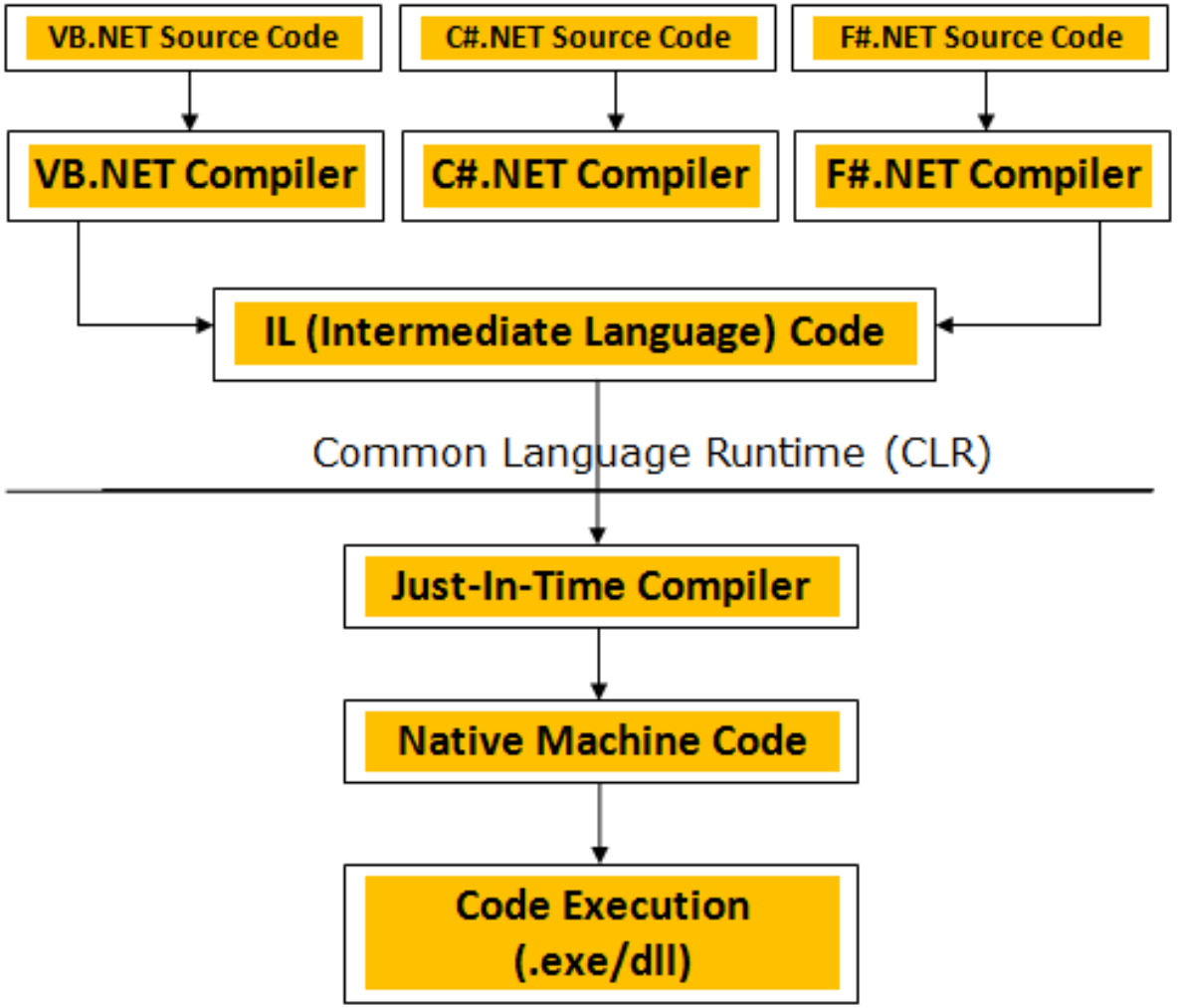
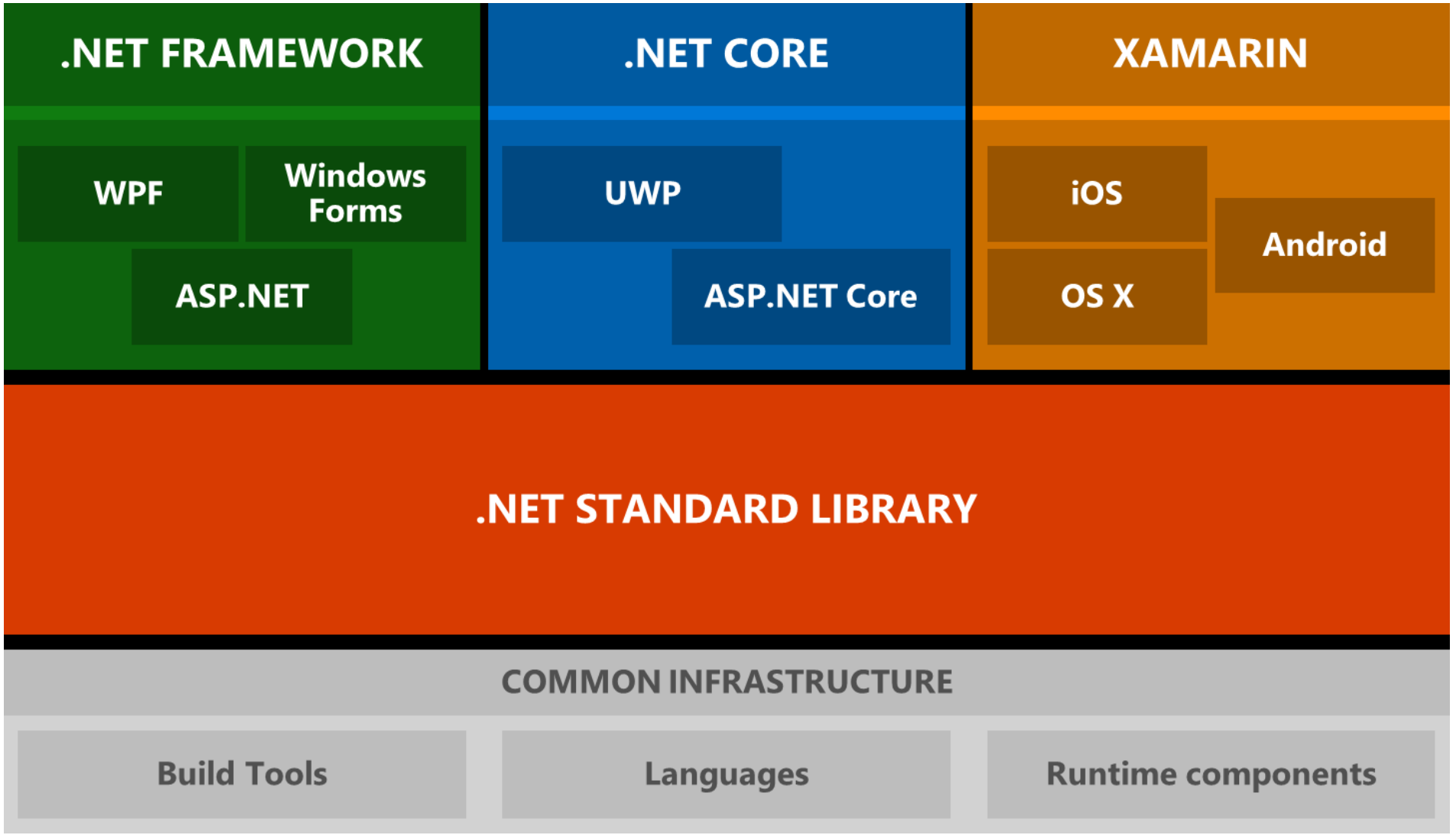
C#

Olivier.leguyader@smallbard.com

CLR


VB.NET Source Code
C#.NET Source Code
F#.NET Source Code
VB.NET Compiler
C#.NET Compiler
F#.NET Compiler
IL (Intermediate Language) Code
Common Language Runtime (CLR)
Just-In-Time Compiler
Native Machine Code
Code Execution
(.exe/dll)

.Net


.NET FRAMEWORK
.NET CORE
XAMARIN
WPF
Windows Forms
ASP.NET
UWP
ASP.NET Core
iOS
Android
OS X
.NET STANDARD LIBRARY
COMMON INFRASTRUCTURE
Build Tools
Languages
Runtime components

