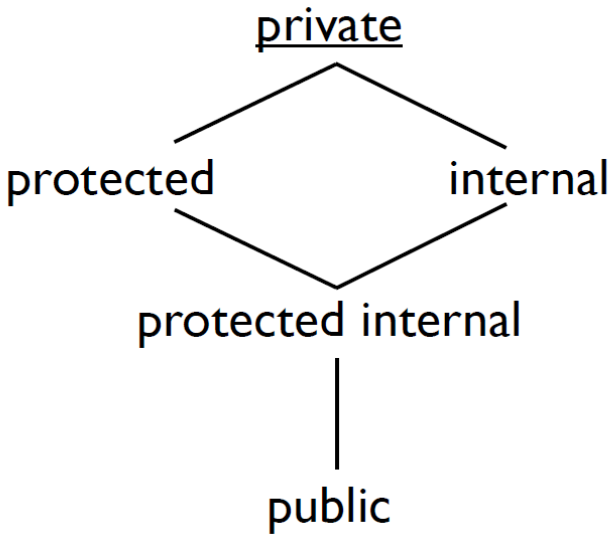
```
struct Faction
{
   public readonly int Nominator;
   public readonly int Denominator;

   Fraction(int n, int d){...}
   static Fraction div(Fraction a, Fraction b){...}
   ...
}
```

- No inheritance
- No polymorphism
- Value type
- No indexers
- Can implement interfaces

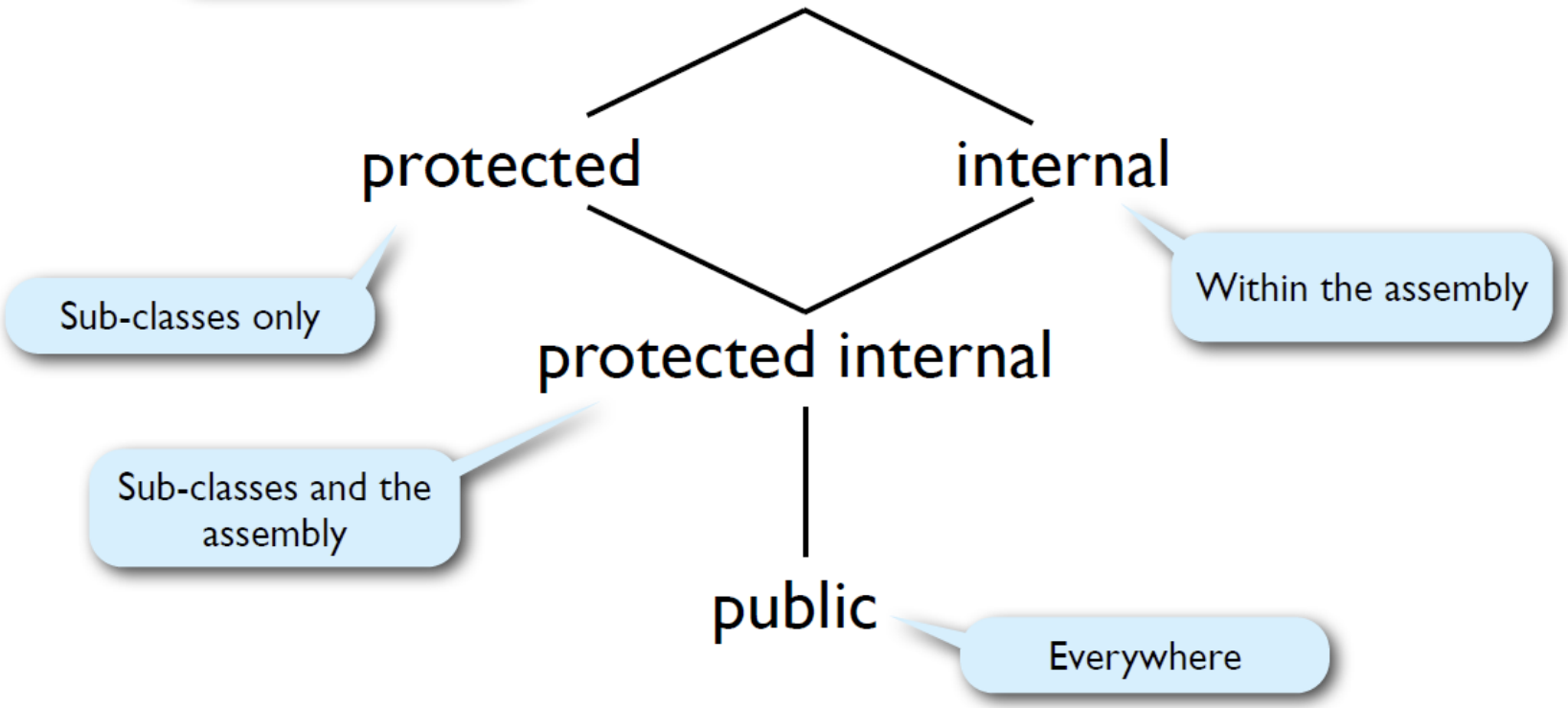
```
class Set
{
   protected int size;
   ...
   Set(){...}
   void Remove(Object element){...}
   void Add(Object element){...}
   static Set Union(Set s1, Set s2){...}
}
```

