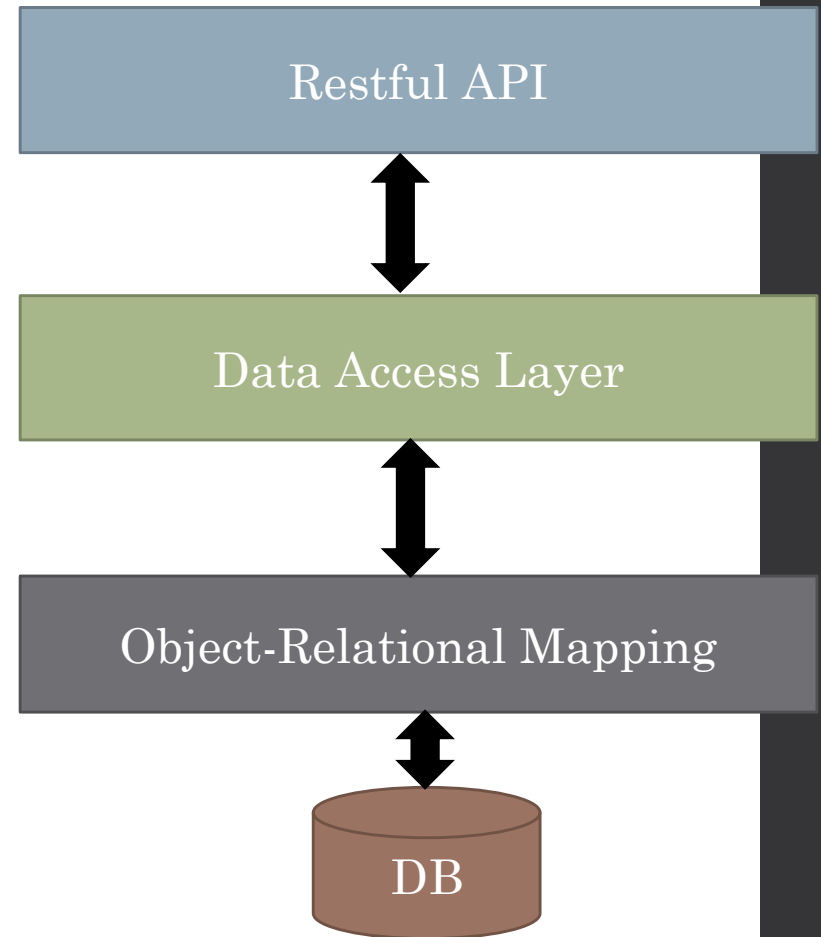


# Complex IT Systems Section 2

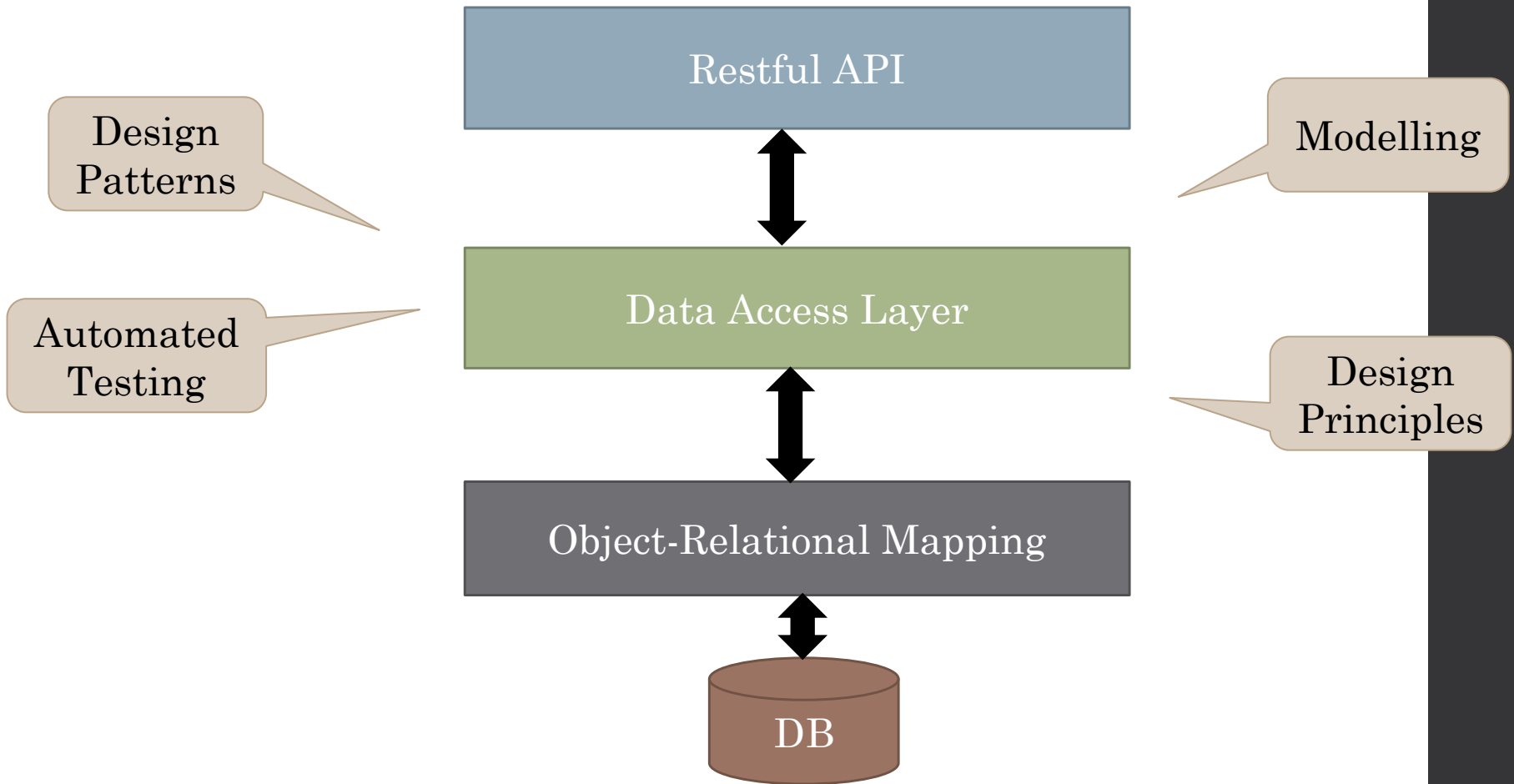
Henrik Bulskov

# What to do in section 2?

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



# System Development



# Development Tools

Visual  
Studio

Visual  
Studio Code

Rider  
JetBrains

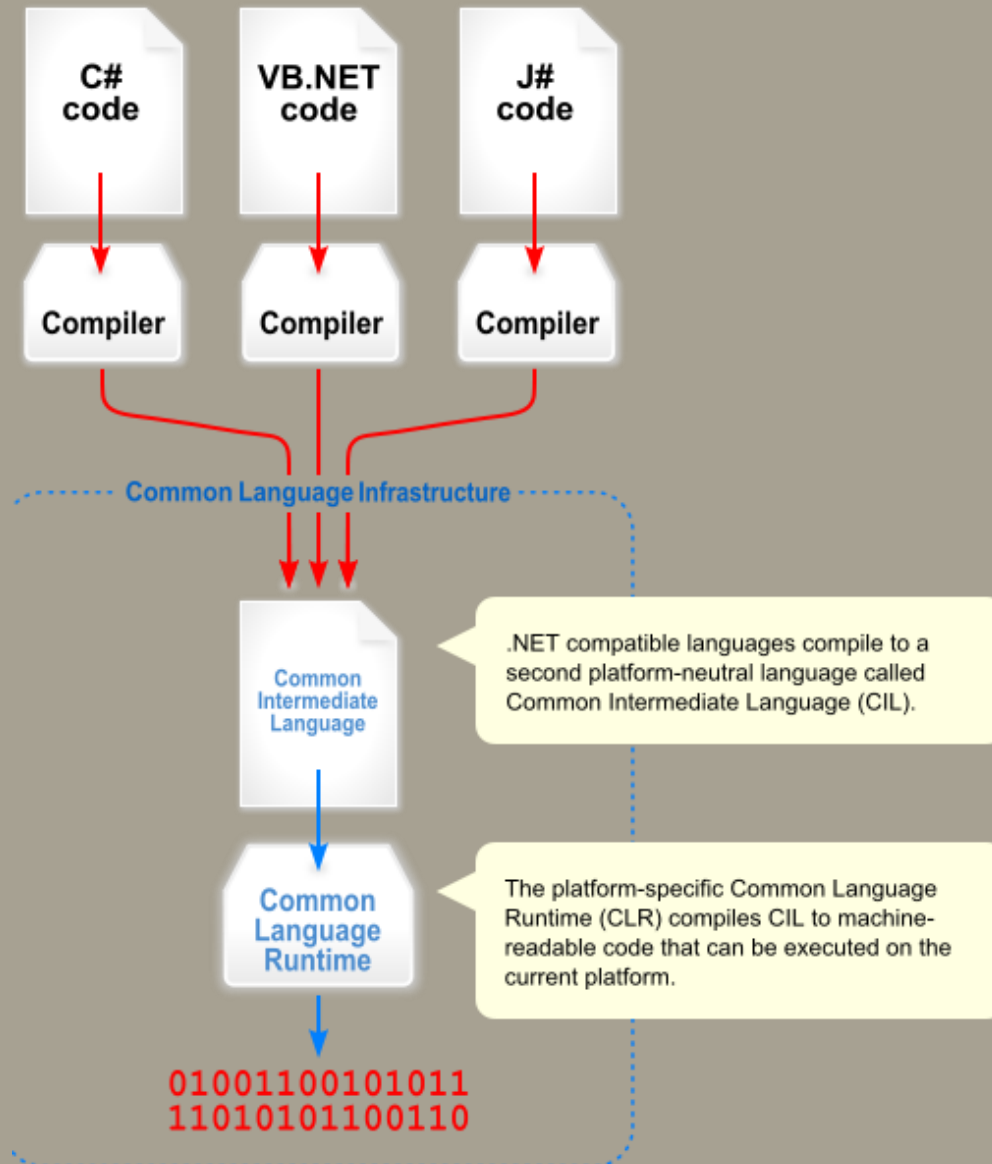


# Tools

- Download:
  - <https://visualstudio.microsoft.com/downloads/>
  - <https://www.jetbrains.com/idea/>
- 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