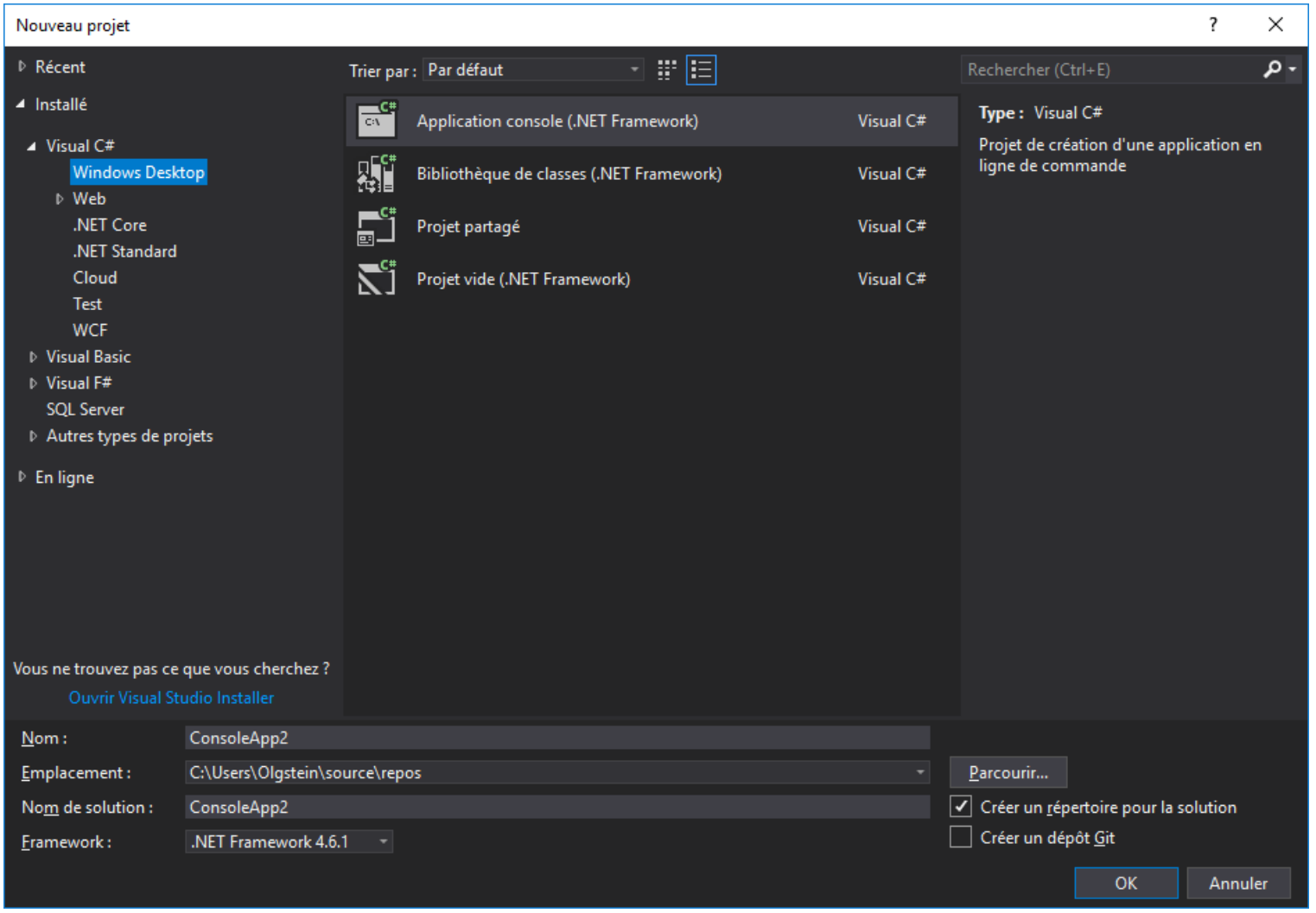
Outils de développement

Visual Studio Code

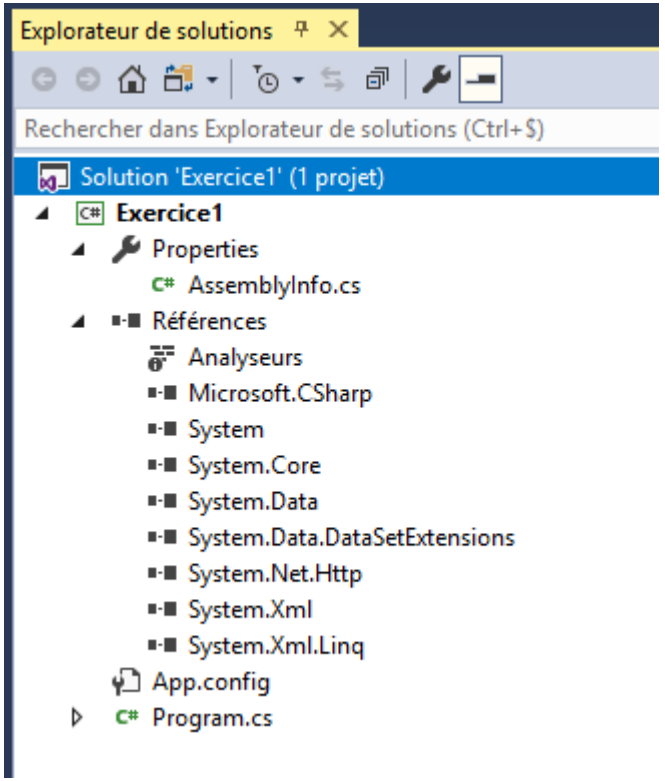
Visual Studio for Mac

JetBrains Rider

Premier programme

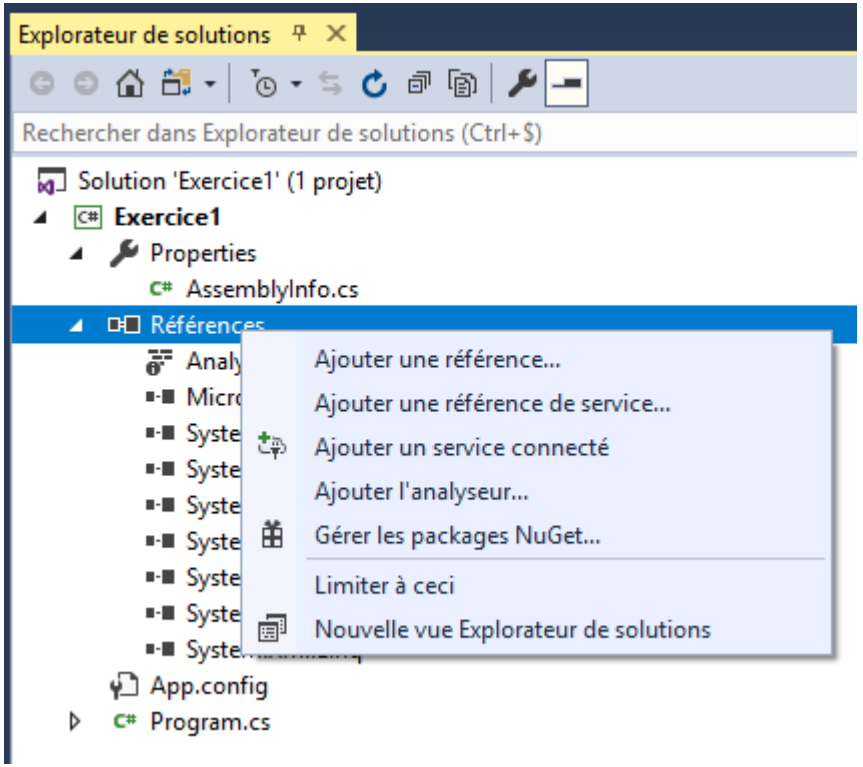


Premier programme



Assembly : .dll ou .exe

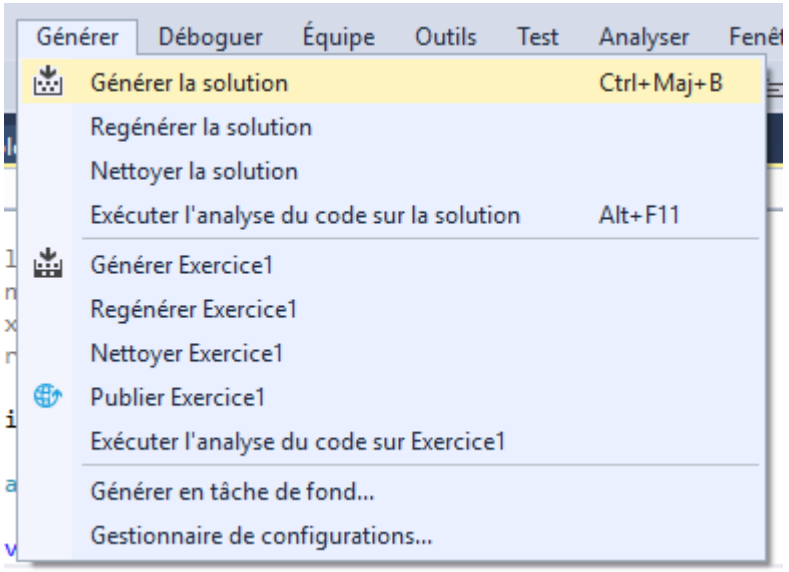
Premier programme

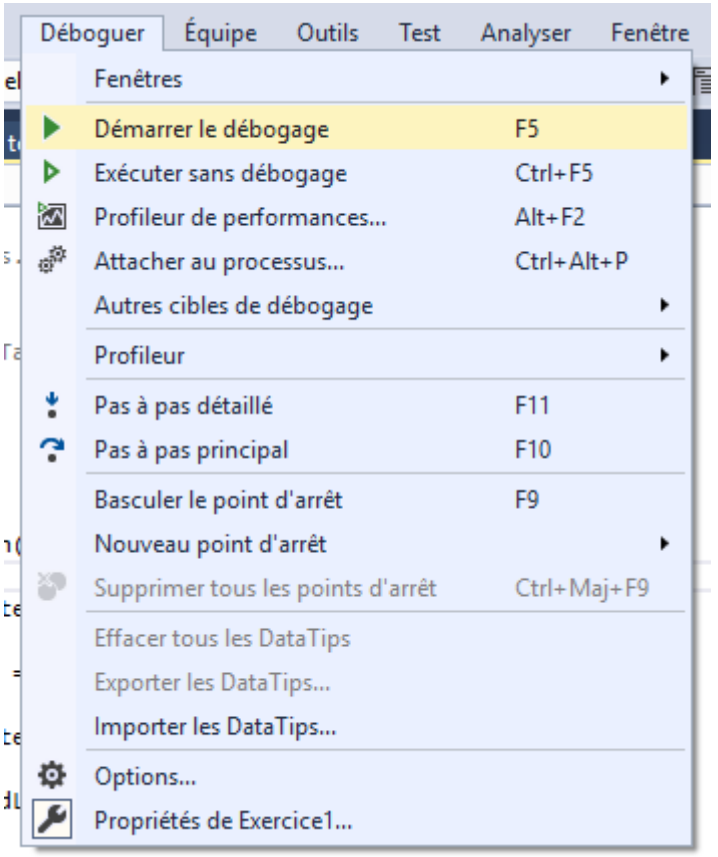


Premier programme

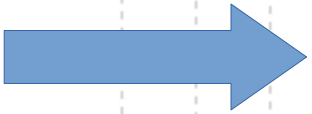
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Exercice1
{
    class Program
    {
        /// <summary>
        /// Points d'entrée de l'application.
        /// </summary>
        /// <param name="args">Arguments de la ligne de commande.</param>
        static void Main(string[] args)
        {
            // Affiche une question
            Console.WriteLine("Quel votre nom ?");

            // Obtient la saisie de l'utilisateur
            string name = Console.ReadLine();

            Console.WriteLine("Bonjour " + name);

            // Pour éviter la fermeture immédiate du programme
            Console.ReadLine();
        }
    }
}
```

Premier programme





Variables



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Exercice1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Quel votre nom ?");

            string name = Console.ReadLine();

            Console.WriteLine("Bonjour " + name);

            Console.ReadLine();
        }
    }
}
```

Variables

Type	*Bytes*	*Range of Values*	*In Use*
`sbyte`	1	–128 to 127	`sbyte sb = 12;`
`byte`	1	0 to 255	`byte b = 12;`
`short`	2	–32,768 to 32,767	`short sh = 12345;`
`ushort`	2	0 to 65,535	`ushort ush = 62345;`
`int`	4	–2 billion to 2 billion	`int n = 1234567890;`
`uint`	4	0 to 4 billion (exact values listed in the Cheat Sheet on this book's website)	`uint un = 3234567890U`
`long`	8	-10^{20} to 10^{20} — "a whole lot"	`long l = 123456789012L`
`ulong`	8	0 to 2×10^{20}	`long ul = 123456789012UL`

Type	*Bytes*	*Range of Values*	*Accuracy to Number of Digits*	*In Use*
`float`	8	$1.5 * 10^{-45}$ to $3.4 * 10^{38}$	6 to 7	`float f = 1.2F;`
`double`	16	$5.0 * 10^{-324}$ to $1.7 * 10^{308}$	15 to 16	`double d = 1.2;`

Variables

```
decimal v = 5.5M;
```

10^-28 à 10^28

```
bool b = true;
```

true ou false

```
char c = 'a';
```

\n, \t, \0, \r, \\

```
string s = "exemple";
string s2 = @"ligne1
ligne2";
string s3 = $"Bonjour {name}";
```

Inférence de type

```
int i = 5;
string s = "exemple";
double d = 2.0;
```

```
var i = 5;
var s = "exemple";
var d = 2.0;
```

Inférence de type

```
int i = 5;
string s = "exemple";
double d = 2.0;
```

```
var i = 5;
var s = "exemple";
var d = 2.0;
```

```
dynamic i = 5;
dynamic s = "exemple";
dynamic d = 2.0;
```

Opérateurs

- Arithmétique : -, *, /, +, %, ++, --
- Affectation : =
- Comparaison : ==, >, >=, <, <=, !=
- Binaire et logique : !, &, |, ^, &&, ||

Instruction de sélection

```
if (condition)
{
    // si la condition vaut true
}
```

```
if (condition)
{
    // si la condition vaut true
}
else
{
    // si la condition vaut false
}
```

Instruction de sélection

```
switch(expression)
{
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    case 3:
        // ...
        break;
    default:
        // ...
        break;
}
```