CLASS


Access Modifiers
Field
Constructor
Methods
Static method
class Set
{
protected int size;
...
Set(){...}
void Remove(Object element){...}
void Add(Object element){...}
static Set Union(Set s1, Set s2)
{...}
}
Set set1 = new Set();
set1.Add("Hello");
set1.Add(3);
set1.Remove(3);
Set set2 = new Set();
set2.Add("Test");
Set set3 = Set.Union(set1,set3);



MEMBER ACCESS MODIFIERS



MEMBER ACCESS MODIFIERS

PARAMETER PASSING

```
class MyClass
{
   public static int Max(int a, int b){...}

   public static void Increment(ref int a)
   {
      a++;
   }

   public static bool LookupUser(string name, out int ID){...}
}
```

```
int index = 0;
MyClass.Increment(index); //compiler error
MyClass.Increment(ref index);// index=1

int ID;
MyClass.LookupUser("Joe",ID);//compiler error
MyClass.LookupUser("Joe",out ID);//ID is now assigned
```

PARAMETER PASSING

- Implement a Max function for *n* parameters

```
public int Max(int v1, int v2){...}

public int Max(int v1, int v2, int v3){...}

public int Max(int v1, int v2, int v3, int v4){...}

public int Max(int v1, int v2, int v3, int v4, int v5){...}

.....????
```

```
class Math
{
   public static int Max(int v1, params int[] vals)
   {
      foreach (int val in vals)
         if (val > v1)
            v1 = val;
      return v1;
   }
}

....

int max = Math.Max(3,6,78,3,5);
```

It cannot be `ref` or `out`

Allows an undefined number of parameters. Only once, and at the end

PARAMETER PASSING

Implement a Max function for *n* parameters

PROPERTY DECLARATION

- A property is declared like a field, but with a get/set block added. Here's how to implement CurrentPrice as a property:

```
public class Stock
{
  decimal currentPrice;           // The private "backing" field

  public decimal CurrentPrice     // The public property
  {
    get { return currentPrice; }
    set { currentPrice = value; }
  }
}
```

- Automatic properties

```
public class Stock
{

  ...
  public decimal CurrentPrice { get; set; }
}
```

READ-ONLY, CALCULATED PROPERTIES AND INITIALIZERS

- Read only

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
  get { return currentPrice * sharesOwned; }
}
```

- Expression-bodied (read only) properties

```
public decimal Worth => currentPrice * sharesOwned;
```

- Property initializers

```
public decimal CurrentPrice { get; set; } = 123;
```

Matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

```
class Matrix
{
   ...
   public Matrix(int cols, int rows){...}

   public double GetValue(int col, int row){...}

   public static Matrix Add(Matrix m1, Matrix m2){...}

   public static Matrix Subtract(Matrix m1, Matrix m2){...}

   public static Matrix DotProduct(Matrix m1, Matrix m2){...}

   ...
}
```

INDEXER DECLARATION

INDEXER DECLARATION

Matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

```
indexer-modifiers type this[parameters]
{
   get bet-body;
   set set-body;
}
```

```
class Matrix
{
   ...

   public double GetValue(int col, int row){...}

   public double this[int col, int row]
   {
      get{...}
      set{...}
   }
   ...
}
```

```
Matrix m = new Matrix(3,4);
m[3,4] = 5;
```

OPERATOR OVERLOADING

Matrix $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

It would be nice to use +, - and *

```
class Matrix
{
   ...
   public Matrix(int cols, int rows){...}

   public double this[int col, int row]{...}

   public static Matrix Add(Matrix m1, Matrix m2){...}

   public static Matrix Subtract(Matrix m1, Matrix m2){...}

   public static Matrix DotProduct(Matrix m1, Matrix m2){...}

   ...
}
```

```
class Matrix
{
   ...

   public static Matrix Add(Matrix m1, Matrix m2){...}

   public static Matrix operator +(Matrix m1, Matrix m2){...}


   public static Matrix Subtract(Matrix m1, Matrix m2){...}

   public static Matrix operator -(Matrix m1, Matrix m2){...}


   public static Matrix DotProduct(Matrix m1, Matrix m2){...}

   public static Matrix operator *(Matrix m1, Matrix m2){...}


   ...