## .NET FRAMEWORK

.NET



Cloud



Web



Desktop



Mobile



Gaming



IoT



AI



Visual Studio



Visual Studio  
Code



CLI



GitHub  
Copilot

+



Windows



Linux



macOS

+



NuGet



GitHub



.NET  
Aspire



Components, tools,  
library vendors

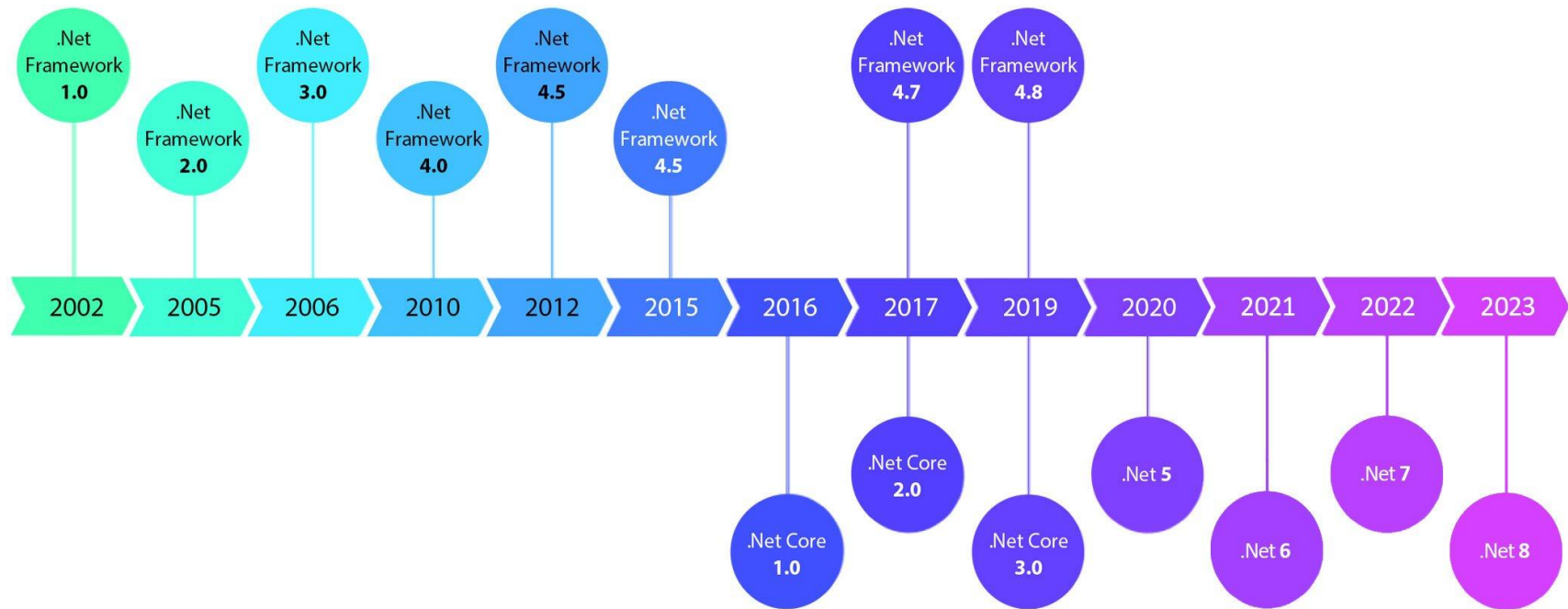
Tools

Operating systems

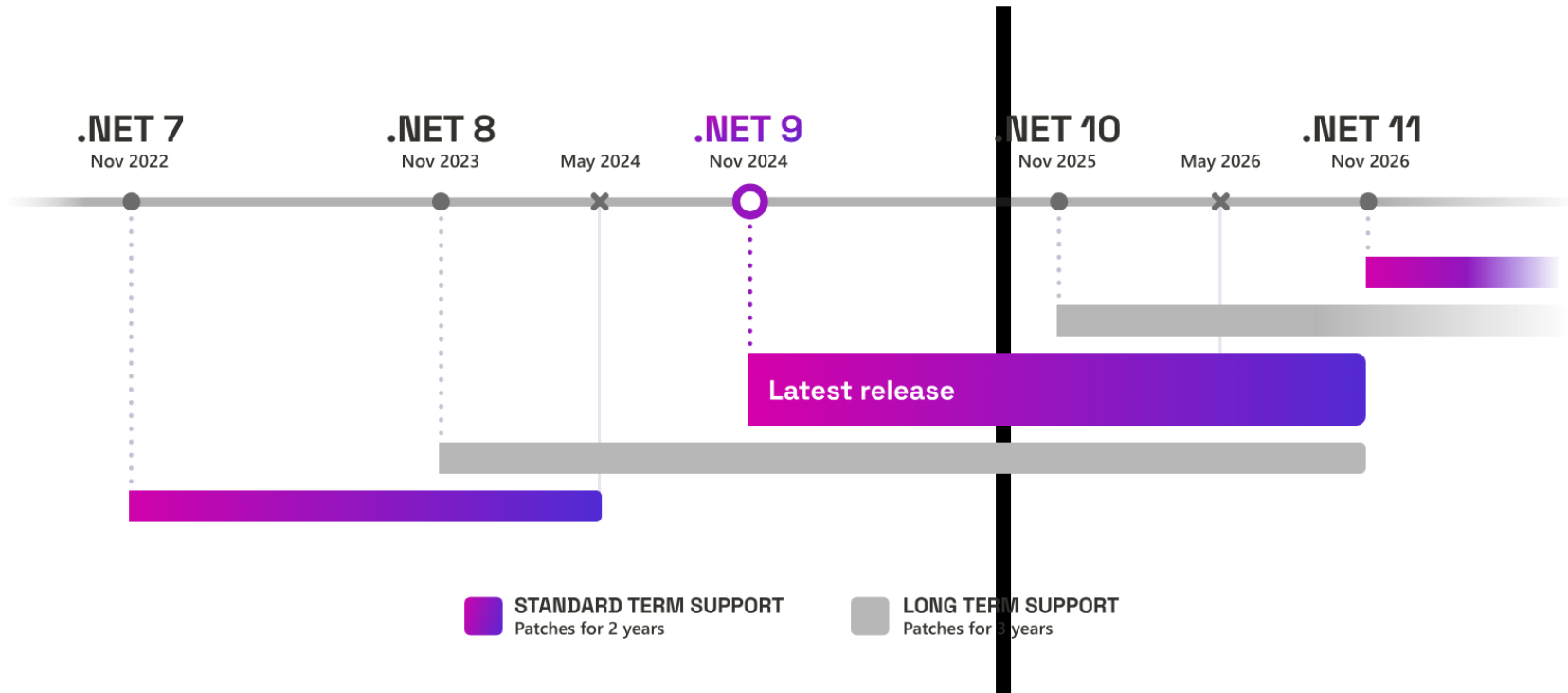
Ecosystem

# .NET 9





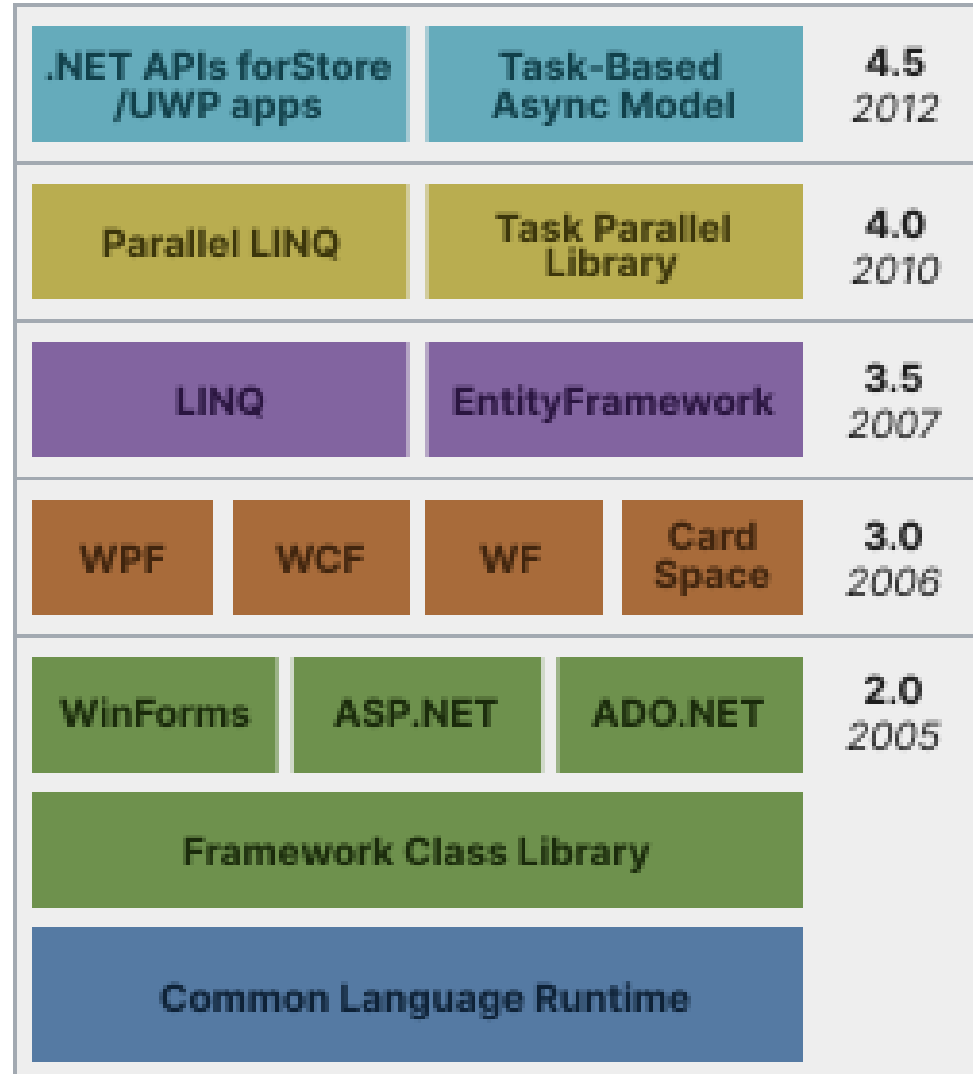
# History of a Framework

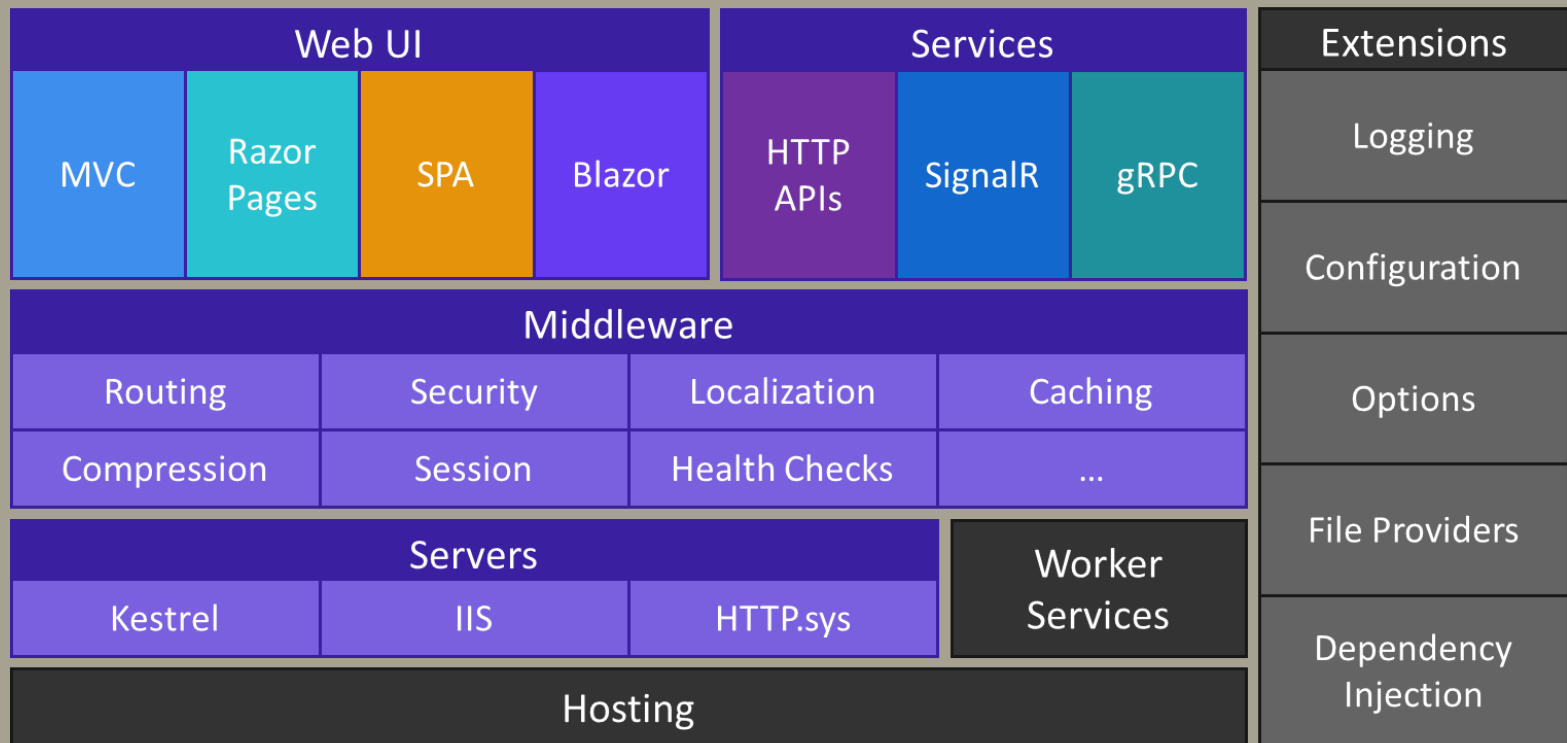


# History of a Framework

# .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





# ASP.NET Core High-Level Overview

# Learn Programming

How to  
become a  
great  
programmer?

You need to  
practice

# What is practice?

- Explore gap in knowledge
- Ephemeral
- Not competitive
- Not research
- Not a project
- Daily
- You can fit in 30 min.  
Plan!



(Mike Acton)

# AI is Not a Solution — It's a Tool

- Don't forget what it will take from you

Knowledge

Skills

Insight

Understanding

Problem-solving ability



# Final Note on AI and Code



Do not submit code you do not understand!



At the exam, you will be expected to **explain your code** clearly and confidently. Submitting code that you cannot explain — whether generated by AI or copied from elsewhere — is not only academically risky, it undermines your learning.