Instruction d'itération

```
while (condition)
{
    // exécuté tant que la condition vaut true
}

do
{
    // exécuté une fois puis tant que la condition vaut true
}
while (condition);
```

Instruction d'itération

```
for(int i = 0; i < 10; i++)
{
    // exécuté 10 fois
}
```

Instruction d’itération

- L’instruction break permet de forcer la sortie d’une boucle
- L’instruction continue permet de forcer le passage à l’itération suivante

Exercice

- A l'aide des exemples de codes suivants, réaliser un programme qui demande à l'utilisateur de deviner un nombre entre 1 et 100 (en lui indiquant « plus petit » ou « plus grand »)

```
var rnd = new Random();
var nb = rnd.Next(100) + 1;
```

```
int value = Convert.ToInt32(Console.ReadLine());
```

Exercice

```
class Program
{
    static void Main(string[] args)
    {
        var rnd = new Random();
        var nb = rnd.Next(100) + 1;

        while (true)
        {
            Console.WriteLine("Saisissez un nombre entre 1 et 100 (compris ) :");
            int value = Convert.ToInt32(Console.ReadLine());

            if (value > nb)
            {
                Console.WriteLine("Trop grand");
            }
            else if (value < nb)
            {
                Console.WriteLine("Trop petit");
            }
            else
            {
                Console.WriteLine("Vous avez trouvé! Appuyez sur entrée pour quitter");
                Console.ReadLine();
                break;
            }
        }
    }
}
```

Programmation Orientée Objet

- Abstraction
- Classification
- Interface
- Contrôle d’accès

- Pourquoi ?
  - Maintenabilité
  - Ré utilisabilité

Contenu d’une classe

- Champs (données membres)

```
public class MaClasse
{
    private int _value;
    private MaClasse _instance;
    private float _aFloat = 2.0F;
    private readonly string _welcome = "bienvenue";

}
```

Contenu d'une classe

- Constructeurs

```
public class MaClasse
{
    private int _value;

    public MaClasse()
    {
        _value = 0;
    }

    public MaClasse(int initialValue)
    {
        _value = initialValue;
    }

    public MaClasse(string initialValue) : this(Convert.ToInt32(initialValue))
    { }

    // ...

}
```

Contenu d’une classe

- Méthodes

```
public int Add(int value)
{
    return _value + value;
}

public int Add(float value)
{
    return _value + (int)value;
}

public void Reset()
{
    _value = 0;
}

public int Substract(int value) => _value -= value;
```

Contenu d'une classe

- Propriétés

```
public int Value
{
    get { return _value; }
    set { _value = value; }
}

public int ReadValue
{
    get { return _value; }
}

public int AutoValue { get; set; }

public int ReadAutoValue { get; } = 5;

public int Value2
{
    get => _value;
    set => _value = value;
}
```

Exercice

- Réaliser une classe pour le jeu précédent
  - Constructeur par défaut : obtention de la valeur aléatoire et valeur maximale 100
  - Constructeur avec en paramètre la valeur maximale
  - Méthode Play pour lancer le jeu
  - Modifier la méthode main pour utiliser cette classe

Exercice

```
class Program
{
    static void Main(string[] args)
    {
        var game = new Game();
        game.Play();
    }
}
```

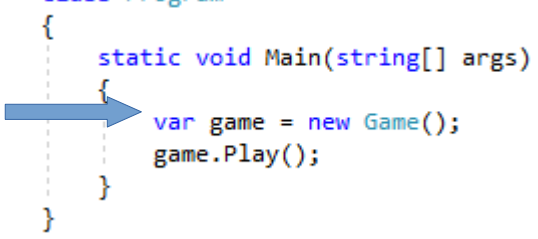
```
public class Game
{
    private readonly int _randomValue;

    public Game(int maxValue)
    {
        var rnd = new Random();
        _randomValue = rnd.Next(maxValue) + 1;
    }

    public Game()
        : this(100)
    { }

    public void Play()
    {
        while (true)
        {
            Console.WriteLine("Saisissez un nombre entre 1 et 100 (compris ) :");
            int value = Convert.ToInt32(Console.ReadLine());

            if (value > _randomValue)
            {
                Console.WriteLine("Trop grand");
            }
            else if (value < _randomValue)
            {
                Console.WriteLine("Trop petit");
            }
            else
            {
                Console.WriteLine("Vous avez trouvé! Appuyez sur entrée pour quitter");
                Console.ReadLine();
                break;
            }
        }
    }
}
```

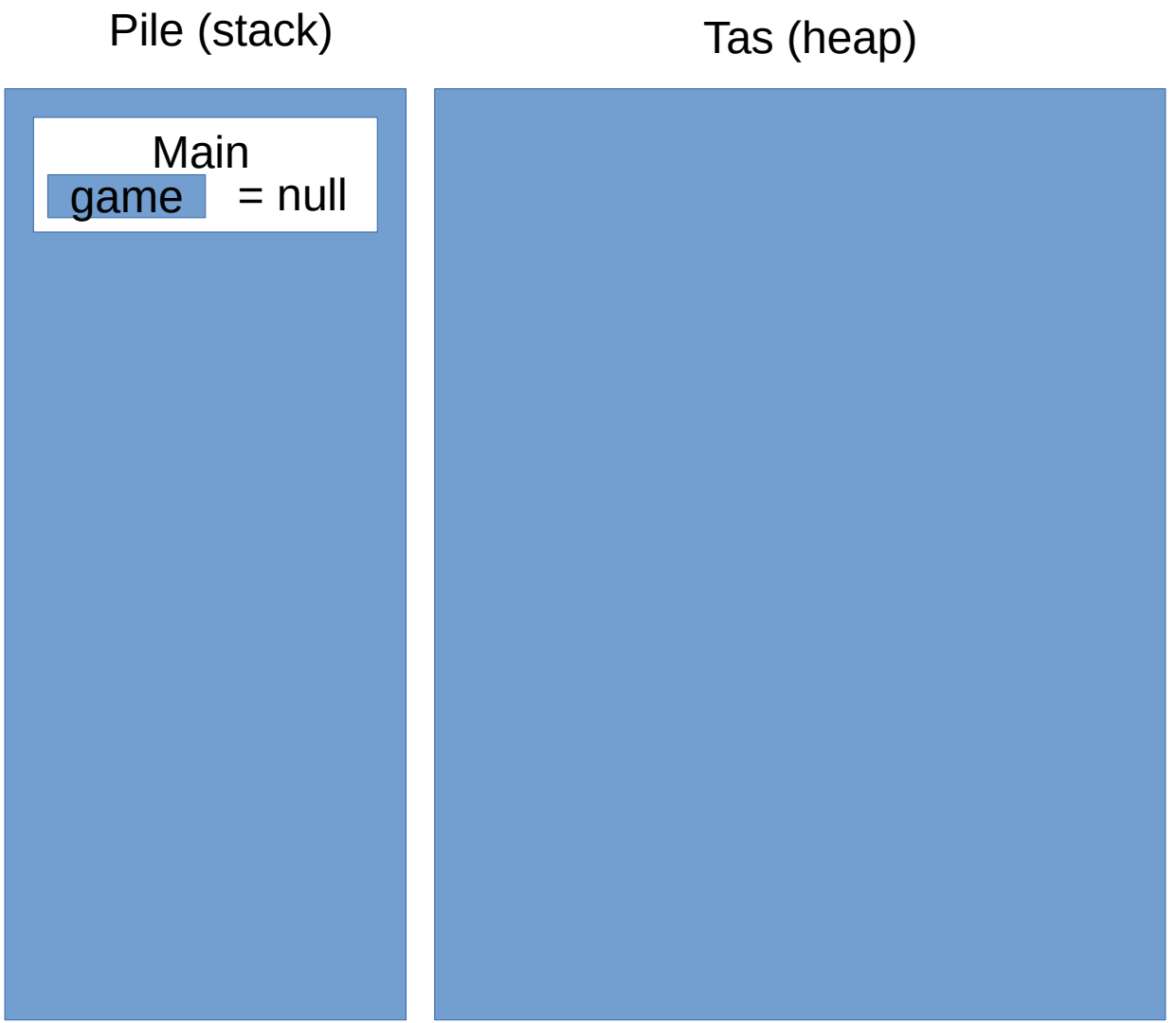
Value types & Reference types

- Value types : int, float, double, char, enum, struct …
  - ~~En mémoire : Toujours sur la pile~~
  - En mémoire : là où ils sont utilisés
- Reference types : classes
  - En mémoire : Toujours dans le tas

Value types & Reference types

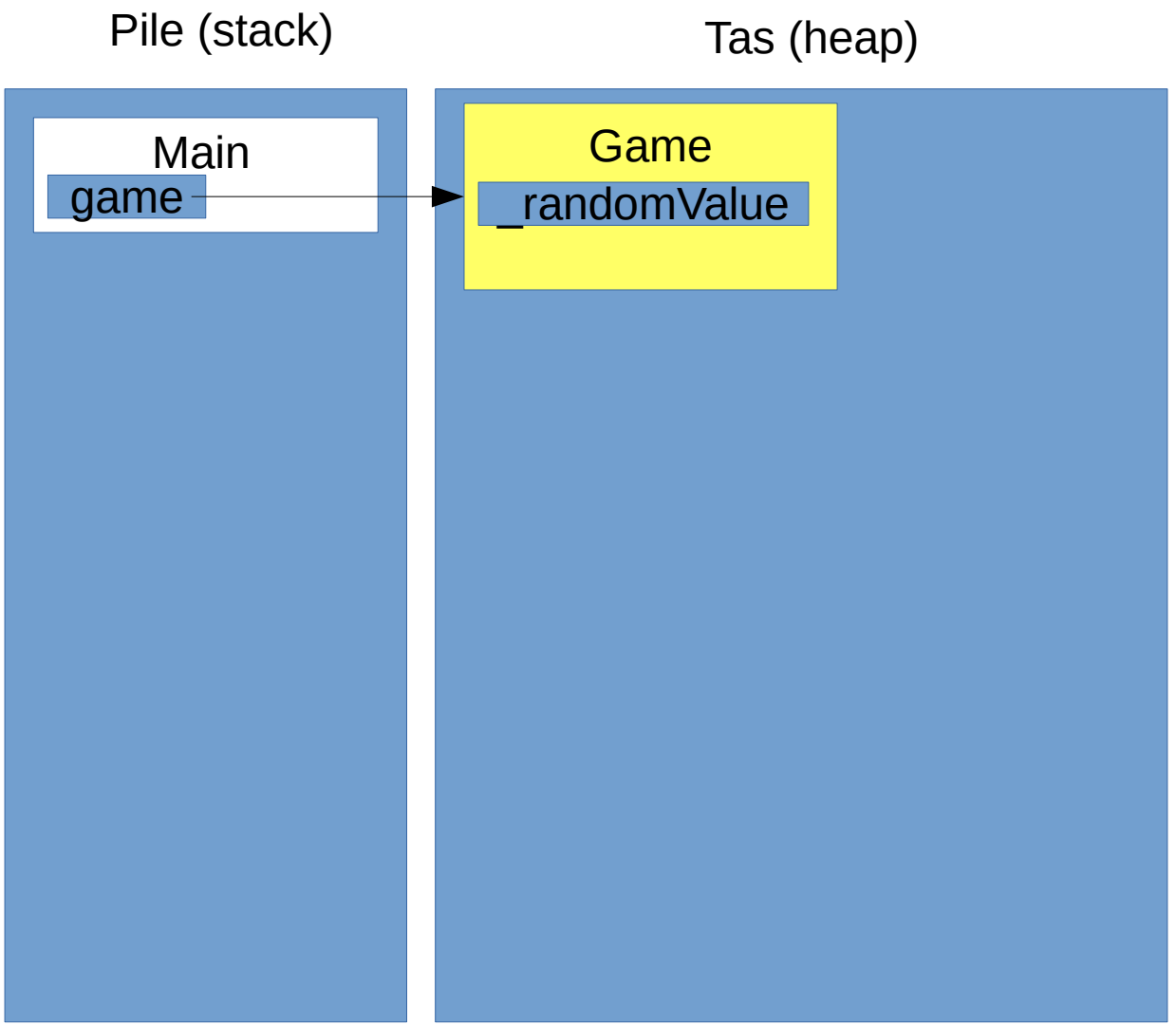
```
class Program
{
    static void Main(string[] args)
    {
        var game = new Game();
        game.Play();
    }
}
```