}
```

```
Matrix m1 = Matrix(3,3);
Matrix m2 = Matrix(3,3);
....

Matrix m3 = m1 + m2;
Matrix m4 = m1 - m2;
Matrix m5 = m1 * m2;
```

OPERATOR OVERLOADING

OPERATOR OVERLOADING

Operators	Overloadability
+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, |, ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded (but see the note that follows this table).
&&, ||	The conditional logical operators cannot be overloaded, but they are evaluated using **&** and **|**, which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
(T)x	The cast operator cannot be overloaded, but you can define new conversion operators (see explicit and implicit).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment operators cannot be overloaded, but **+=**, for example, is evaluated using **+**, which can be overloaded.
=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	These operators cannot be overloaded.

Note

The comparison operators, if overloaded, must be overloaded in pairs; that is, if **==** is overloaded, **!=** must also be overloaded. The reverse is also true, and similar for **<** and **>**, and for **<=** and **>=**.

http://msdn.microsoft.com/en-us/library/8edha89s.aspx

Private fields and attributes are no accessible, but are inherited

```
class BaseClass
{
   protected int count;
   private int hidden;

   public BaseClass(){...}
}

class SubClass: BaseClass
{
   public SubClass():base()
   {
      count = 3;
   }
}
```

Multiple inheritance is NOT allowed

Calls the base class constructor

INHERITANCE

Abstract

Static

Sealed

CLASS MODIFIERS

INTERFACE DECLARATION

```
public interface IPerson
{
   public string Name
   {
      get;
   }
   ...
}
```

Similar to a contract. It defines methods a class must implement

```
class Person: IPerson
{
   protected string title;
   ...
   public string Name
   {
      get
      {
         return title + " " + name;
      }
   }
   ...
}
```

EXTENSION METHODS

- Extension methods allow an existing type to be extended with new methods without altering the definition of the original type
- An extension method is a static method of a static class, where the this modifier is applied to the first parameter

```
class Text
{
   ...
   public Position Search(string text){...}

   public void SaveAs(FileFormat format, string destination){...}
   ...
}
```

```
static class TextAnnotationExtensions
{
   public static void Highlight(this Text t, Position p1, Position p2){...}

   public static void Strikethrough(this Text t, Position p1, Position p2){...}
}
```

EXTENSION METHODS

```
Position start = article.Search("optimization");
Position end = article.Search("combinatorial");

article.Highlight(start,end);
```

syntactic sugar

```
static class TextAnnotationExtensions
{
   public static void Highlight(this Text t, Position p1, Position p2){...}

   public static void Strikethrough(this Text t, Position p1, Position p2){...}
}
```

EXTENSION METHODS

EXTENSION METHODS

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
```


```
Console.WriteLine ("Perth".IsCapitalized());
```

```
Console.WriteLine (StringHelper.IsCapitalized ("Perth"));
```

PARTIAL CLASSES

```
partial class PartialClass
{
    public PartialClass()
    {}

    public String ToString()
    {
        return "hejsa";
    }
}


partial class PartialClass
{
    public void Print()
    {
        Console.WriteLine("print");
    }
}
```

INSTANTIATION, NAMED PARAMETERS & DEFAULT VALUES

```
class Person {
    public string Name { get; set; }
    public int Age { get; set; }
    public int[] Phones { get; set; }
    public void foo(string name, int length = 5) {…}
}

static void Main() {
    var p = new Person {
                    Name = "Peter", Age = 23,
                    Phones = new int[]{1234, 2345, 3456}
                };
    p.foo(name: "Allan");
    p.foo(length: 7, name: "Allan");
}
```

GIT

A BRIEF HISTORY OF VERSION CONTROL

- First Generation
  - Single-file
  - No networking
  - e.g. SCCS, RCS
- Second Generation
  - Multi-file
  - Centralized
  - e.g. CVS, VSS, SVN, TFS, Perforce
- Third Generation
  - Changesets
  - Distributed
  - e.g. Git, Hg, Bazaar, BitKeeper
- Further reading
  http://www.ericsink.com/vcbe/html/history_of_version_control.h