It is **far better** to submit code that you have written yourself and fully understand, even if it's not perfect. Your grade reflects **your understanding**, not the sophistication of the code. At the exam you need to be able to explain your code!!!



If you submit code that you cannot explain, it may be considered **plagiarism** and will be reported accordingly.



## A BRIEF HISTORY OF

# VERSION CONTROL

**Early 1960s**

IEBUPDTE for IBM OS/360  
(punched card system)

**1982**

RCS (Revision Control System) created  
by Walter Tichy

**2000**

Subversion (SVN) created by  
CollabNet

**1972**

SCCS (Source Code Control  
System) created by Marc  
Rochkind

**1986**

CVS (Concurrent Versions Systems)  
created by Dick Grune

**2005**

Git created by Linus Torvalds

CREATED BY TARYN MCMILLAN

# A Brief History of Version Control

# 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.html](http://www.ericSink.com/vcbe/html/history_of_version_control.html)

# 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

# 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

# Working Locally with Git

Creating a  
local  
repository

Adding files

Committing  
changes

Viewing  
history

Viewing a diff

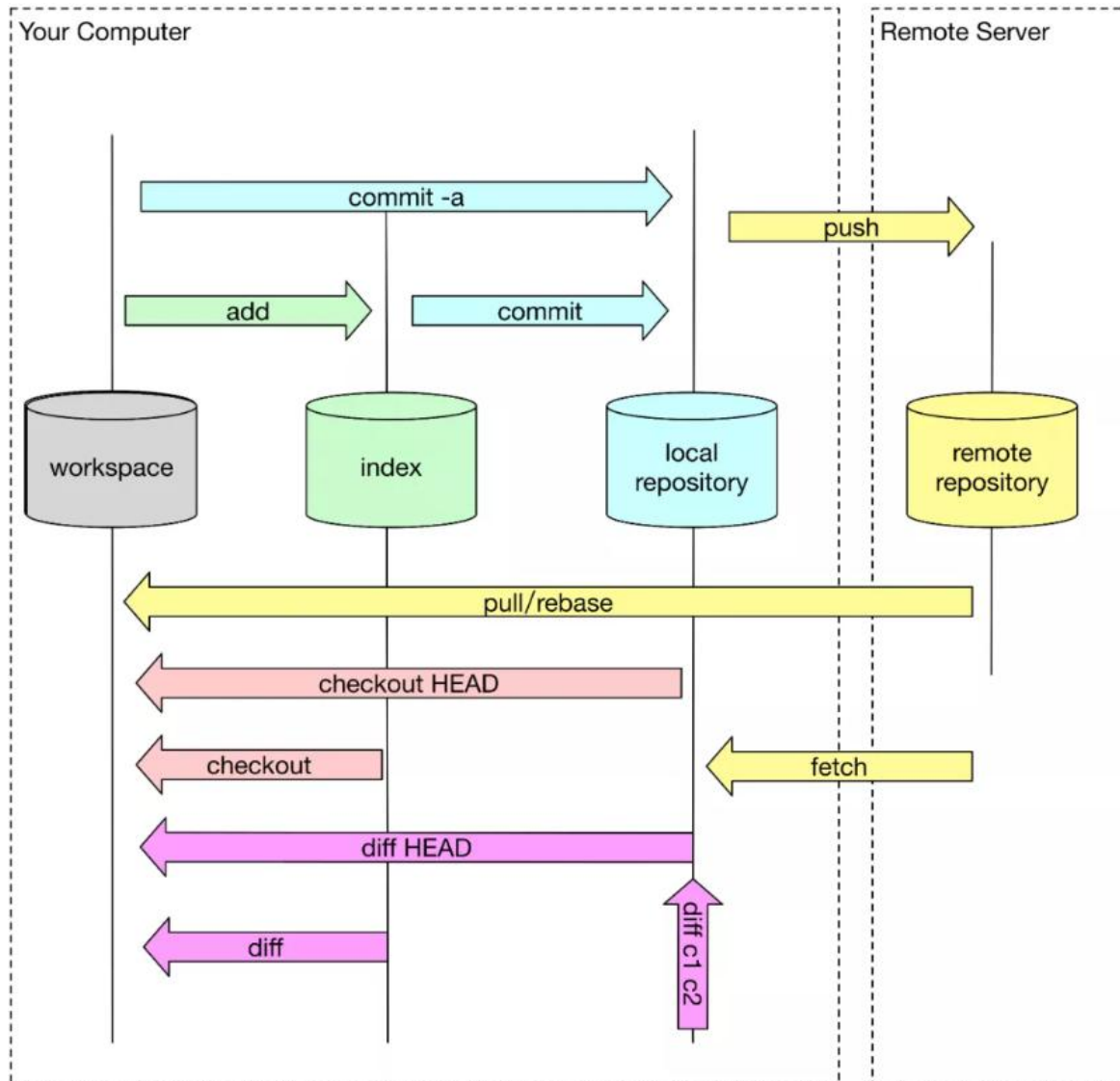
Working copy,  
staging, and  
repository