Value types & Reference types

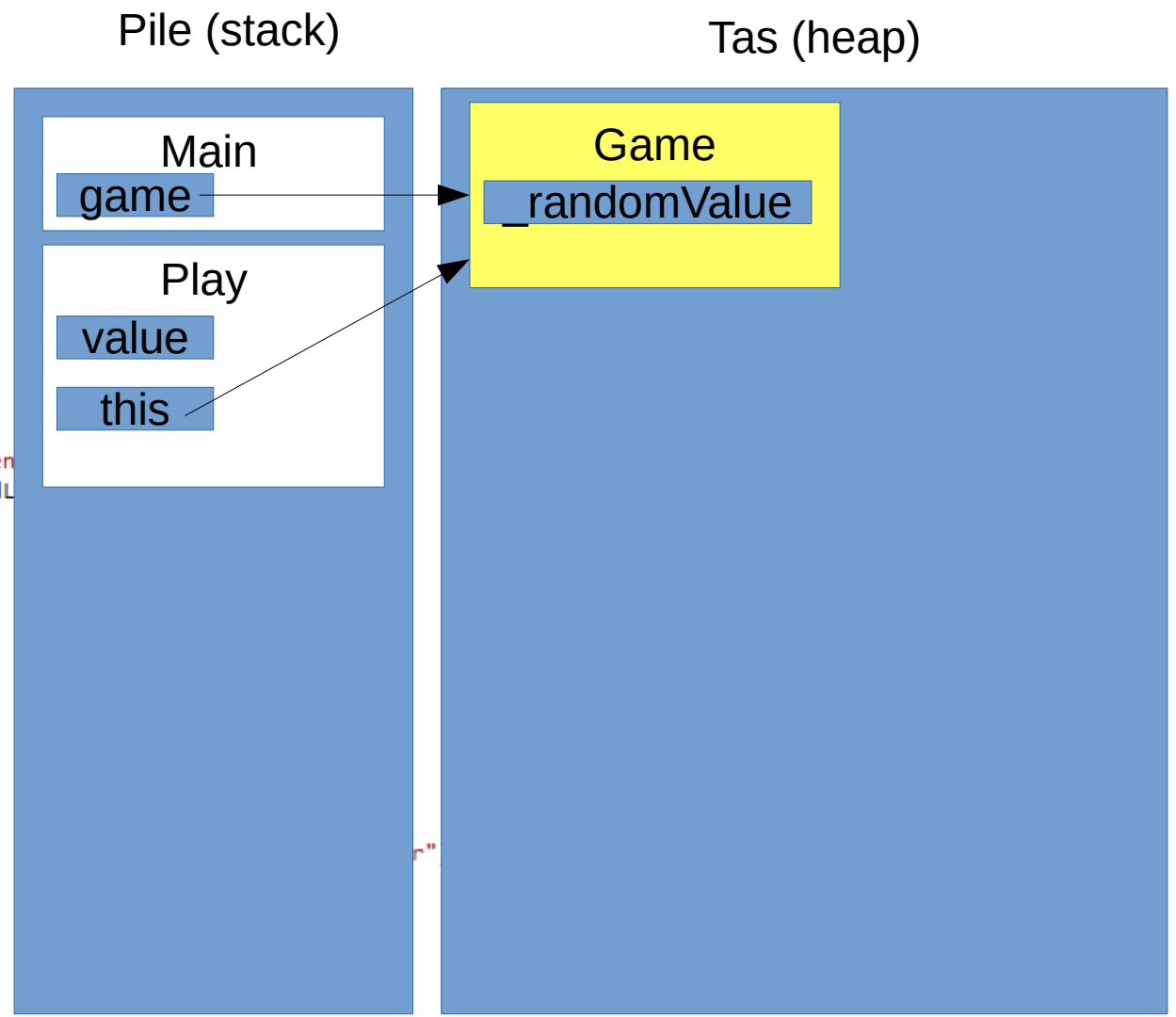
```
class Program
{
    static void Main(string[] args)
    {
        var game = new Game();
        game.Play();
    }
}
```



Value types & Reference types

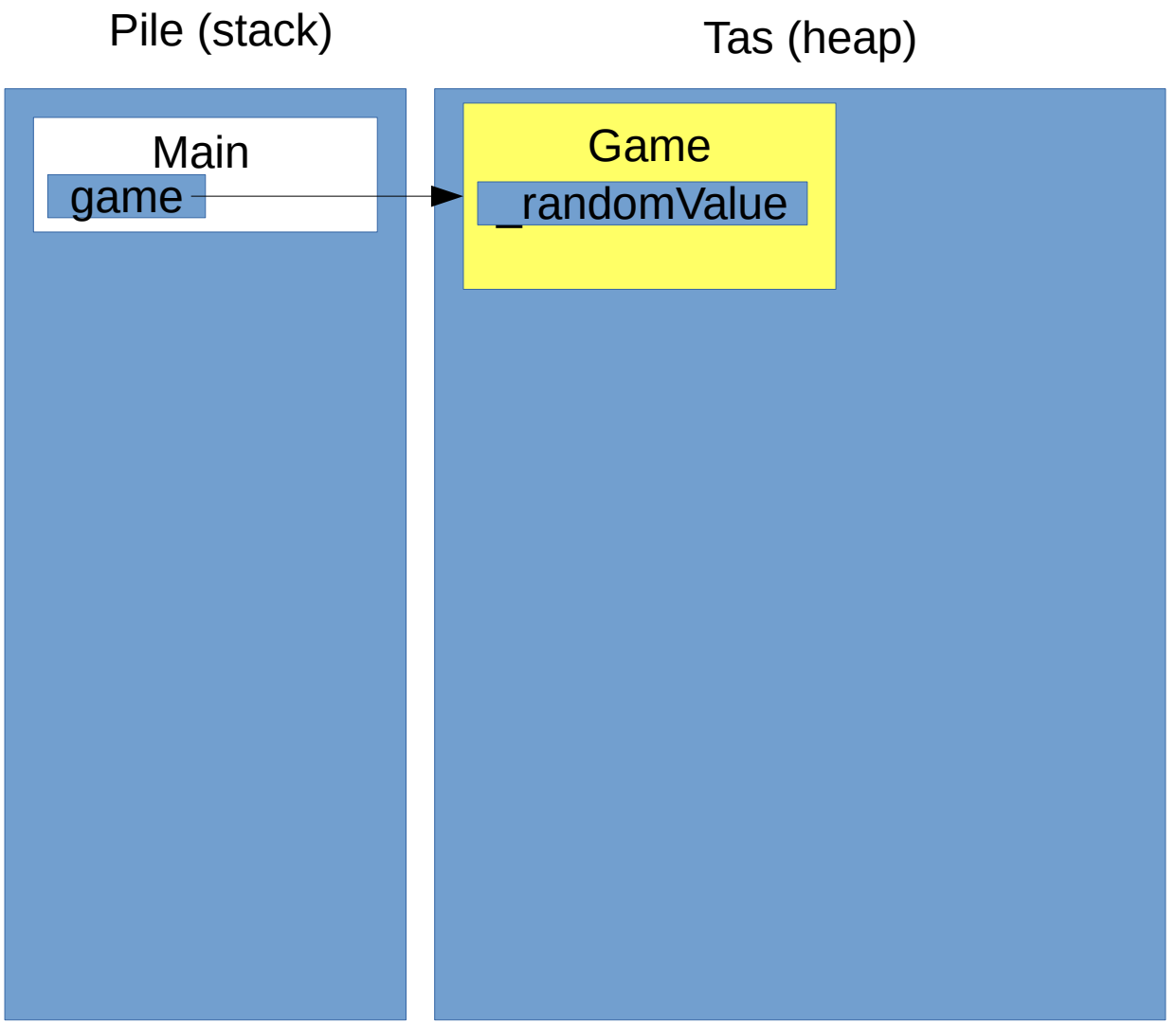
```
class Program
{
    static void Main(string[] args)
    {
        var game = new Game();
        game.Play();
    }
}
```

```
public void Play()
{
    while (true)
    {
        Console.WriteLine("Saisissez un nombre en
        int value = Convert.ToInt32(Console.ReadL

        if (value > _randomValue)
        {
            Console.WriteLine("Trop grand");
        }
        else if (value < _randomValue)
        {
            Console.WriteLine("Trop petit");
        }
        else
        {
            Console.WriteLine("Vous avez trouvé!
            Console.ReadLine();
            break;
        }
    }
}
```



Value types & Reference types

```
class Program
{
    static void Main(string[] args)
    {
        var game = new Game();
        game.Play();
    }
}
```



Value types & Reference types

- Value types :

  passage par copie

- Reference types :