DVCS TOPOLOGIES

- Different topologies
  - Centralized
    - Developers push changes to one central repository
  - Hierarchical
    - Developers push changes to subsystem-based repositories
    - Sub-system repositories are periodically merged into a main repository
  - Distributed
    - Developers push changes to their own repository
    - Project maintainers pull changes into the official repository
- Backups are easy
  - Each clone is a full backup

ADVANTAGES OF DVCS

- Reliable branching/merging
  - Feature branches
  - Always work under version control
  - Applying fixes to different branches
- Full local history
  - Compute repository statistics
  - Analyze regressions
- New ideas
  - Deployment
    - git push heroku prod_branch

ABOUT GIT

- Created by Linus Torvalds, who also created Linux
- Design goals
  - Speed
  - Simplicity
  - Strong branch/merge support
  - Distributed
  - Scales well for large projects

INSTALLING GIT

- Windows
  - https://git-scm.com/download/win
- Mac OSX
  - brew install git
  - DMG (http://git-scm.com/download/mac)
- Linux
  - apt-get install git-core (Debian/Ubuntu distros)
  - yum install git-core (Fedora distros)
  - For other distros, check your package manager

CONFIGURING GIT

System-level configuration	• git config --system • Stored in /etc/gitconfig or c:\Program Files (x86)\Git\etc\gitconfig
User-level configuration	• git config --global • Stored in ~/.gitconfig or c:\Users\<NAME>\.gitconfig
Repository-level	• configuration git config • Stored in .git/config in each repo

Creating a local repository

Adding files

Commiting changes

Viewing history

Viewing a diff

Working copy, staging, and repository

Deleting files

Cleaning the working copy

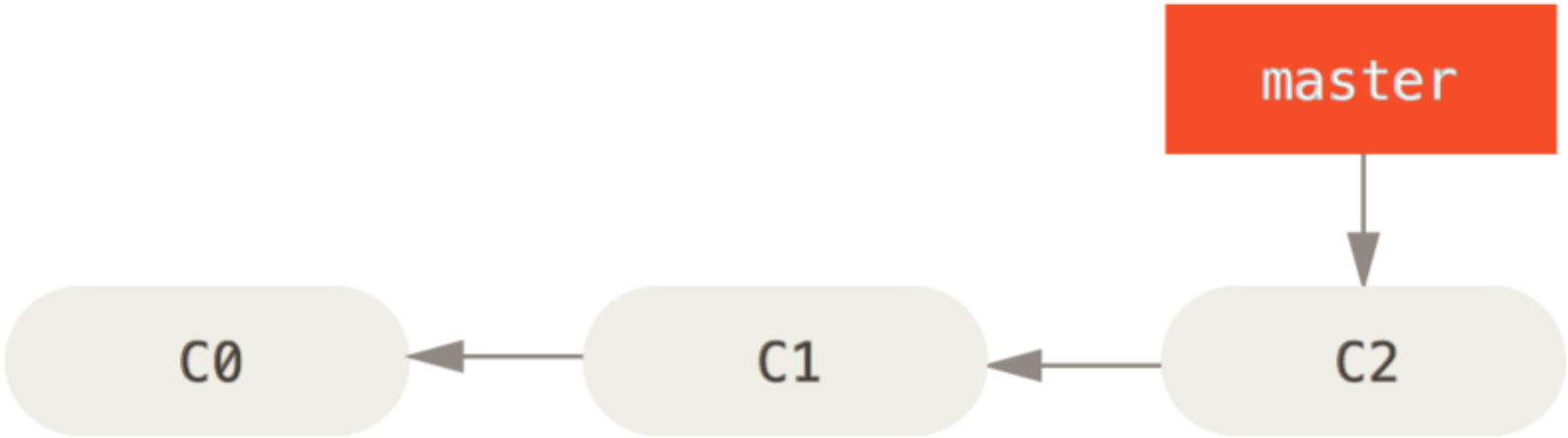
Ignoring files with .gitignore

WORKING LOCALLY WITH GIT

BRANCHING, MERGING, AND REBASING WITH GIT

- Working with local branches
- Stashing changes
- Merging branches
- Rebasing commits
- Cherry-picking commits
- Working with remote branches