Deleting files

Cleaning the  
working copy

Ignoring files  
with  
.gitignore

# GIT Storage Model

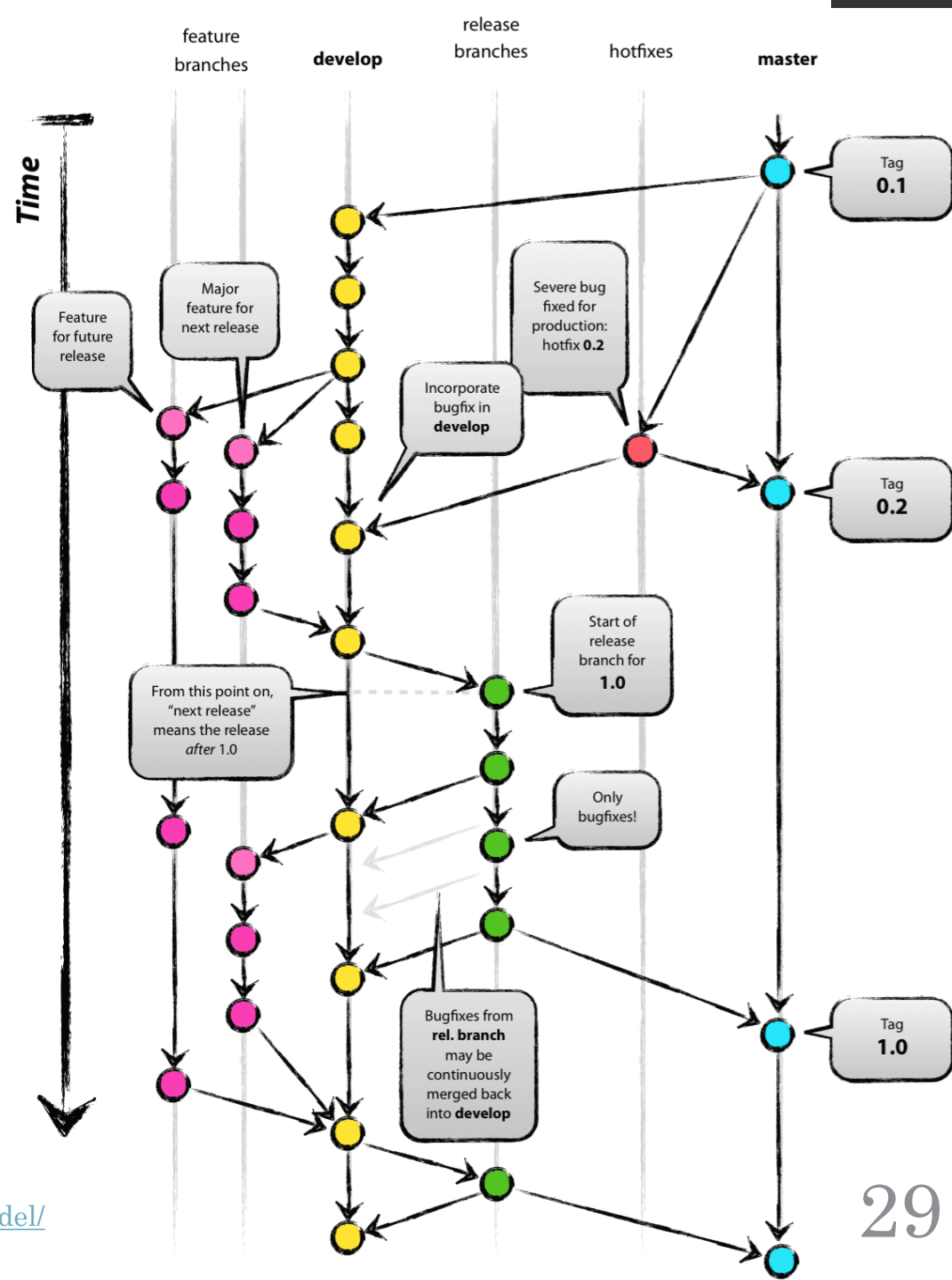


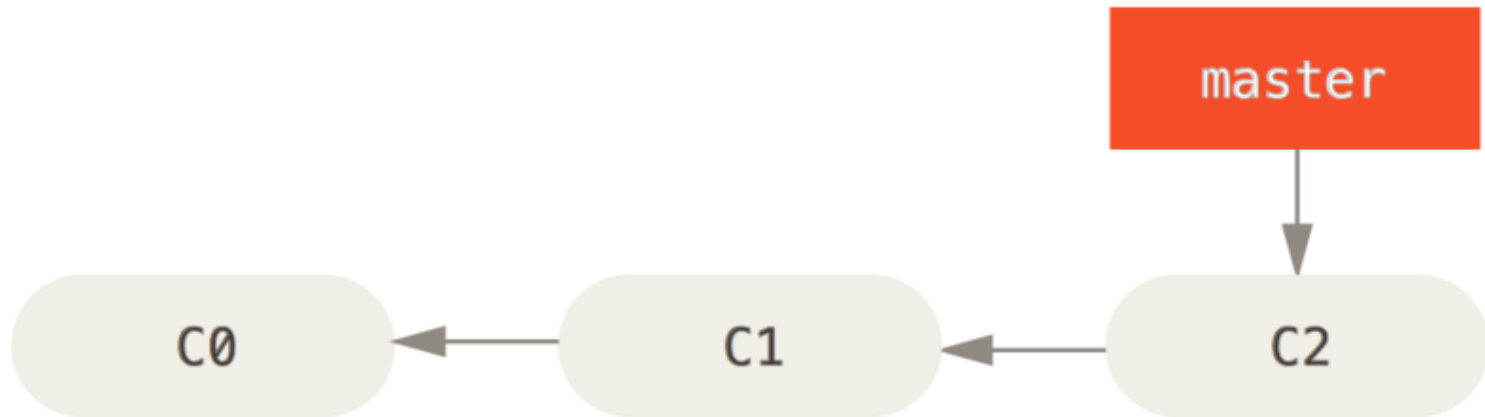
# Branching, AND Merging with Git

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

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

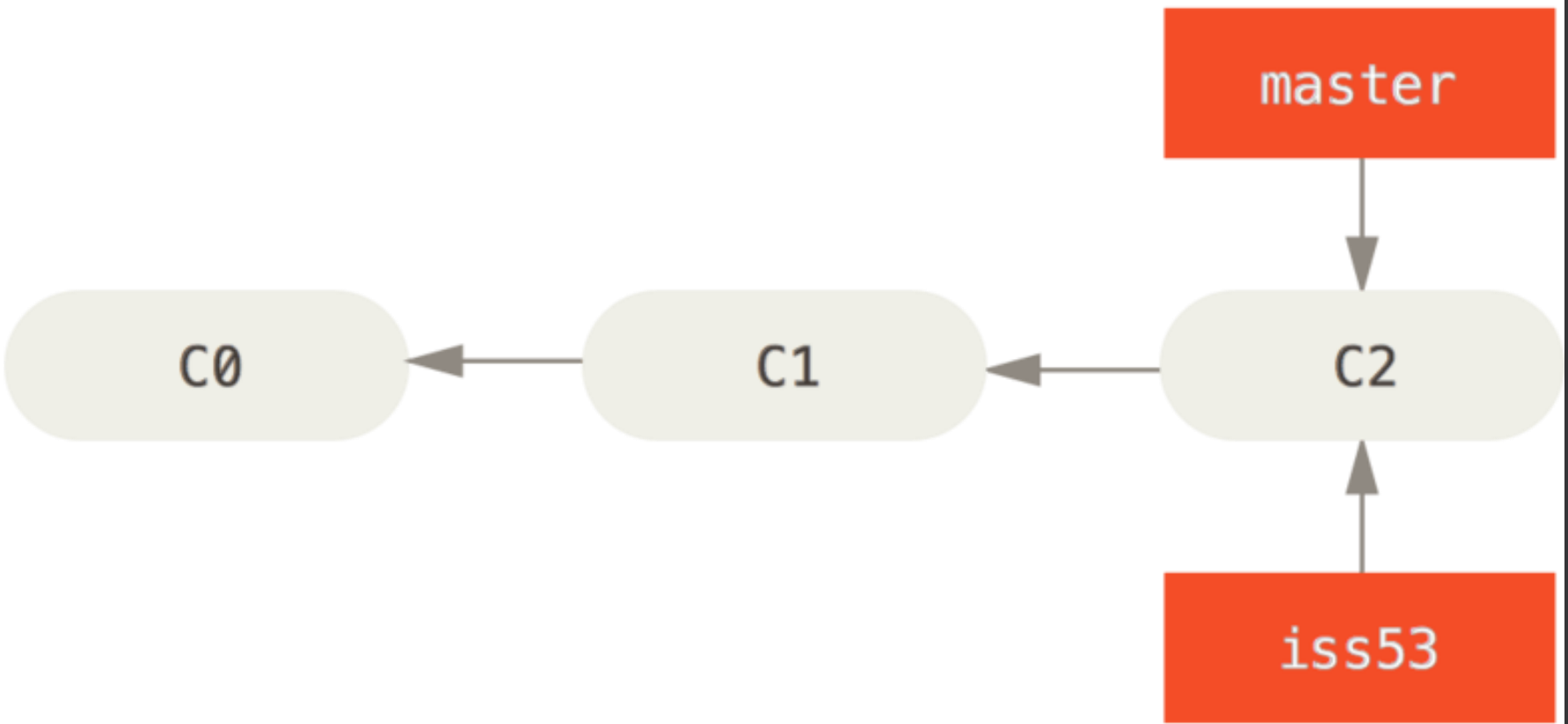
# Branching with Git





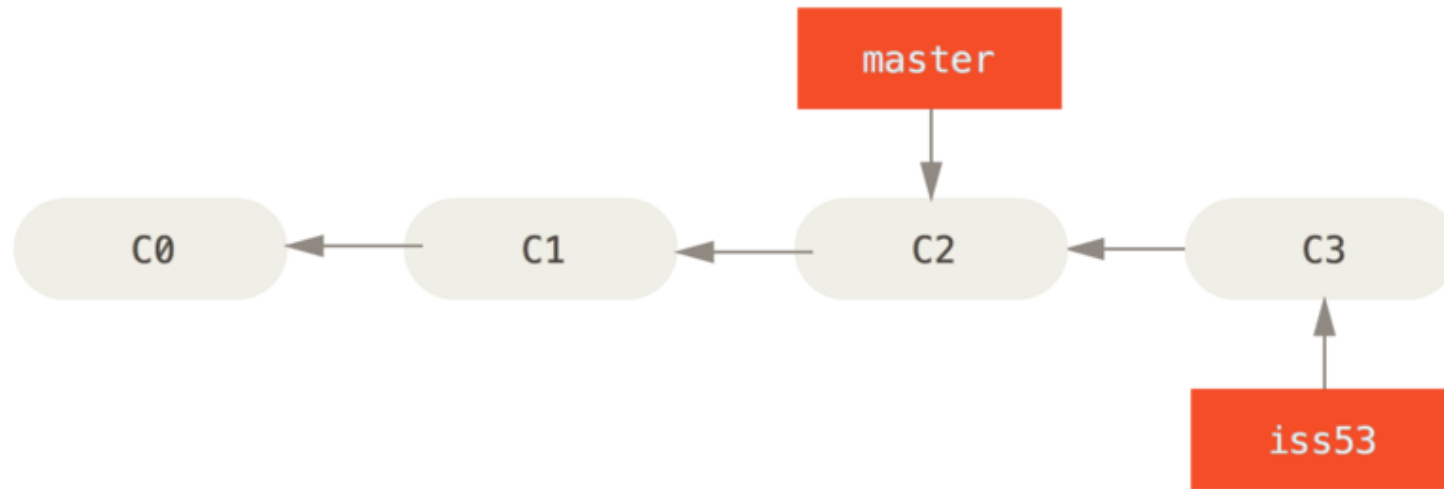
# Basic Branching

A simple commit history



## Basic Branching

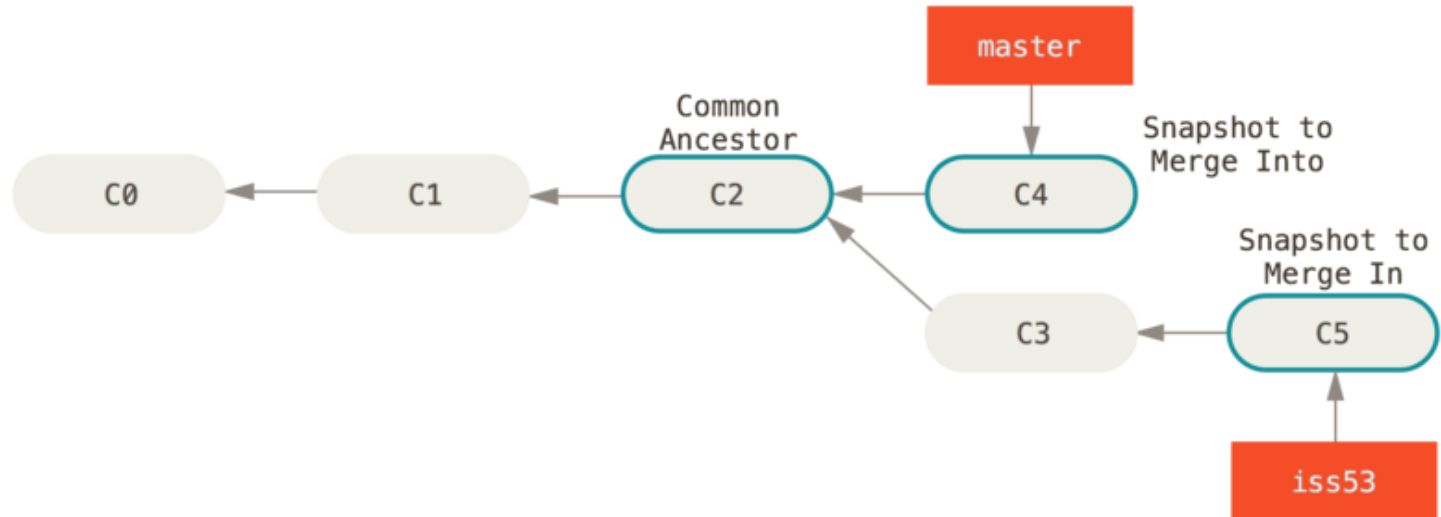
- `$ git checkout -b iss53`
- Shorthand for
  - `$ git branch iss53`
  - `$ git checkout iss53`