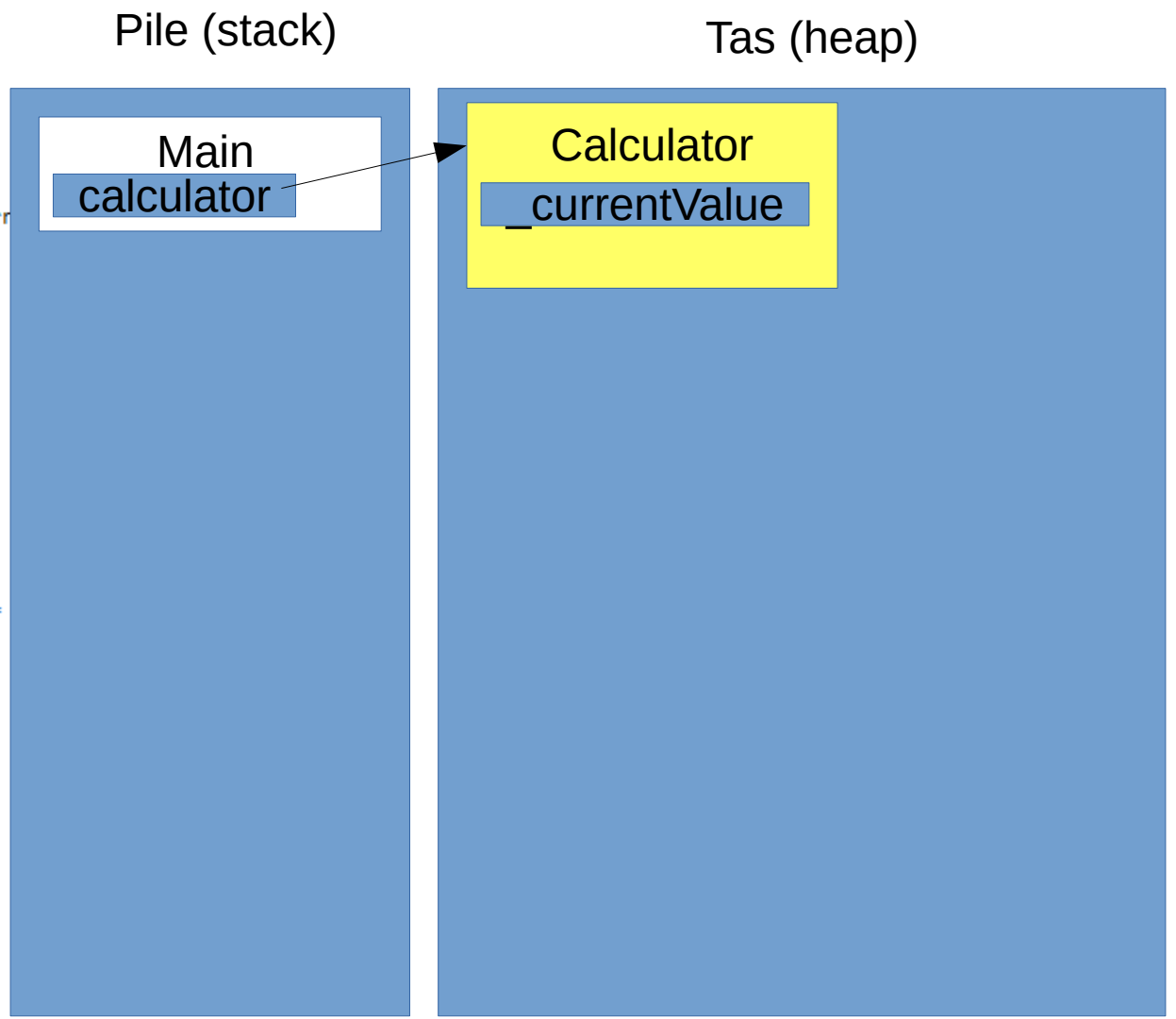
  passage par référence

Value types & Reference types

```
class Program
{
    static void Main(string[] args)
    {
        var calculator = new Calculator();

        AddFive(calculator);

        Console.WriteLine($"Value : {calculator.CurrentValue}");
        Console.ReadLine();
    }

    public static void AddFive(Calculator calc)
    {
        var five = 5;
        calc.Add(five);
    }
}

public class Calculator
{
    public int CurrentValue { get; private set; } = 0;

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}
```

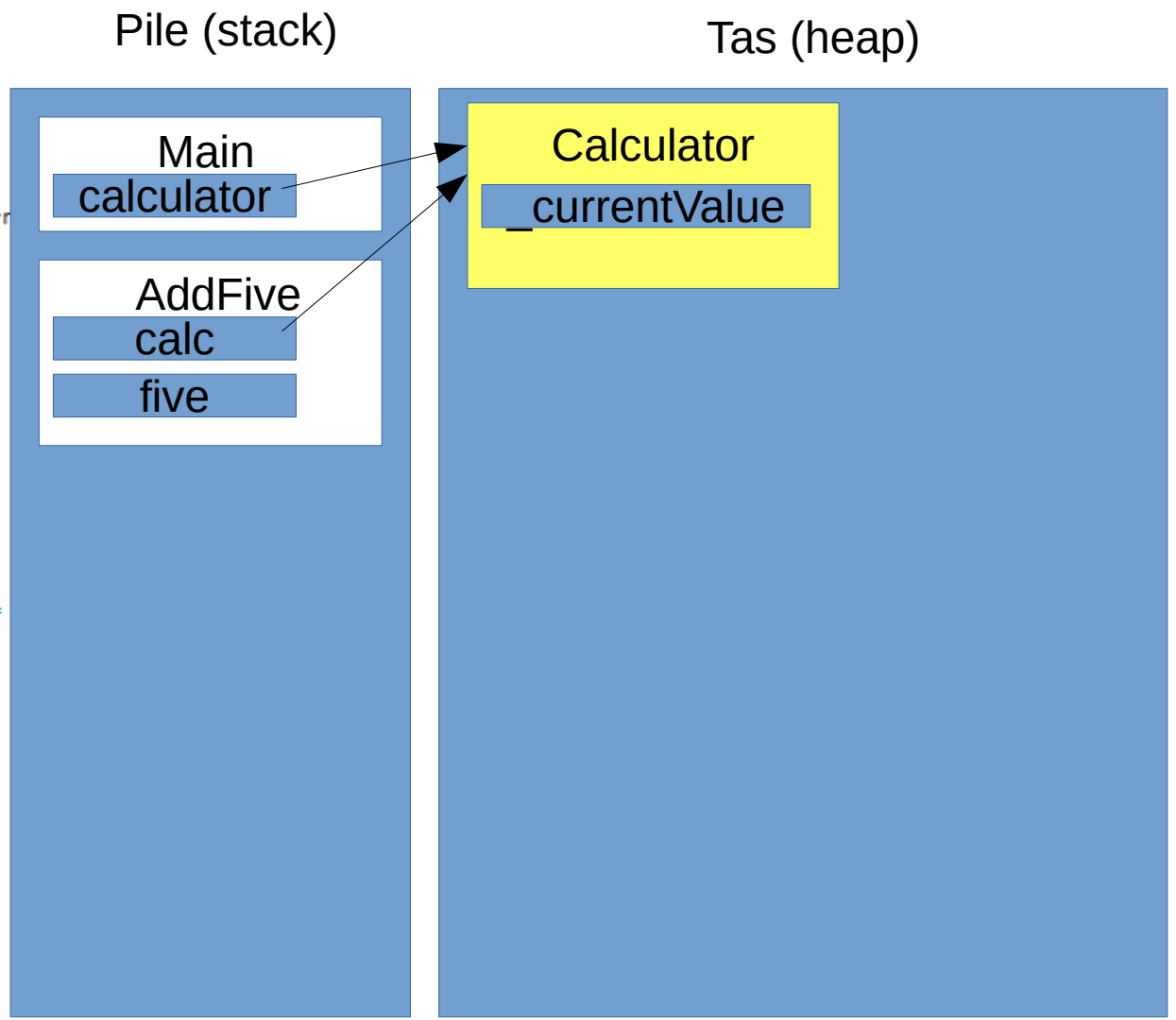
Value types & Reference types

```
class Program
{
    static void Main(string[] args)
    {
        var calculator = new Calculator();

        AddFive(calculator);

        Console.WriteLine($"Value : {calculator.Curr
        Console.ReadLine();
    }

    public static void AddFive(Calculator calc)
    {
        var five = 5;
        calc.Add(five);
    }
}

public class Calculator
{
    public int CurrentValue { get; private set; } =

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}
```



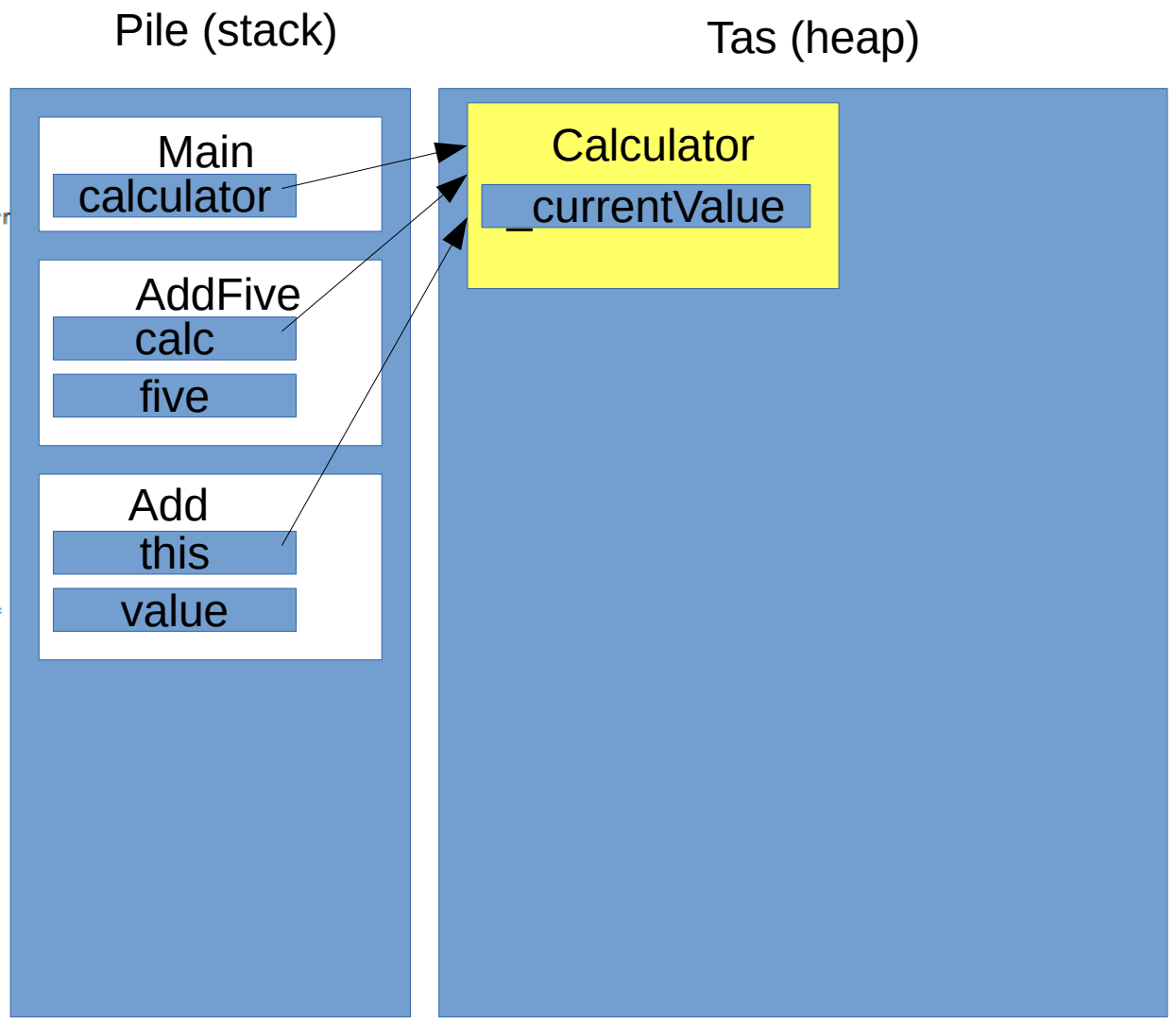
Value types & Reference types

```
class Program
{
    static void Main(string[] args)
    {
        var calculator = new Calculator();

        AddFive(calculator);

        Console.WriteLine($"Value : {calculator.Curr
        Console.ReadLine();
    }

    public static void AddFive(Calculator calc)
    {
        var five = 5;
        calc.Add(five);
    }
}

public class Calculator
{
    public int CurrentValue { get; private set; } =

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}
```