http://nvie.com/posts/a-successful-git-branching-model/



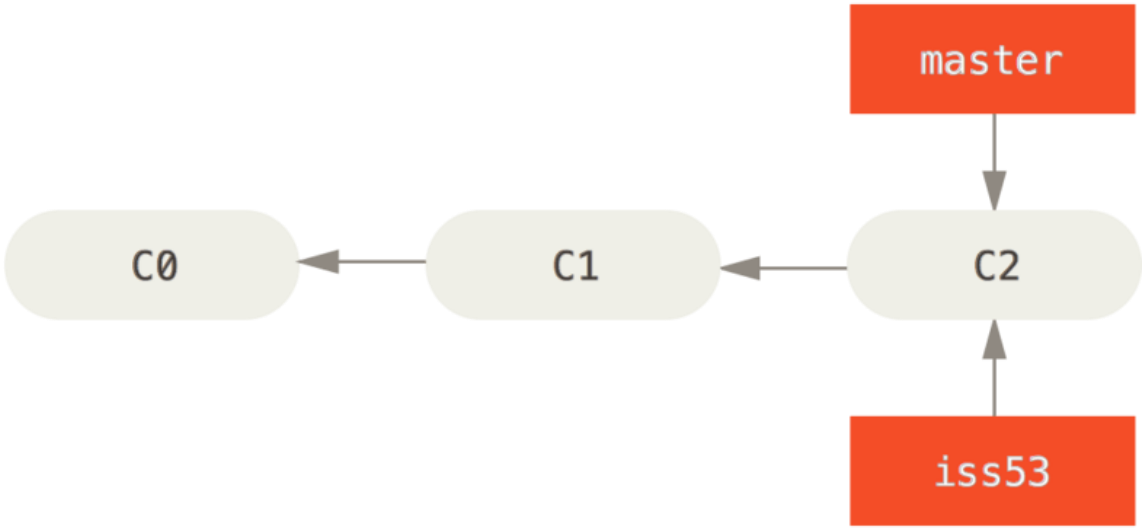
BASIC BRANCHING

A simple commit history

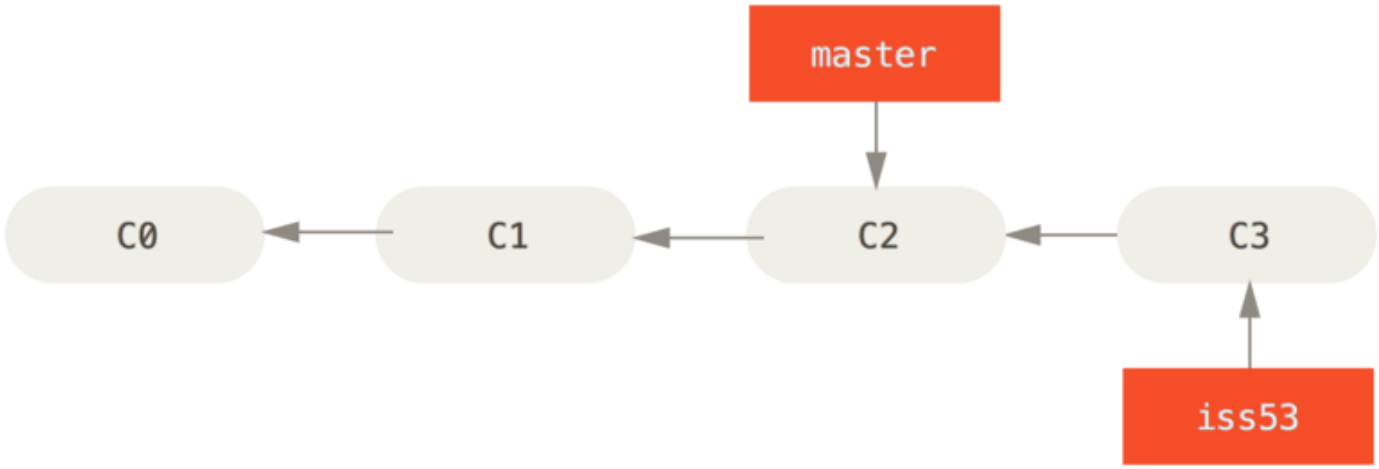
https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#_basic_merging

BASIC BRANCHING

- $ git checkout -b iss53
- Shorthand for
  - $ git branch iss53
  - $ git checkout iss53