# Basic Branching

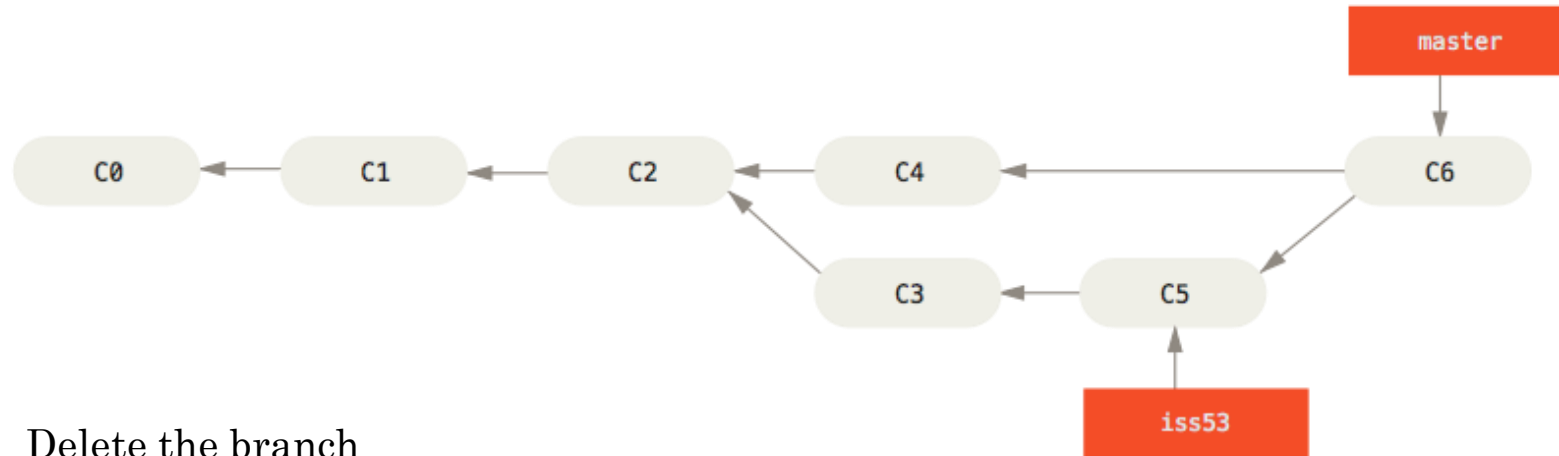


- Checkout the branch you wish to merge into
  - `$ git checkout master`



# Basic Merging

- Merge
  - `$ git merge iss53`



- Delete the branch
  - `$ git branch -d iss53`

# 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



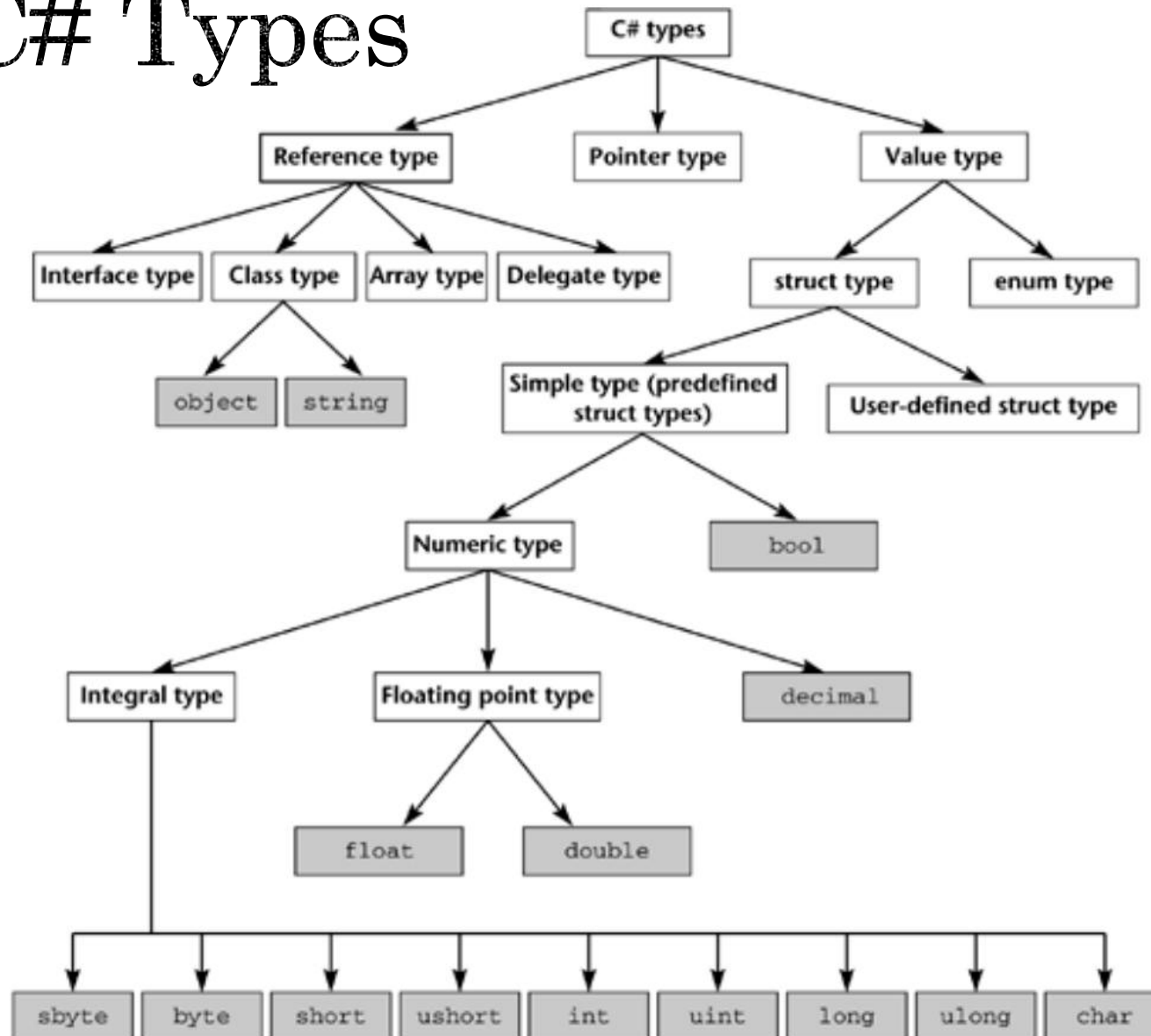
Language  
basics

# 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`

<https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines>

# C# Types

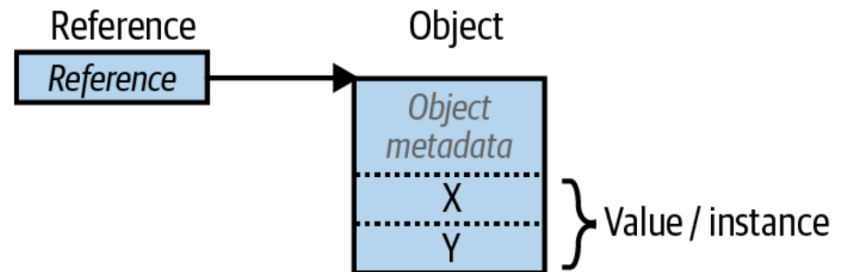


# Value Types Versus Reference Types

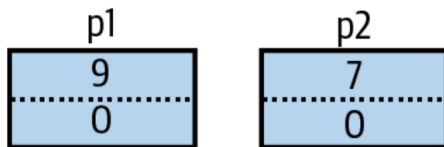
Point struct



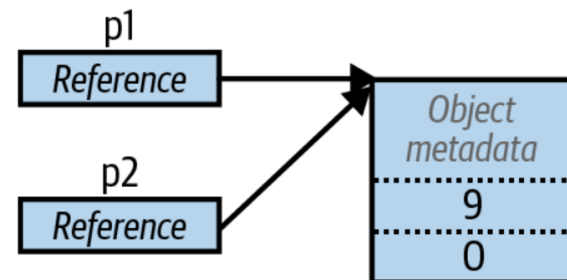
Point class



Point struct



```
Point p1 = new Point();  
p1.X = 7;  
Point p2 = p1;  
p1.X = 9;
```



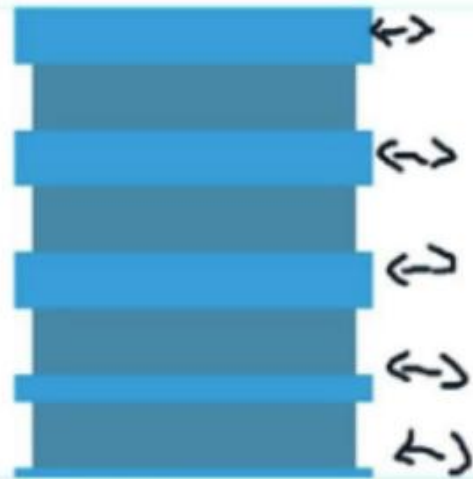
# Value Types Versus Reference Types

Value-type instances occupy precisely the memory required to store their fields.

Reference types require separate allocations of memory for the reference and object.

The object consumes as many bytes as its fields, plus additional administrative overhead.