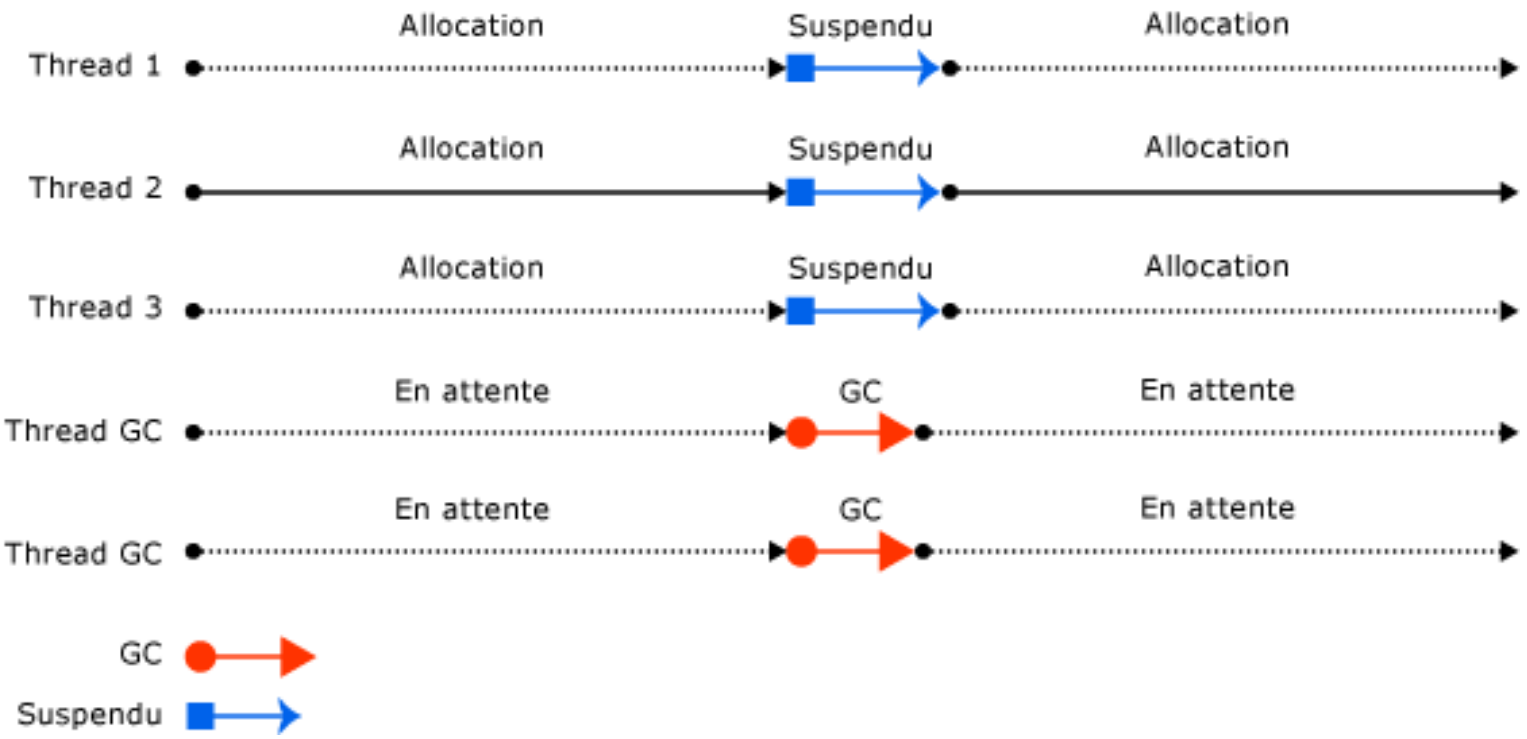


Value types & Reference types

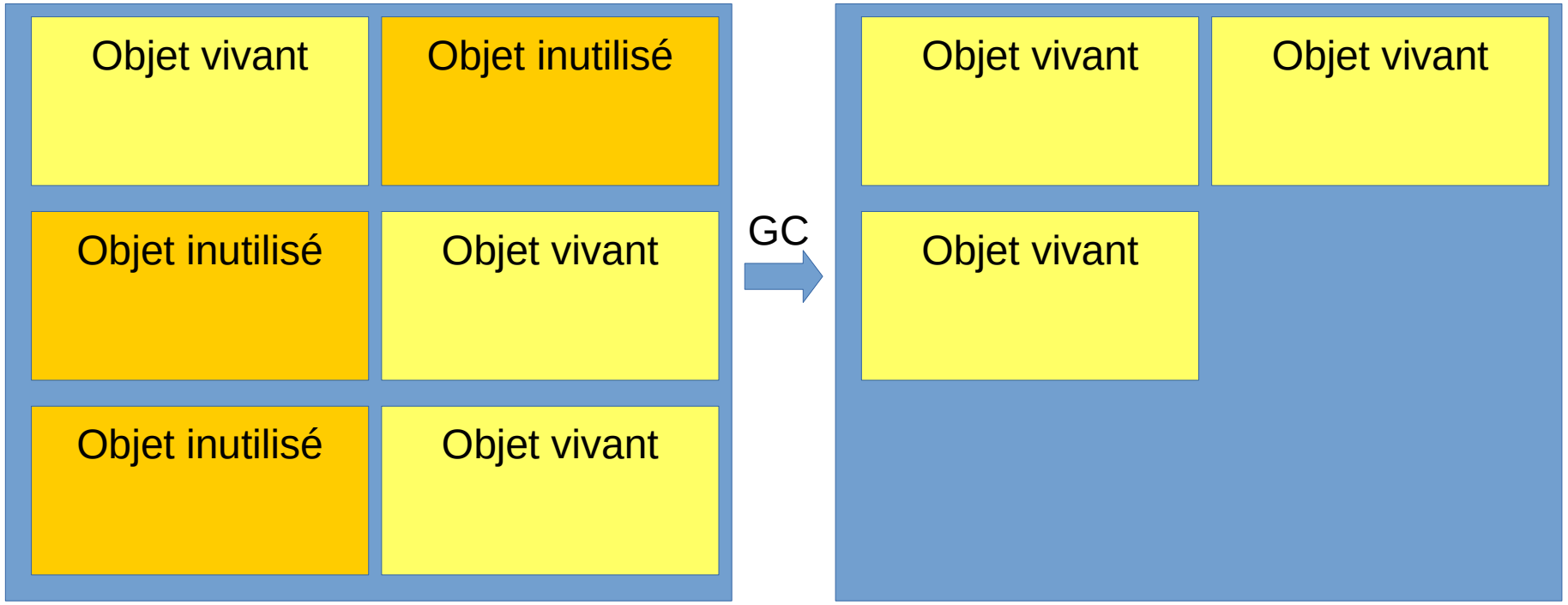
```
class Program
{
    static void Main(string[] args)
    {
        var calculator = new Calculator();

        AddFive(calculator);

        Console.WriteLine($"Value : {calculator.Curr
        Console.ReadLine();
    }

    public static void AddFive(Calculator calc)
    {
        var five = 5;
        calc.Add(five);
    }
}

public class Calculator
{
    public int CurrentValue { get; private set; } =

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}
```



Ramasse miette (garbage collector)



Ramasse miette (garbage collector)



Héritage

```
public class Calculator
{
    public int CurrentValue { get; private set; } = 0;

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}

public class AdvanceCalculator : Calculator
{
    public void Multiply(int value)
    {
        CurrentValue *= value;
    }

    public void Divide(int value)
    {
        CurrentValue *= value;
    }
}
```

Héritage

```
public class Calculator
{
    public int CurrentValue { get; protected set; } = 0;

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}

public class AdvanceCalculator : Calculator
{
    public void Multiply(int value)
    {
        CurrentValue *= value;
    }

    public void Divide(int value)
    {
        CurrentValue *= value;
    }
}
```

Héritage

```
public class Calculator
{
    public int CurrentValue { get; protected set; } = 0;

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}

public class LogCalculator : Calculator
{
    public new void Add(int value)
    {
        Console.WriteLine($"Add {value}");
        base.Add(value);
    }

    public new void Substract(int value)
    {
        Console.WriteLine($"Substract {value}");
        base.Substract(value);
    }
}
```

Héritage

```
public class Calculator
{
    public int CurrentValue { get; protected set; } = 0;

    public void Add(int value)
    {
        CurrentValue += value;
    }

    public void Substract(int value)
    {
        CurrentValue -= value;
    }
}

public class LogCalculator : Calculator
{
    public new void Add(int value)
    {
        Console.WriteLine($"Add {value}");
        base.Add(value);
    }

    public new void Substract(int value)
    {
        Console.WriteLine($"Substract {value}");
        base.Substract(value);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Calculator calculator = new LogCalculator();

        calculator.Add(5);
        calculator.Add(3);

        Console.ReadLine();
    }
}
```

Héritage

```
public class Calculator
{
    public int CurrentValue { get; protected set; } = 0;

    public virtual void Add(int value)
    {
        CurrentValue += value;
    }

    public virtual void Substract(int value)
    {
        CurrentValue -= value;
    }
}

public class LogCalculator : Calculator
{
    public override void Add(int value)
    {
        Console.WriteLine($"Add {value}");
        base.Add(value);
    }

    public override void Substract(int value)
    {
        Console.WriteLine($"Substract {value}");
        base.Substract(value);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Calculator calculator = new LogCalculator();

        calculator.Add(5);
        calculator.Add(3);

        Console.ReadLine();
    }
}
```

Héritage

- Toutes les classes héritent par défaut de la classe Object (ToString, Equals, GetHashcode)
- Une classe ne peut hériter que d’une seule classe
- Les méthodes sont par défaut non virtuelles

Héritage

```
public class Calculator
{
    public int CurrentValue { get; protected set; }

    public Calculator(int initialValue)
    {
        CurrentValue = initialValue;
    }

    public virtual void Add(int value)
    {
        CurrentValue += value;
    }

    public virtual void Substract(int value)
    {
        CurrentValue -= value;
    }
}

public class LogCalculator : Calculator
{
    public override void Add(int value)
    {
        Console.WriteLine($"Add {value}");
        base.Add(value);
    }

    public override void Substract(int value)
    {
        Console.WriteLine($"Substract {value}");
        base.Substract(value);
    }
}
```

Héritage

```
public class Calculator
{
    public int CurrentValue { get; protected set; }

    public Calculator(int initialValue)
    {
        CurrentValue = initialValue;
    }

    public virtual void Add(int value)
    {
        CurrentValue += value;
    }

    public virtual void Substract(int value)
    {
        CurrentValue -= value;
    }
}

public class LogCalculator : Calculator
{
    public LogCalculator(int initialValue)
        : base(initialValue)
    { }

    public override void Add(int value)
    {
        Console.WriteLine($"Add {value}");
        base.Add(value);
    }

    public override void Substract(int value)
    {
        Console.WriteLine($"Substract {value}");
        base.Substract(value);
    }
}
```

Exercice

- Modifier la classe du jeu précédent afin qu'elle soit extensible par héritage
- Sans duplication de code, créer une nouvelle classe de jeu héritant de la première et ajoutant les comportements suivants :
  - Un nombre d'essai maximum limité
  - La possibilité de quitter le jeu en saisissant -1

Exercice

```
public class Game
{
    private readonly int _randomValue;

    public Game(int maxValue)
    {
        var rnd = new Random();
        _randomValue = rnd.Next(maxValue) + 1;
    }

    public Game()
        : this(100)
    { }

    public void Play()
    {
        int value = 0;
        do
        {
            Console.WriteLine("Saisissez un nombre entre 1 et 100 (compris ) :");
            value = Convert.ToInt32(Console.ReadLine());
        } while (!PlayTurn(value));
    }

    protected virtual bool PlayTurn(int value)
    {
        if (value > _randomValue)
        {
            Console.WriteLine("Trop grand");
        }
        else if (value < _randomValue)
        {
            Console.WriteLine("Trop petit");
        }
        else
        {
            Console.WriteLine("Vous avez trouvé! Appuyez sur entrée pour quitter");
            Console.ReadLine();
            return true;
        }

        return false;
    }
}
```