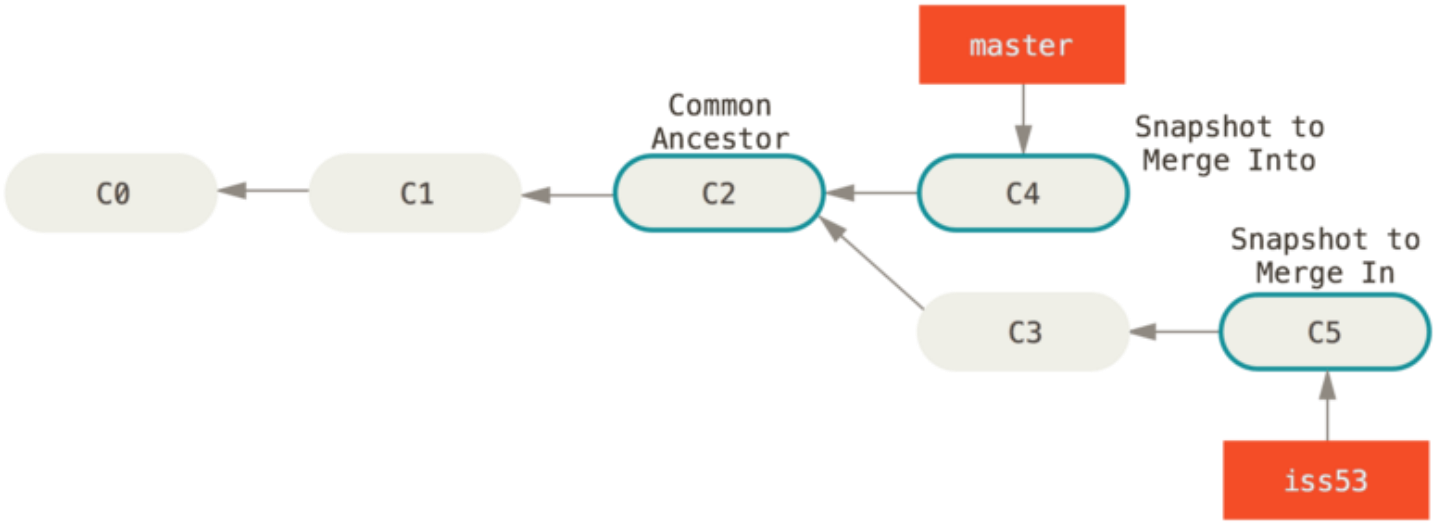
https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#_basic_merging



BASIC BRANCHING

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#_basic_merging

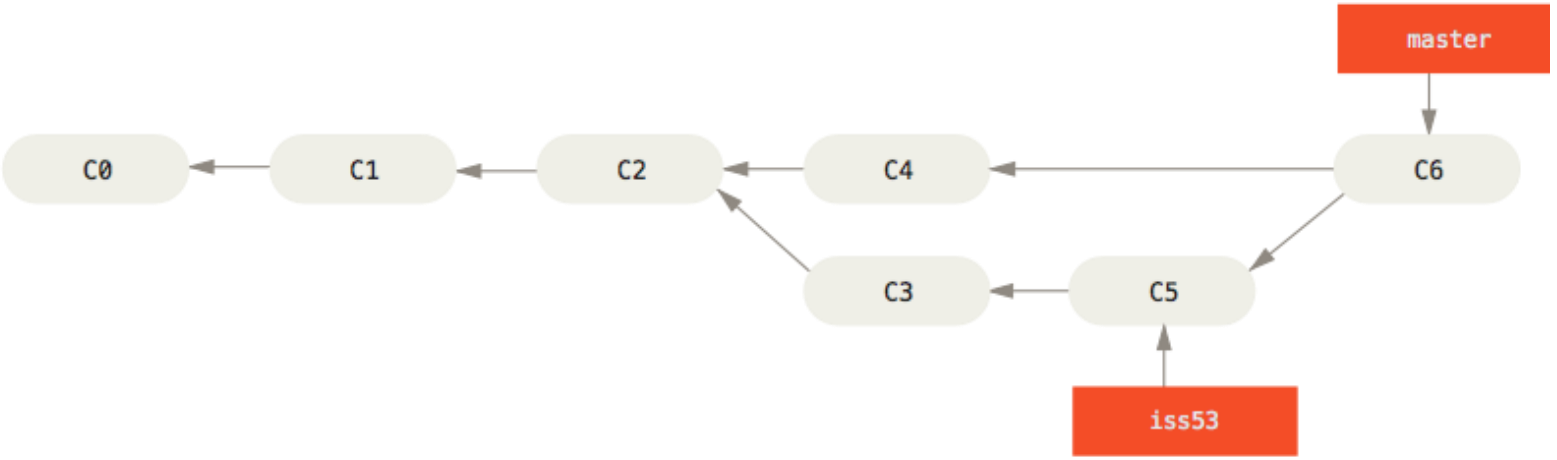
- Checkout the branch you wish to merge into
  - $ git checkout master



BASIC MERGING

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#_basic_merging

- Merge
  - $ git merge iss53



- Delete the branch
  - $ git branch -d iss53

BASIC MERGING

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging#_basic_merging

WORKING REMOTELY WITH GIT

- Cloning a remote repository
- Listing remote repositories
- Fetching changes from a remote
- Merging changes
- Pulling from a remote
- Pushing changes remotely
- Working with tags