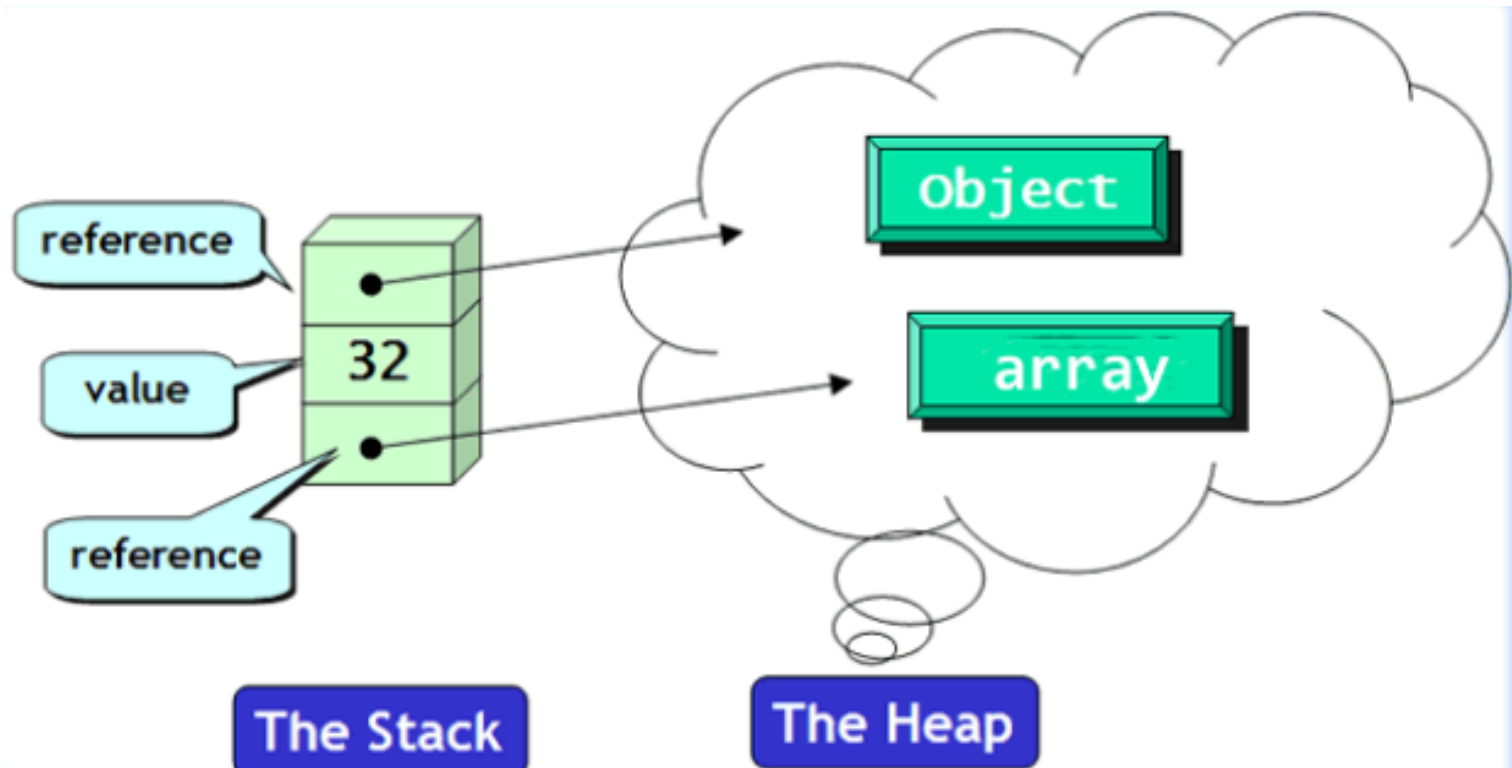


**Stack**



**Heap**

# Heap and Stack Memory



# Memory

# Stack and Heap Memory

## Stack

- It is an array of memory.
- It is a LIFO (Last In First Out) data structure.
- In it, data can be added to and deleted only from the top of it.
- Is allocated when the program start (static).

## Heap

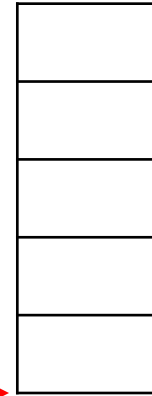
- It is an area of memory where chunks are allocated to store certain kinds of data objects.
- In it, data can be stored and removed in any order.
- Is allocated when needed (dynamic).

# Stack and HeaP Memory

→

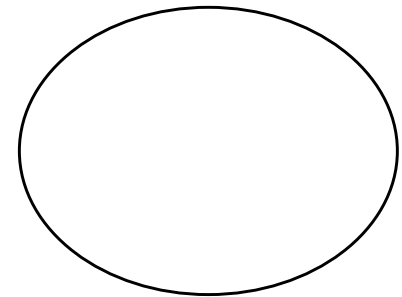
```
public void SomeMethod()  
{  
    int x = 101;  
    int y = 102;  
    SomeClass cls = new SomeClass();  
}
```

Stack



...

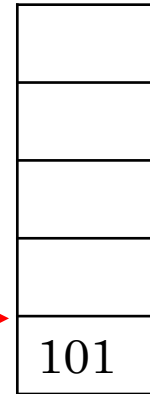
Heap



# Stack and HeaP Memory

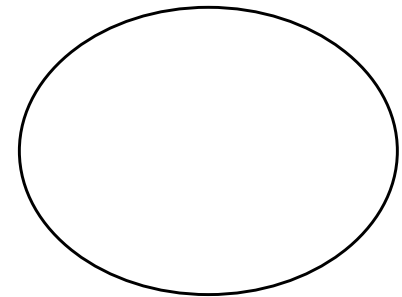
```
public void SomeMethod()  
{  
    int x = 101;  
    int y = 102;  
    SomeClass cls = new SomeClass();  
}
```

Stack



...

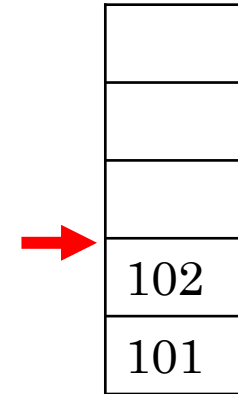
Heap



# Stack and HeaP Memory

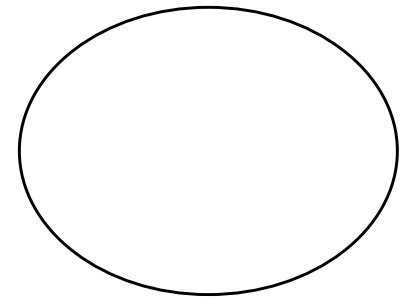
```
public void SomeMethod()  
{  
    int x = 101;  
    int y = 102;  
    SomeClass cls = new SomeClass();  
}
```

Stack



...

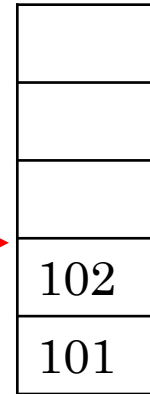
Heap



# Stack and HeaP Memory

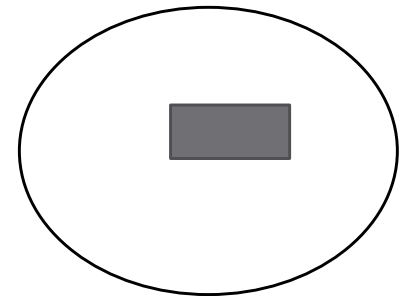
```
public void SomeMethod()  
{  
    int x = 101;  
    int y = 102;  
    SomeClass cls = new SomeClass();  
}
```

Stack



...

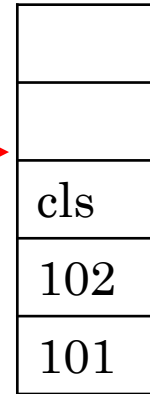
Heap



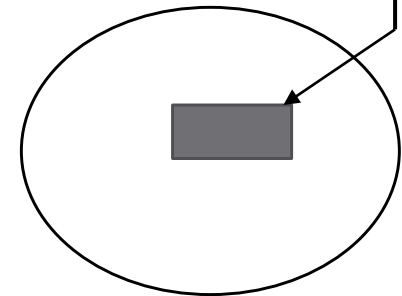
# Stack and HeaP Memory

```
public void SomeMethod()  
{  
    int x = 101;  
    int y = 102;  
    SomeClass cls = new SomeClass();  
}
```

Stack



Heap





# Stack and HeaP Memory

```
public void SomeMethod()  
{  
    int x = 101;  
    int y = 102;  
    SomeClass cls = new SomeClass();  
}
```

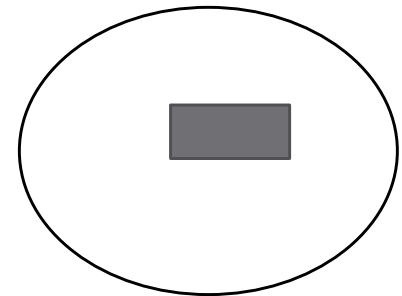


Stack



...

Heap



# C# features

# 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
```

- Nullable<T> struct

```
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"
```

```
string s1 = null;  
s1 ??= "something";
```

C# 8

- 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

```
var x = "hello";  
var y = new System.Text.StringBuilder();  
var z = (float)Math.PI;
```

This is precisely equivalent to the following:

```
string x = "hello";  
System.Text.StringBuilder y = new System.Text.StringBuilder();  
float z = (float)Math.PI;
```

# Enums

```
public enum BorderSide { Left, Right, Top, Bottom }
```

```
BorderSide topSide = BorderSide.Top;  
bool isTop = (topSide == BorderSide.Top);    // true
```

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```

```
public enum BorderSide : byte  
{ Left=1, Right, Top=10, Bottom }
```



# Selection Statements

Selection statements (if, switch)

Conditional operator (?:)

Loop statements (while, do-while, for, foreach)

# 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 ☺

```
while (condition)  
    body
```

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

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

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

```
int[] values = {1,2,3,4};  
  
foreach (int value in values){  
    Console.WriteLine(value);  
}
```

# Loop Statements

# ARRAYS

```
char[] vowels = new char[5];    // Declare an array of 5 characters
```

Array initialization expression:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

```
char[] vowels = {'a','e','i','o','u'};
```

Indices and Ranges (C# 8)

```
char[] vowels = new char[] {'a','e','i','o','u'};  
char lastElement = vowels [^1];    // 'u'  
char secondToLast = vowels [^2];   // 'o'
```

```
char[] firstTwo = vowels [..2];     // 'a', 'e'  
char[] lastThree = vowels [2..];    // 'i', 'o', 'u'  
char[] middleOne = vowels [2..3];   // 'i'
```

# Multidimensional Arrays

Rectangular arrays:

```
int[,] matrix = new int[3,3];
```

```
int[,] matrix = new int[,]  
{  
    {0,1,2},  
    {3,4,5},  
    {6,7,8}  
};
```

Jagged arrays

```
int[][] matrix = new int[3][];
```

```
int[][] matrix = new int[][]  
{  
    new int[] {0,1,2},  
    new int[] {3,4,5},  
    new int[] {6,7,8,9}  
};
```

# Parameter Passing

- Parameters are passed as a copy

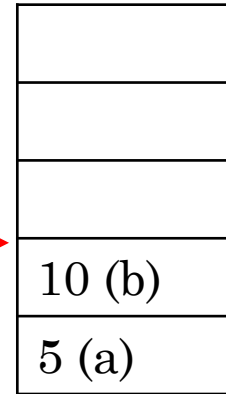
```
static void Main(string[] args)
{
    int a = 5;
    int b = 10;
    Swap(a, b);

    Console.WriteLine($"a = {a}, b = {b}");
}
```

1 reference

```
public static void Swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Stack



# Parameter Passing

- Parameters are passed as a copy

```
static void Main(string[] args)
{
    int a = 5;
    int b = 10;


    Swap(a, b);

    Console.WriteLine($"a = {a}, b = {b}");
}
```

1 reference

```
public static void Swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Stack



5 (tmp)
10 (y)
5 (x)
10 (b)
5 (a)

# Parameter Passing

- Parameters are passed as a copy

```
static void Main(string[] args)
{
    int a = 5;
    int b = 10;


    Swap(a, b);

    Console.WriteLine($"a = {a}, b = {b}");
}
```

1 reference

```
public static void Swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Stack



5 (tmp)
5 (y)
10 (x)
10 (b)
5 (a)





# Parameter Passing

- Parameters are passed as a copy

```
static void Main(string[] args)
{
    int a = 5;
    int b = 10;

    Swap(a, b);

    Console.WriteLine($"a = {a}, b = {b}");
}
```

1 reference

```
public static void Swap(int x, int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Stack

5 (tmp)
5 (y)
10 (x)
10 (b)
5 (a)



# Parameter Passing

- Parameters are passed as a reference

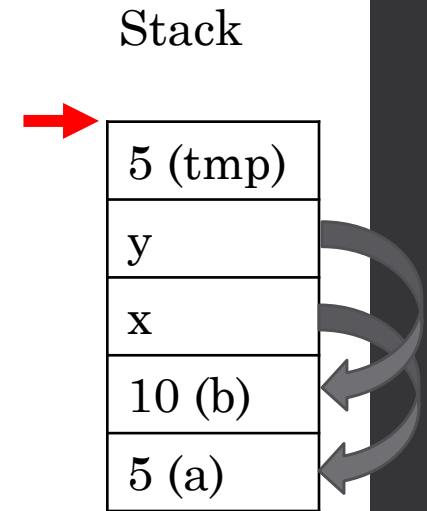
```
static void Main(string[] args)
{
    int a = 5;
    int b = 10;

    Swap(ref a, ref b);

    Console.WriteLine($"a = {a}, b = {b}");
}
```

1 reference

```
public static void Swap(ref int x, ref int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```



# 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;
    }
}

....