Exercice

```
public class AdvancedGame : Game
{
    private readonly int _maximulTrials;
    private int _trials = 0;

    public AdvancedGame(int maximumTrials)
        : base()
    {
        _maximulTrials = maximumTrials;
    }

    protected override bool PlayTurn(int value)
    {
        if (value == -1)
        {
            return true;
        }

        if (!base.PlayTurn(value))
        {
            if (++_trials == _maximulTrials)
            {
                Console.WriteLine("Vous avez atteint le nombre maximum d'essais! Appuyez sur entrée pour quitter");
                Console.ReadLine();
                return true;
            }
        }
        else
            return true;

        return false;
    }
}
```

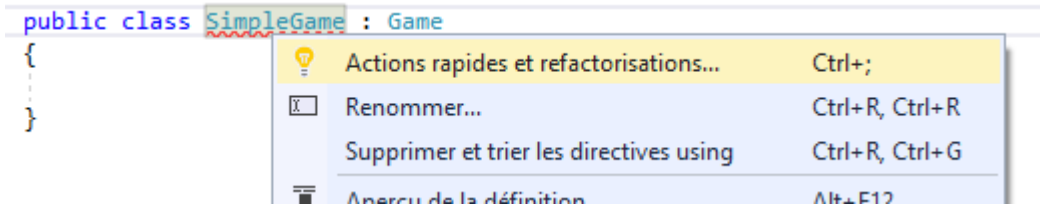
Opérateurs is et as

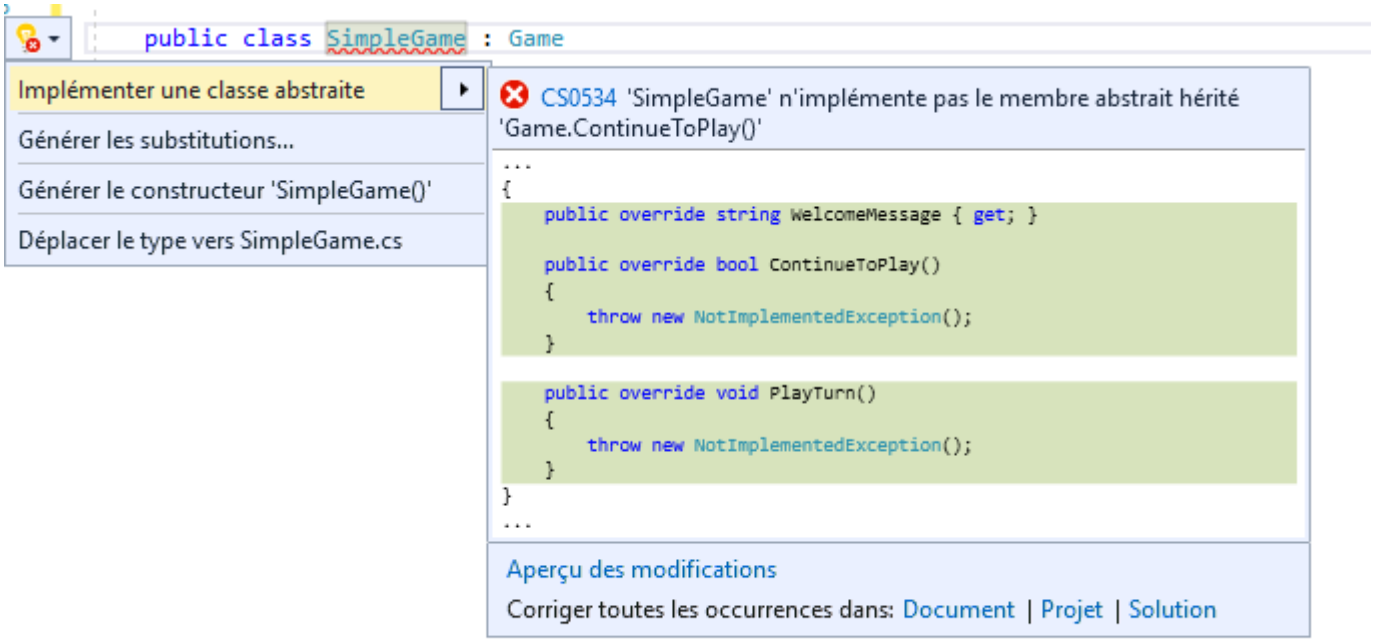
```
Game game // ...

if (game is AdvancedGame)
{
    // ...
}

var advancedGame = game as AdvancedGame;
if (advancedGame != null)
{
    // ...
}
```

Classe abstraite

```
public abstract class Game
{
    public abstract string WelcomeMessage { get; }

    public abstract bool ContinueToPlay();

    public abstract void PlayTurn();

    public void Play()
    {
        Console.WriteLine(WelcomeMessage);

        do
        {
            PlayTurn();
        }
        while (ContinueToPlay());
    }
}
```

Classe abstraite





Classe abstraite

```
public class SimpleGame : Game
{
    public override string WelcomeMessage { get; }

    public override bool ContinueToPlay()
    {
        throw new NotImplementedException();
    }

    public override void PlayTurn()
    {
        throw new NotImplementedException();
    }
}
```

Sealed

```
public sealed class LastGame : Game
{
    public override string WelcomeMessage { get; }

    public override bool ContinueToPlay()
    {
        throw new NotImplementedException();
    }

    public override void PlayTurn()
    {
        throw new NotImplementedException();
    }
}
```

Enum

```
public enum Colors
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Purple
}
```

```
var color = Colors.Red;

var nbColor = (int)color;
```

Enum

```
public enum Colors : short
{
    Red = 10,
    Orange = 20,
    Yellow = 30,
    Green = 40,
    Blue = 50,
    Purple = 60
}
```

```
var color = Colors.Red;
var nbColor = (short)color;
```

Enum

```
[Flags]
public enum Colors
{
    Red = 1,
    Orange = 2,
    Yellow = 4,
    Green = 8,
    Blue = 16,
    Purple = 32
}
```

```
var color = Colors.Red | Colors.Yellow;

if ((color & Colors.Yellow) == Colors.Yellow)
{
    Console.WriteLine("Contient du jaune");
}
```

Tableau

```
int[] array = new int[5];

int[] array2 = { 1, 2, 3, 4, 5 };

Console.WriteLine(array2[0]);

Console.WriteLine(array2.Length);

int[,] array3 = new int[3, 5];
int[,] array4 = { { 1, 2 }, { 2, 3 }, { 4, 5 } };

int[,,,,] array5 = new int[2, 5, 4, 3, 4];
```

Collections

- Espace de noms : System.Collections.Generic
- List<T> : tableau de taille variable
- Queue<T> : FIFO
- Stack<T> : LIFO (pile)
- Dictionary<TKey, Tvalue> : dictionnaire, hashmap
- HashSet<T> : ensemble sans élément dupliqué

List<T>

```
var lst = new List<string>();

lst.Add("aaa");
lst.Add("bbb");
lst.Remove("aaa");
lst.RemoveAt(0);

var lst2 = new List<int>() { 1, 2, 3, 4, 5 };

lst2.Add(5);

Console.WriteLine(lst2[0]);
```

Queue<T>

```
var qe = new Queue<Colors>();

qe.Enqueue(Colors.Red);
var color = qe.Dequeue();
```

Stack<T>

```
var stck = new Stack<int>();

stck.Push(5);
var nb = stck.Pop();

var count = stck.Count;
```

Dictionary<TKey, TValue>

```
var dict = new Dictionary<int, string>();

dict.Add(1, "un");
dict.Add(2, "deux");

Console.WriteLine(dict[1]);

dict.Remove(2);
```

foreach

```
foreach (var key in dict.Keys)
{
    Console.WriteLine(dict[key]);
}

foreach (var i in new int[] { 1, 2, 3, 4, 5 })
{
    Console.WriteLine(i);
}
```

Interfaces

```
public interface IEnumerable<out T> : IEnumerable
{
    //
    // Résumé :
    //     Returns an enumerator that iterates through the collection.
    //
    // Retourne :
    //     An enumerator that can be used to iterate through the collection.
    IEnumerator<T> GetEnumerator();
}
```

Interfaces

```
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    //
    // Résumé :
    //     Gets the element in the collection at the current position of the enumerator.
    //
    // Retourne :
    //     The element in the collection at the current position of the enumerator.
    T Current { get; }
}
```

Interfaces

```
public interface IEnumerator
{
    //
    // Résumé :
    //     Gets the element in the collection at the current position of the enumerator.
    //
    // Retourne :
    //     The element in the collection at the current position of the enumerator.
    object Current { get; }

    //
    // Résumé :
    //     Advances the enumerator to the next element of the collection.
    //
    // Retourne :
    //     true if the enumerator was successfully advanced to the next element; false if
    //     the enumerator has passed the end of the collection.
    //
    // Exceptions :
    //   T:System.InvalidOperationException:
    //     The collection was modified after the enumerator was created.
    bool MoveNext();
    //
    // Résumé :
    //     Sets the enumerator to its initial position, which is before the first element
    //     in the collection.
    //
    // Exceptions :
    //   T:System.InvalidOperationException:
    //     The collection was modified after the enumerator was created.
    void Reset();
}
```

Interfaces

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>, IEnumerable,
   IList, ICollection, IReadOnlyList<T>, IReadOnlyCollection<T>
{
```

Exercice

- Mastermind
  - En utilisant une énumération de couleurs et une énumération pour les résultats
  - Un tableau pour la combinaison
  - Éventuellement des collections

Exercice

```
public class Mastermind
{
    private Colors[] _combination = new Colors[4];
    private int _trials = 0;

    public Mastermind()
    {
        var rnd = new Random();
        for (var i = 0; i < 4; i++)
        {
            _combination[i] = (Colors)rnd.Next(6);
        }
    }

    public void Play()
    {
        var found = false;

        do
        {
            Console.WriteLine("Entrer votre combinaison de 4 chiffres (0: Rouge, 1: Orange, 2: Jaune, 3: Vert, 4: Bleu, 5: Violet)");
            var trial = StringToColors(Console.ReadLine());

            found = CompareTrial(trial);

        } while (++_trials < 12 && !found);

        if (found)
            Console.WriteLine("Bravo!");
        else
            Console.WriteLine("Vous avez perdu.");
    }
```

Exercice

```
public enum Results
{
    NOk = 0,
    Ok = 1,
    Misplaced = 2
}
```

```
public enum Colors
{
    Red = 0,
    Orange = 1,
    Yellow = 2,
    Green = 3,
    Blue = 4,
    Purple = 5
}
```

```
private Colors[] StringToColors(string trial)
{
    var combination = new Colors[4];
    for (var i = 0; i < 4 && i < trial.Length; i++)
    {
        combination[i] = (Colors)int.Parse(trial[i].ToString());
    }

    return combination;
}
```

Exercice

```
private bool CompareTrial(Colors[] trial)
{
    var result = new Results[4];

    // Correctement placé
    for (var i = 0; i < 4; i++)
    {
        if (trial[i] == _combination[i])
            result[i] = Results.Ok;
    }

    // Présent mais incorrectement placé
    var combinationToFound = _combination.ToList();
    for (var i = 0; i < 4; i++)
    {
        if (result[i] == Results.Ok) continue;

        var index = combinationToFound.IndexOf(trial[i]);
        if (index >= 0 && result[index] != Results.Ok)
        {
            result[i] = Results.Misplaced;
            combinationToFound.RemoveAt(index);
        }
    }

    var strResult = string.Empty;
    foreach (var r in result)
    {
        strResult += r.ToString() + " ";
    }