UNIT TESTING

UNIT TESTING

- In C# there a number of tools for unit testing
- Some of the common tools are MSTest, Nunit, and XUnit

```
namespace Bank
{
  public class Account
  {
    private decimal balance;

    public void Deposit(decimal amount)
    {
      balance += amount;
    }

    public void Withdraw(decimal amount)
    {
      balance -= amount;
    }

    public void TransferFunds(Account destination, decimal amount)
    {
    }

    public decimal Balance
    {
      get { return balance; }
    }
  }
}
```

UNIT TESTING

UNIT TESTING

```
public void Deposit(decimal amount)
```

Return type: what to expect

Check the internal state of the class

Check for ref or out parameters

Parameters: Test for expected values, unexpected values and corner cases

UNIT TESTING

```
using Xunit;

class AccountTest
{
    [Fact]
    0 references
    public void TransferFounds()
    {
        var source = new Account();
        source.Deposit(200m);

        var destination = new Account();
        destination.Deposit(150m);

        source.TransferFounds(destination, 100m);

        Assert.Equal(250m, destination.Balance());
        Assert.Equal(100m, source.Balance());
    }

}
```

TYPES OF TEST

- Unit Testing
- Integration Testing
- Alpha Testing
- Beta Testing
- Use Case Testing
- Component Testing

TEST-DRIVEN DEVELOPMENT (TDD)

Idea behind test driven development

How to start using test driven development

TEST-DRIVEN DEVELOPMENT

Make it Fail

- No code without a failing test

Make it Work

- As simply as possible

Make it Better

- Refactor

UNIT TESTING MAKES YOUR DEVELOPER LIVES EASIER

Easier to find bugs

Easier to maintain

Easier to understand

Easier to Develop

START WITH THE TEST

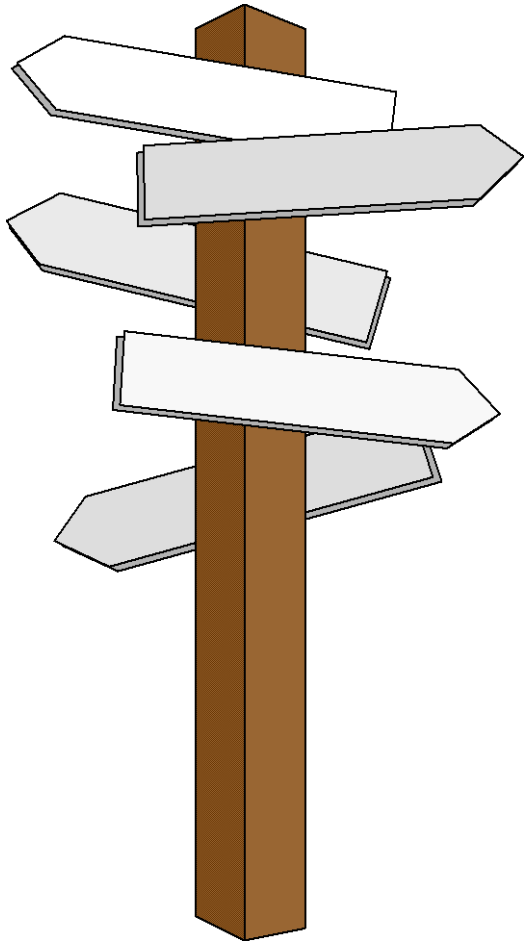
- Forces thinking about what the method does
- Provides an easy starting point
- Provides a finishing line
- You're done when the test succeeds

1. Design some minor functionality
2. Write a test that fails
3. Add the smallest amount of code to make it pass
4. GoTo 1

When all test pass – release the software

REFACTORING

- Refactoring
  - What is it?
  - Why is it necessary?
  - Examples
  - Tool support
- Refactoring principles
- Refactoring Strategy
  - Code Smells
  - Examples of Cure
- Refactoring and Reverse Engineering
  - Refactor to Understand



WHAT IS REFACTORING?

- Refactoring is a technique which allows for the re-structuring of object-oriented code into classes and methods that are more readable, modifiable, and generally sparse in code. Refactoring yields a "better" design.
- The process of changing a software system in a way that preserves the external behavior of the code, yet improves its internal structure.
- It is a disciplined way to clean up code that minimizes the chances of introducing bugs.
- Improving the design after it has been written.

WHY REFACTORING?