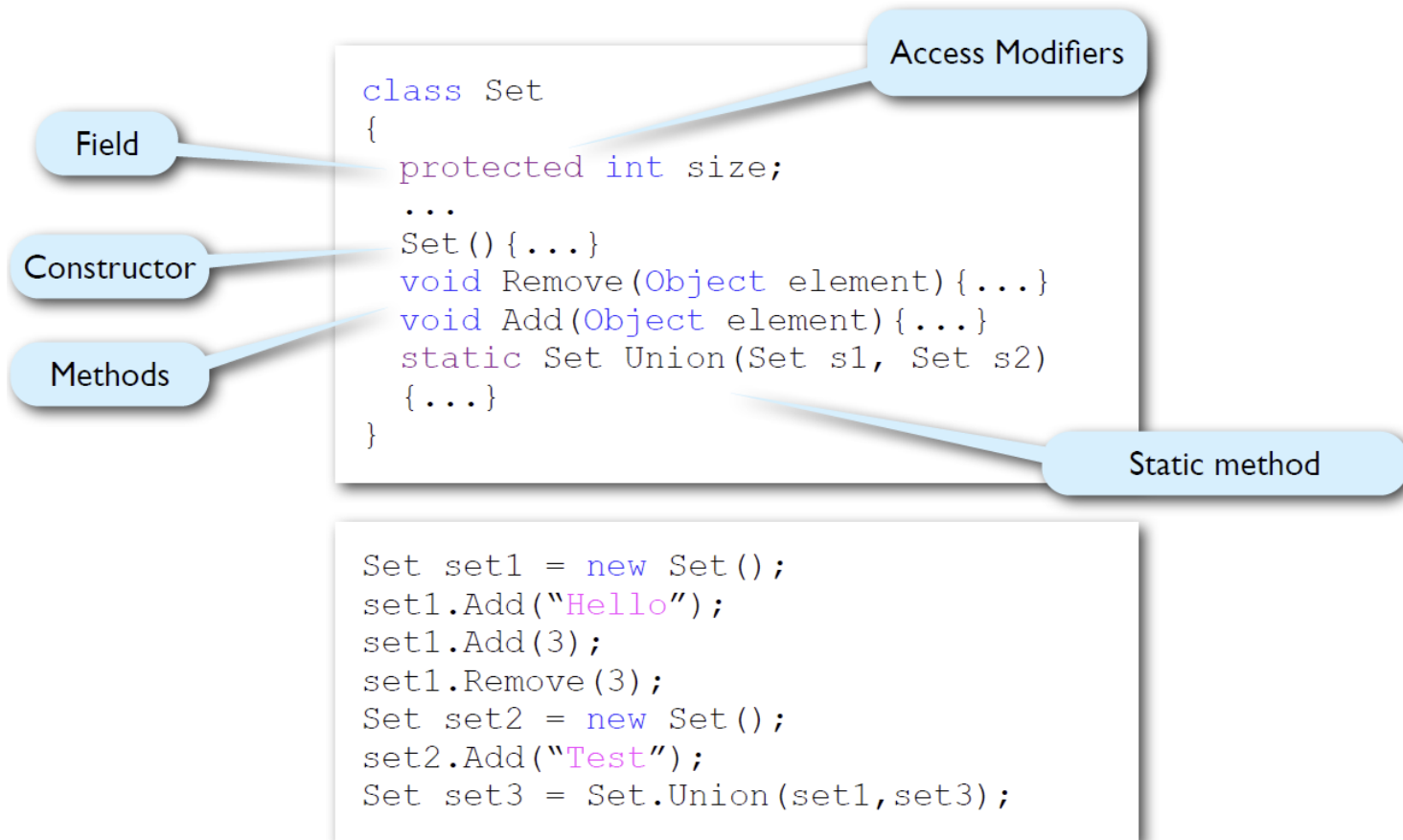
int max = Math.Max(3, 6, 78, 3, 5);
```

It cannot be ref or out

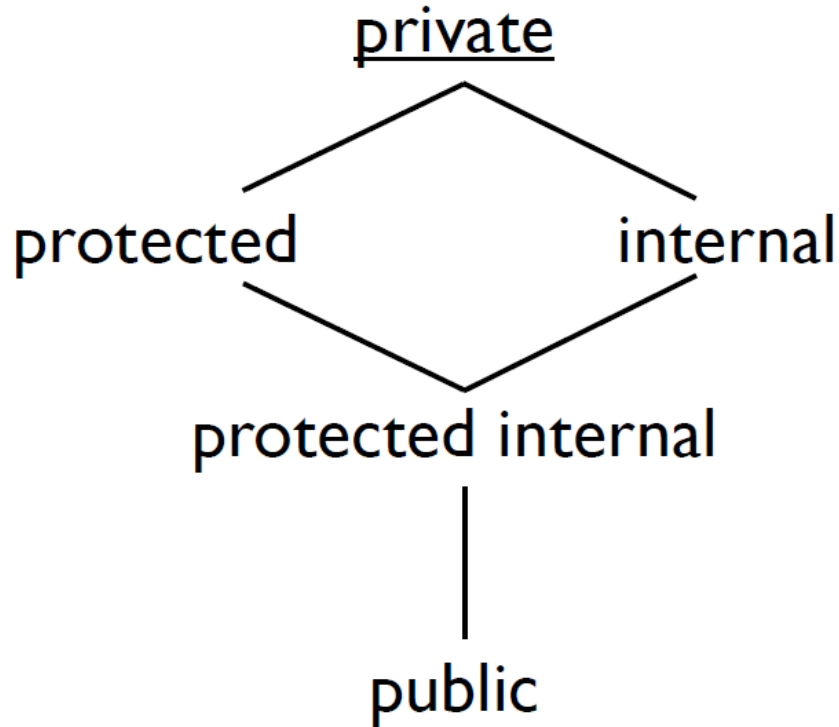
Allows an undefined number of parameters. Only once, and at the end

# PARAMETER PASSING

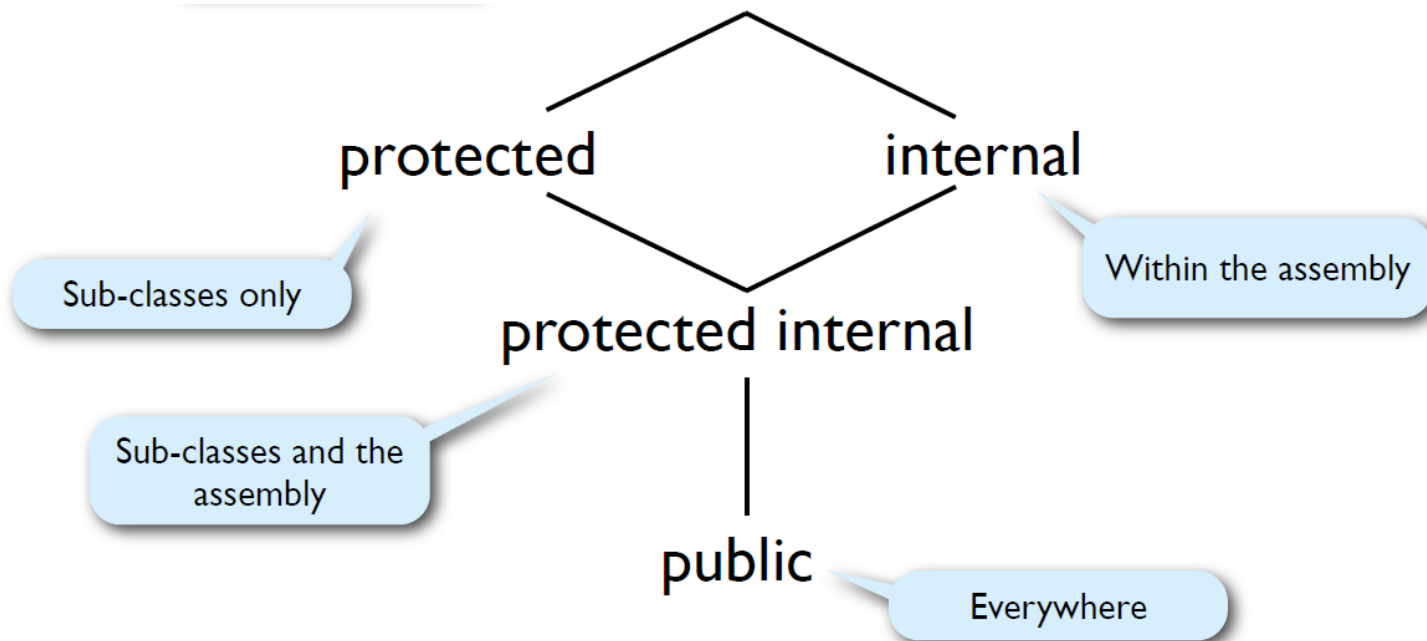
Implement a Max function for  $n$  parameters



# Class



# Member Access Modifiers



# MEMBER ACCESS MODIFIERS

# 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 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; }
    }
}
```

```
public class Stock
{
```

- Automatic properties

```
...
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 int Maximum { get; } = 999;
```

- **Init-only setters**

```
public int Pitch { get; init; } = 20;
```

Private fields and attributes are not 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

# Class Modifiers

Abstract

Static

Sealed

```
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;
        }
    }
    ...
}
```

# Interface Declaration

```
struct Fraction
{
    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 vs Struct

# Indexer Declaration

- Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values.