    Console.WriteLine(strResult);
    return result[0] == Results.Ok && result[1] == Results.Ok && result[2] == Results.Ok && result[3] == Results.Ok;
}
```

Génériques

- Avantages :
  - Réutilisabilité
  - Performances

Génériques

```
public class MaList<T>
{
    private T[] _arrayOfT;

    public void Add(T item)
    {
        // ...
    }

    // ...
}
```

Génériques

```
public class MaClasseNonGenerique
{

    public void MaMethodeGenerique<T>(T param)
    {
        // ...
    }

}
```

Génériques

```
public class MaGenerique<T, U>
    where T : new()
    where U : class
{
    // ...
}
```

```
public void MaMethodeGenerique<T,U>(T param1, U param2)
    where T : struct
    where U : T
{
    // ...
}
```

Contrainte
`where T : struct`
`where T : class`
`where T : unmanaged`
`where T : new()`
`where T :` *<nom_classe_de_base>*
`where T :` *<nom_interface>*
`where T : U`

Exercice

- Créer une classe générique PriorityQueue<T> comportant deux méthodes :
  - public void Enqueue(T item) : ajoutant un élément à la file d'attente
  - public T Dequeue() : retirant de la file d'attente les éléments prioritaires
- L'élément T doit implémenter l'interface :

```
public interface IPrioritizable
{
    Priority Priority { get; }
}

public enum Priority { Low, Medium, High }
```

Exercice

```
public class ProrityQueue<T> where T : IPrioritizable
{
    private Queue<T> _queueHigh = new Queue<T>();
    private Queue<T> _queueMedium = new Queue<T>();
    private Queue<T> _queueLow = new Queue<T>();

    public void Enqueue(T item)
    {
        switch(item.Priority)
        {
            case Priority.High:
                _queueHigh.Enqueue(item);
                break;
            case Priority.Medium:
                _queueMedium.Enqueue(item);
                break;
            case Priority.Low:
                _queueLow.Enqueue(item);
                break;
        }
    }

    public T Dequeue()
    {
        if (_queueHigh.Count > 0)
            return _queueHigh.Dequeue();
        if (_queueMedium.Count > 0)
            return _queueMedium.Dequeue();
        if (_queueLow.Count > 0)
            return _queueLow.Dequeue();

        return default(T);
    }
}
```

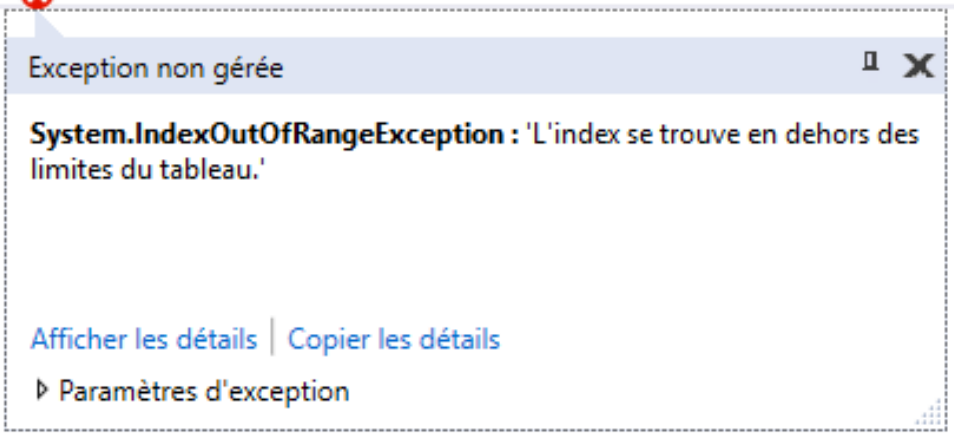
```
public interface IPrioritizable
{
    Priority Priority { get; }
}

public enum Priority { Low, Medium, High }
```

Exceptions

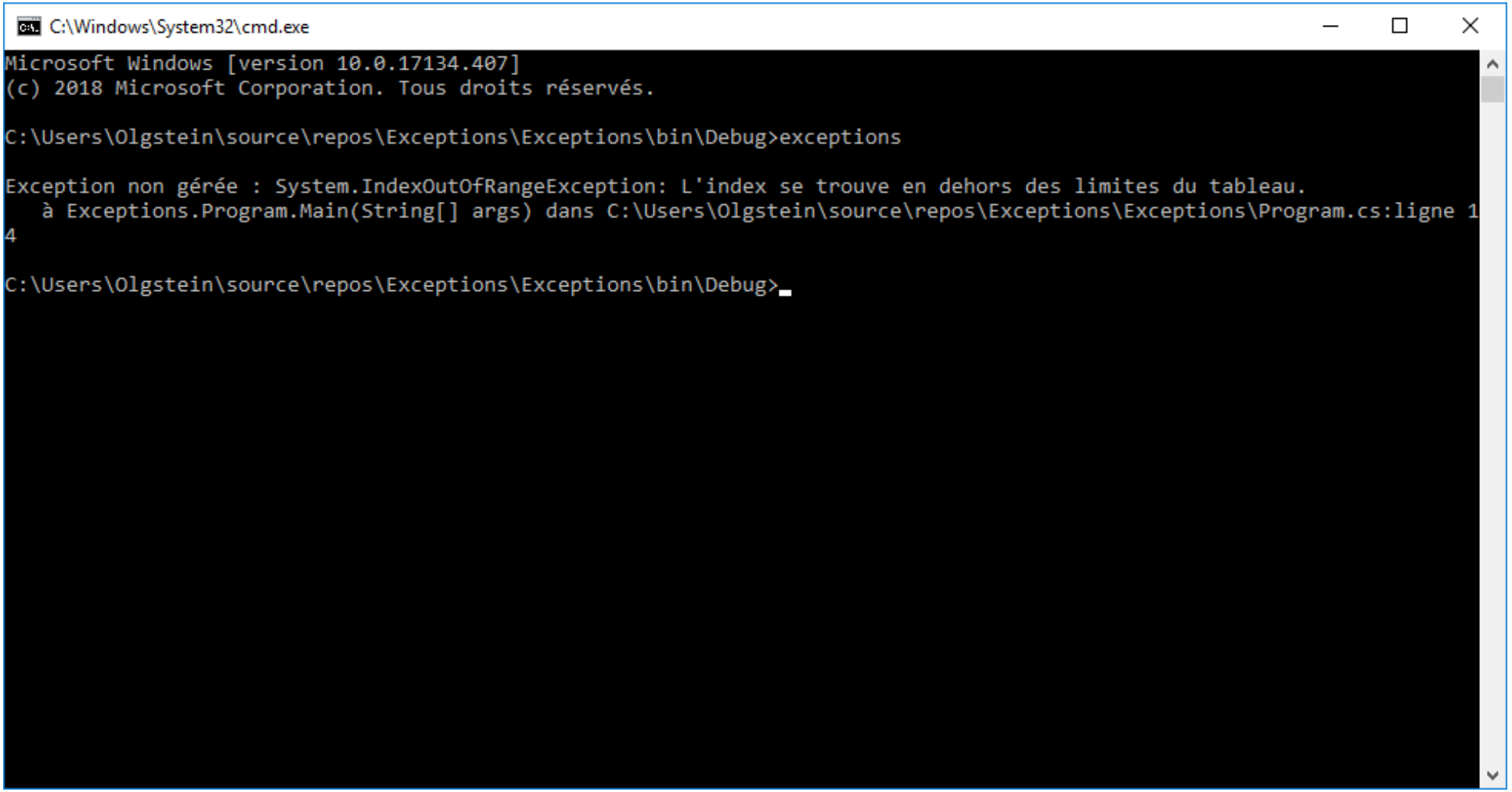
```
class Program
{
    static void Main(string[] args)
    {
        var array = new int[] { 1, 2, 3 };
        Console.WriteLine(array[11]);
        Console.ReadLine();
    }
}
```

Exceptions

```
class Program
{
    static void Main(string[] args)
    {
        var array = new int[] { 1, 2, 3 };
        Console.WriteLine(array[11]);
        Console.ReadLine();
    }
}
```



Exceptions


C:\Windows\System32\cmd.exe
Microsoft Windows [version 10.0.17134.407]
(c) 2018 Microsoft Corporation. Tous droits réservés.
C:\Users\Olgstein\source\repos\Exceptions\Exceptions\bin\Debug>exceptions
Exception non gérée : System.IndexOutOfRangeException: L'index se trouve en dehors des limites du tableau.
à Exceptions.Program.Main(String[] args) dans C:\Users\Olgstein\source\repos\Exceptions\Exceptions\Program.cs:ligne 14
C:\Users\Olgstein\source\repos\Exceptions\Exceptions\bin\Debug>

Exceptions

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            var array = new int[] { 1, 2, 3 };
            Console.WriteLine(array[11]);
        }
        catch(IndexOutOfRangeException ex)
        {
            Console.WriteLine("oups!");
            Console.WriteLine($"Exception : {ex.Message}");
        }

        Console.ReadLine();
    }
}
```

Exceptions

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            var array = new int[] { 1, 2, 3 };
            Console.WriteLine(array[11]);
        }
        catch(IndexOutOfRangeException ex)
        {
            Console.WriteLine("oups!");
            Console.WriteLine($"Exception : {ex.Message}");
        }
        catch(KeyNotFoundException ex)
        {
            // ..
        }
        catch
        {
            // ...
        }
        finally
        {
            Console.WriteLine("Toujours exécuté!");
        }

        Console.ReadLine();
    }
}
```

Exceptions

```
try
{
    throw new Exception("Une erreur s'est produite!");
}
catch(Exception ex)
{
    // ...

    throw ex;
}
```

Exceptions

```
try
{
    throw new Exception("Une erreur s'est produite!");
}
catch(Exception ex)
{
    // ...

    throw;
}
```

Exercice

- Créer deux classes d’exceptions personnalisées
  - Pour une combinaison saisie de longueur incorrecte
  - Pour une combinaison comportant des caractères incorrects
- Lancer ces exceptions lorsque nécessaire
- Ajouter les blocs try catch pour le traitement de ces exceptions

Événements

```
static void Main(string[] args)
{
    MonDelegue d = MaMethode;
    d(5);
}

delegate void MonDelegue(int p1);

public static void MaMethode(int i)
{
    // ...
}
```

Événements

- Des raccourcis :
  - Action
  - Action<T>
  - Action<T1, T2>
  - Func<T>
  - Func<T,T1>
  - Func<T,T1,T2>

Événements

```
public delegate void MonDelegue(int p1);

public class ClassWithEvent
{
    public event MonDelegue Evt1;

    public void DoSomething()
    {
        // ...

        if(Evt1 != null)
        {
            Evt1(5);
        }

        Evt1?.Invoke(5);

        // ...
    }
}
```

```
var c = new ClassWithEvent();
c.Evt1 += MaMethode;

c.DoSomething();

c.Evt1 -= MaMethode;
```

Événements

Méthode recommandée

```
public class MonEventArgs : EventArgs
{
    // ...
}

public class ClassWithEvent
{
    public event EventHandler<MonEventArgs> Evt1;

    protected virtual void OnEvt1()
    {
        var evt = Evt1;
        var arg = new MonEventArgs();
        // ...
        evt?.Invoke(this, arg);
    }
}
```

Exercice

- Externaliser les appels à Console.WriteLine et Console.ReadLine via des événements de la classe du jeu Mastermind