- Some argue that good design does not lead to code needing refactoring,
- But in reality
  - Extremely difficult to get the design right the first time
  - You cannot fully understand the problem domain
  - You cannot understand user requirements, if the user does!
  - You cannot really plan how the system will evolve in five years
  - Original design is often inadequate
  - System becomes brittle, difficult to change
- Refactoring helps you to
  - Manipulate code in a safe environment (behavior preserving)
  - Recreate a situation where evolution is possible
  - Understand existing code

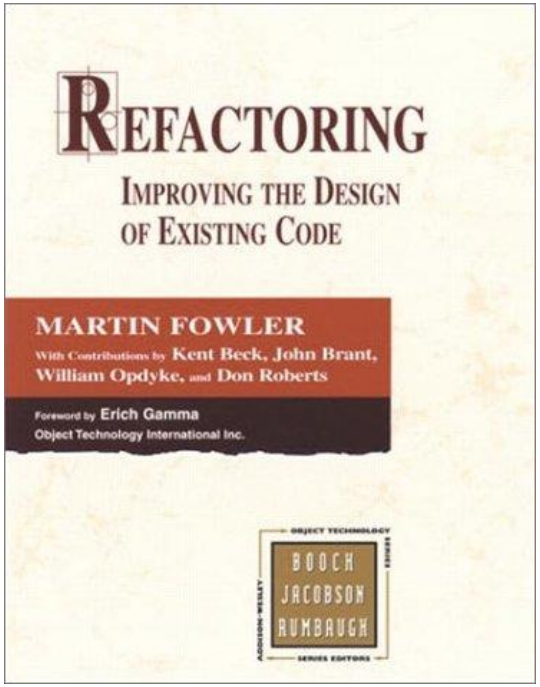
REFACTORING

- Basic metaphor:
  - Start with an existing code base and make it better.
  - Change the internal structure while preserving the overall semantics
    - *i.e.,* rearrange the "factors" but end up with the same final "product"
- The idea is that you should improve the code in some significant way. For example:
  - Reducing near-duplicate code
  - Improved cohesion, lessened coupling
  - Improved parameterization, understandability, maintainability, flexibility, abstraction, efficiency, *etc* …

REFACTORING

- Reference:
  - *Refactoring: Improving the Design of Existing Code*,

    by Martin Fowler (*et al.*), 1999, Addison-Wesley



- Fowler, Beck, *et al.* are big wheels in the OOA&D crowds
  - OO design patterns
  - XP (extreme programming)

EXAMPLE: BEFORE REFACTORING

```
class Game  {
        private Piece _piece;
        private Board _board;
        private Die[] _dice;
        // ...

        public void TakeTurn()
        {
            int rollTotal = 0;
            for (int i = 0; i < _dice.Length; i++)
            {
                _dice[i].Roll();
                rollTotal += _dice[i].FaceValue;
            }

            Square newLoc = _board.GetSquare(_piece.Location, rollTotal);
            _piece.Location = newLoc;
        }

        //...
}
```

EXAMPLE: AFTER REFACTORING (EXTRACT METHOD)

```
class Game
{
        private Piece _piece;
        private Board _board;
        private Die[] _dice;
        // ...

        public void TakeTurn()
        {
                int rollTotal = RollDice();
                Square newLoc = _board.GetSquare(_piece.Location, rollTotal);
                _piece.Location = newLoc;
        }

        private int RollDice()
        {
                int rollTotal = 0;
                foreach (Die die in _dice)
                {
                        die.Roll();
                        rollTotal += die.FaceValue;
                }
                return rollTotal;
        }
        //...
}
```

EXAMPLE: BEFORE REFACTORING

```
// good method name, but the logic of the body is not clear
bool IsLeapYear(int year)
{               return (((year % 400) == 0) ||
                        (((year % 4) == 0) && ((year % 100) != 0)));
}
```

EXAMPLE: BEFORE REFACTORING

```
// good method name, but the logic of the body is not clear
bool IsLeapYear(int year)
{               return (((year % 400) == 0) ||
                        (((year % 4) == 0) && ((year % 100) != 0)));
}


// that's better!
bool IsLeapYear(int year)
 {
    bool isFourthYear = ((year % 4) == 0);
    bool isHundrethYear = ((year % 100) == 0);
    bool is4HundrethYear = ((year % 400) == 0);
    return (is4HundrethYear   || (isFourthYear && !isHundrethYear));
 }
```

RESOURCES

- en.wikipedia.org/wiki/XUnit
- nunit.org
- www.testdriven.com
- www.refactoring.com
- c2.com/cgi/wiki?WhatIsRefactoring