```
class Sentence
{
    string[] words = "The quick brown fox".Split();

    public string this [int wordNum]      // indexer
    {
        get { return words [wordNum]; }
        set { words [wordNum] = value; }
    }
}
```

# Indexer Declaration

```
class Sentence
{
    string[] words = "The quick brown fox".Split();

    public string this [int wordNum]        // indexer
    {
        get { return words [wordNum]; }
        set { words [wordNum] = value; }
    }
}
```

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);           // fox
s[3] = "kangaroo";
Console.WriteLine (s[3]);           // kangaroo
```



# Operator Overloading

```
class Num
{
    private int _value;

    1 reference
    public Num(int value)
    {
        _value = value;
    }
}
```

# Operator Overloading

```
class Num
{
    private int _value;

    3 references
    public Num(int value)
    {
        _value = value;
    }

    0 references
    public Num Add(Num other)
    {
        return new Num(_value + other._value);
    }
}
```

```
var n1 = new Num(5);
var n2 = new Num(7);

var n3:Num = n1.Add(n2);
```

# Operator Overloading

```
class Num
{
    private int _value;

    1 reference
    public Num(int value)
    {
        _value = value;
    }
}
```

```
var n1 = new Num(5);
var n2 = new Num(7);
```

```
var n3:Num = n1 + n2;
var n4:Num = n1 + 5;
n1++;
```

---

# Operator Overloading

1 reference

```
public static Num operator+(Num n1, Num n2)
{
    return new Num(n1._value + n2._value);
}
```

1 reference

```
public static Num operator +(Num n1, int val)
{
    return new Num(n1._value + val);
}
```


1 reference

```
public static Num operator ++(Num n)
{
    n._value++;
    return n;
}
```

```
var n1 = new Num(5);
var n2 = new Num(7);
```

```
var n3:Num = n1 + n2;
var n4:Num = n1 + 5;
n1++;
```

# Operator Overloading

Operators	Overloadability
<code>+, -, !, ~, ++, --, true, false</code>	These unary operators can be overloaded.
<code>+, -, *, /, %, &amp;,  , ^, &lt; &lt;, &gt; &gt;</code>	These binary operators can be overloaded.
<code>==, !=, &lt;, &gt;, &lt;=, &gt;=</code>	The comparison operators can be overloaded (but see the note that follows this table).
<code>&amp;&amp;,   </code>	The conditional logical operators cannot be overloaded, but they are evaluated using <code>&amp;</code> and <code> </code> , which can be overloaded.
<code>[]</code>	The array indexing operator cannot be overloaded, but you can define indexers.
<code>(T)x</code>	The cast operator cannot be overloaded, but you can define new conversion operators (see <a href="#">explicit</a> and <a href="#">implicit</a> ).
<code>+=, -=, *=, /=, %=, &amp;=,  =, ^=, &lt;&lt;=, &gt;&gt;=</code>	Assignment operators cannot be overloaded, but <code>+=</code> , for example, is evaluated using <code>+</code> , which can be overloaded.
<code>=, ., ?;, ??, -&gt;, =&gt;, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof</code>	These operators cannot be overloaded.
 <b>Note</b>	
<p>The comparison operators, if overloaded, must be overloaded in pairs; that is, if <code>==</code> is overloaded, <code>!=</code> must also be overloaded. The reverse is also true, and similar for <code>&lt;</code> and <code>&gt;</code>, and for <code>&lt;=</code> and <code>&gt;=</code>.</p>	

# 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

# 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"));
```



Compiled  
into

# 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");  
}
```



# Unit Testing

# Unit Testing

- In C# there a number of tools for unit testing
- Some of the common tools are MSTest, Nunit, and xUnit
  - We will use the later - xUnit

# Unit Testing

```
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

```
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

1

Make it Fail

- No code without a failing test

2

Make it Work

- As simply as possible

3

Make it  
Better

- Refactor

Unit Testing makes your  
developer lives easier

---

Easier to find bugs

---

Easier to maintain

---

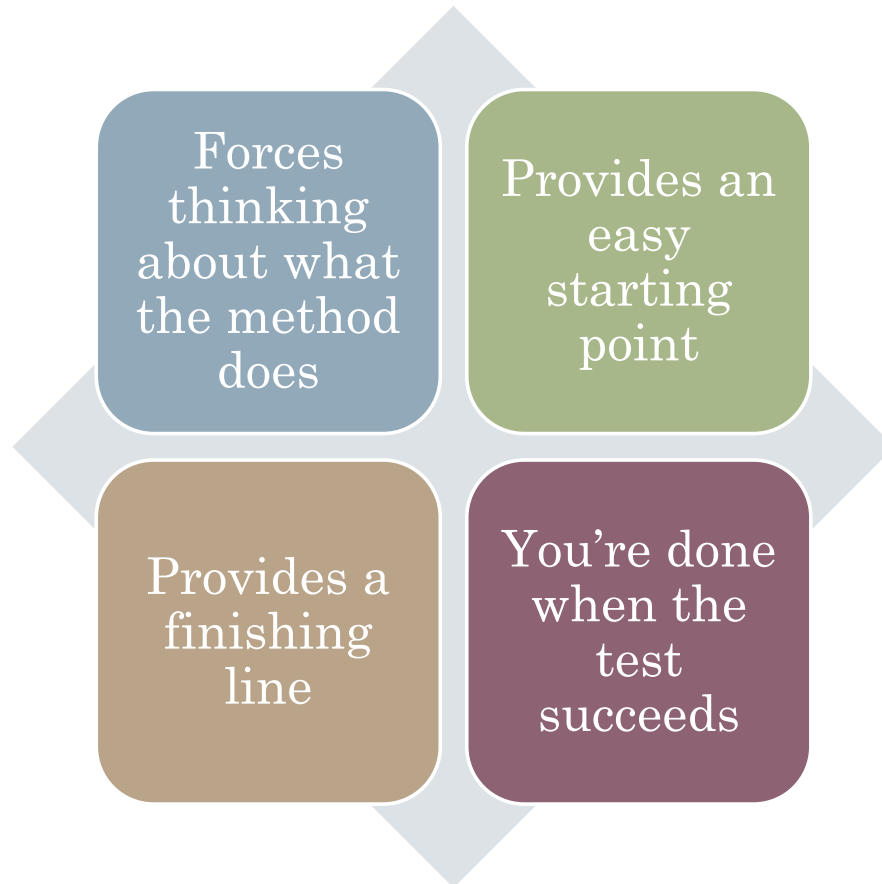
Easier to understand

---

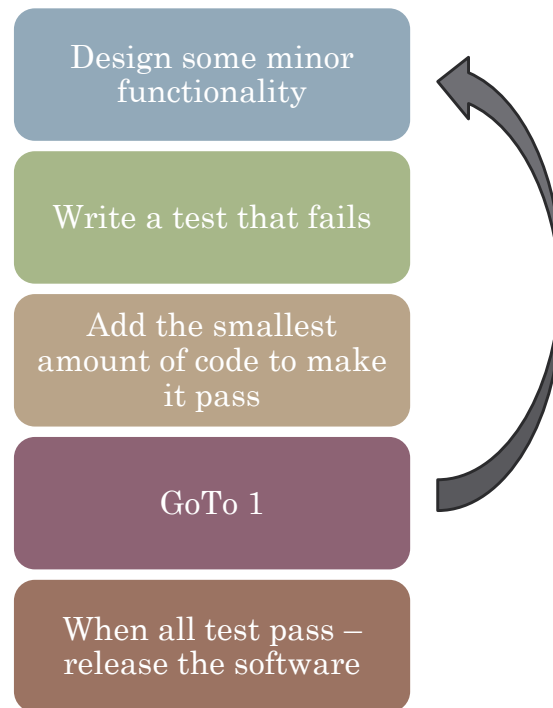
Easier to Develop



# Start with the test

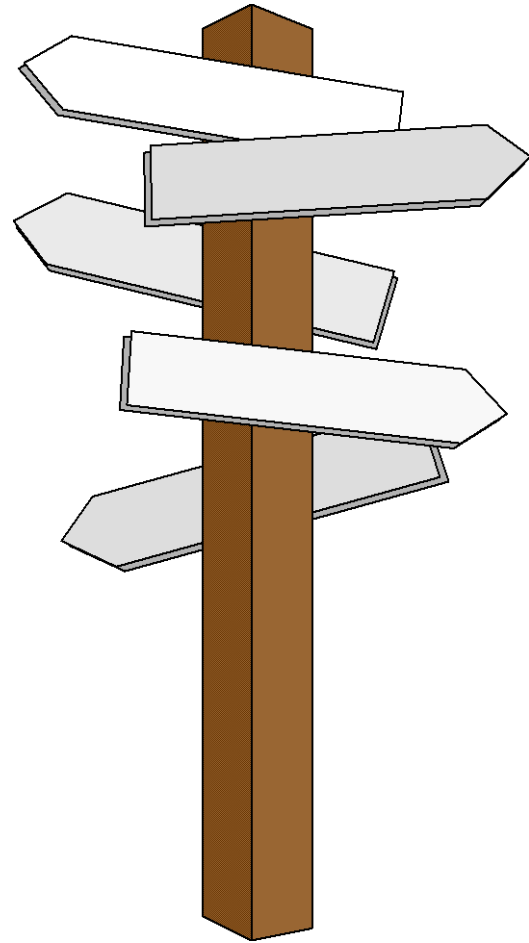


# What are the steps?



# 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?

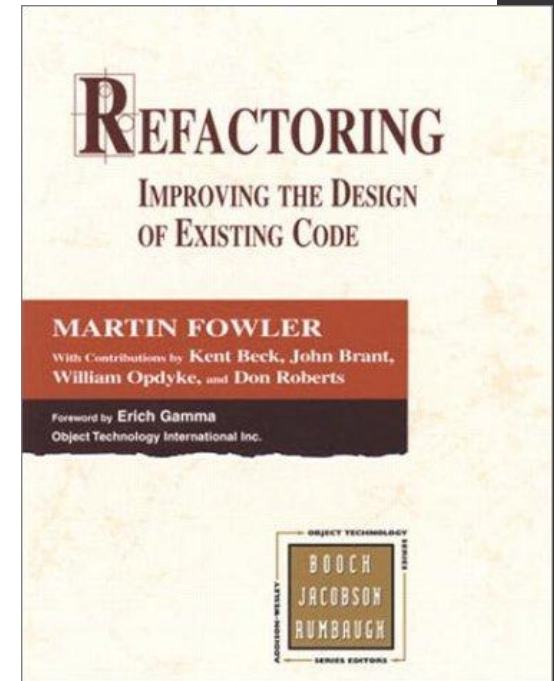
- Refactoring is a technique which allows for the re-structuring of object-oriented code into classes and methods that are more readable, modifiable, and generally sparse in code. Refactoring yields a “better” design.
- The process of changing a software system in a way that preserves the external behavior 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.

# 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

- <https://xunit.net/>
- [nunit.org](https://nunit.org)
- [www.testdriven.com](https://www.testdriven.com)
- [www.refactoring.com](https://www.refactoring.com)
- [c2.com/cgi/wiki?WhatIsRefactoring](https://c2.com/cgi/wiki?WhatIsRefactoring)