

Génie Logiciel

Rappels et compléments C#

BAPTISTE PESQUET

Support de cours

<https://ensc.gitbook.io/programmation-objet-csharp/>

Sommaire

- Rappels sur la POO
- Gestion des objets en mémoire
- Gestion des exceptions

3 sur 29

Rappels sur la POO

La POO en bref

Objet : entité qui modélise (représente) un élément du domaine étudié

Objet = état + actions

Objet <> **classe**

Classe : modèle d'objet (type)

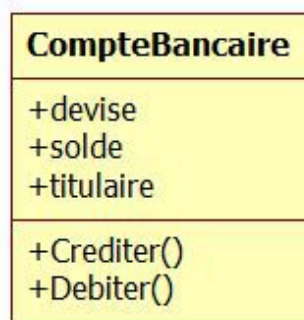
Objet : exemplaire concret, **instance** d'une classe

POO : écriture de logiciels sous forme d'objets en interaction

5 sur 29

Représentation graphique d'une classe

Standard : diagramme de classe **UML**



6 sur 29

Notion de propriété en C#

Propriété au sens de la majorité des langages à objets = **attribut** ou **champ**

Propriété au sens du C# = **accesseur** vers un attribut

Les propriétés permettent de concilier accès aux données de l'objet et **encapsulation**

7 sur 29

Propriété : syntaxe 1

Déclaration explicite de l'attribut (le plus souvent privé)

```
private string titulaire;  
  
public string Titulaire {  
    get { return titulaire; }  
    set { titulaire = value; }  
}  
  
monCompte.Titulaire = "Marco";
```

Avantage : possibilité de contrôles dans les accesseurs (test de valeur, etc)

8 sur 29

Propriété : syntaxe 2

Création implicite de l'attribut par le langage (« propriété automatique »)

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

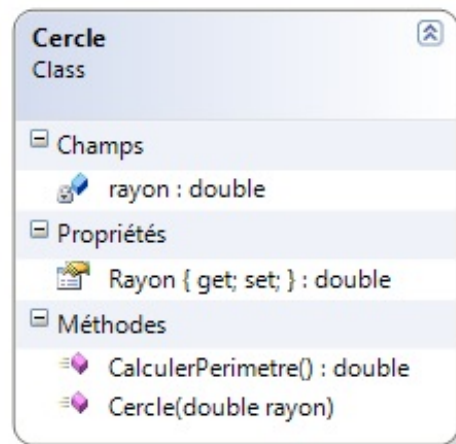
```
monCompte.Titulaire = "Marco";
```

Avantage : concision, bien adapté aux cas simples

9 sur 29

Gestion des objets en mémoire

Classe d'exemple



11 sur 29

Affectation entre entiers

```
int nombre1;  
nombre1 = 5;  
int nombre2 = 3;  
  
nombre2 = nombre1;  
nombre1 = 10;  
  
Console.WriteLine("nombre1 = " + nombre1); // 10  
Console.WriteLine("nombre2 = " + nombre2); // 5
```

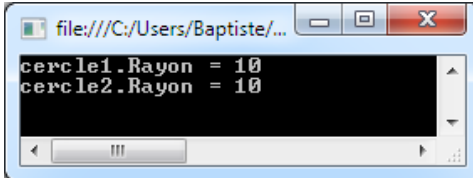
12 sur 29

Affectation entre objets

```
Cercle cercle1;
cercle1 = new Cercle(5);
Cercle cercle2 = new Cercle(3);
```

```
cercle2 = cercle1;
cercle1.Rayon = 10;
```

```
Console.WriteLine("cercle1.Rayon = "
    + cercle1.Rayon); // 10
Console.WriteLine("cercle2.Rayon = "
    + cercle2.Rayon); // ???
```



```
file:///C:/Users/Baptiste/...
cercle1.Rayon = 10
cercle2.Rayon = 10
```



13 sur 29

Types valeur

Valeur directement stockée dans la variable

Types de bases C# : int, float, double...

Création d'une variable => réservation d'une zone mémoire pour stocker sa valeur

```
int nombre1;
nombre1 = 5;
```

```
int nombre2 = 3;
```

```
nombre2 = nombre1;
```

```
nombre1 = 10;
```

nombre1

5

nombre1

5

nombre1

5

nombre1

10

nombre2

3

nombre2

5

nombre2

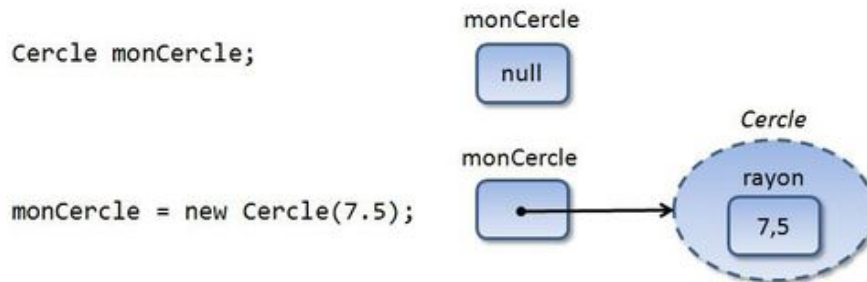
5

14 sur 29

Types référence

Objets et tableaux en C#

Instanciation d'un objet => réservation d'une zone mémoire pour ses données, mais la « valeur » de l'objet est une **référence** vers cette zone

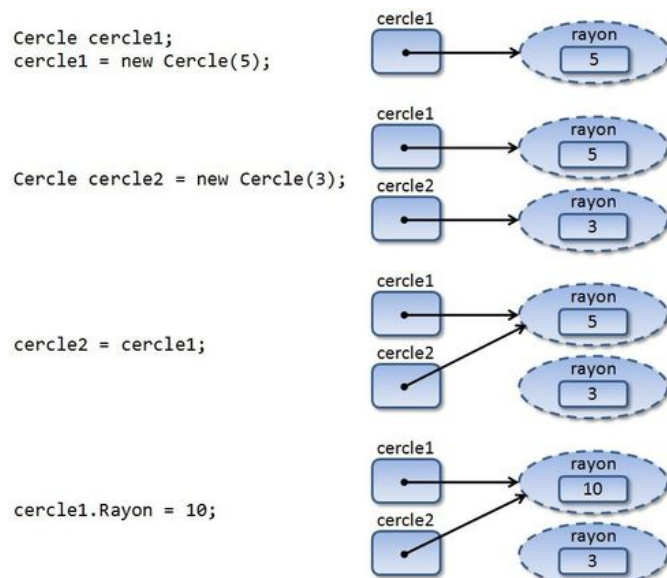


15 sur 29

Types référence et affectation

La valeur de la variable source (une référence) est copiée dans la variable cible

Les deux variables pointent vers la même zone mémoire



16 sur 29

Passage d'un type valeur en paramètre

```
static void Main(string[] args) {
    int nombre = 5;
    Console.WriteLine("Avant l'appel, nombre = " + nombre);
    Augmenter(nombre);
    Console.WriteLine("Après l'appel, nombre = " + nombre);
}

static void Augmenter(int unNombre) {
    Console.WriteLine("Avant l'augmentation, unNombre = " + unNombre);
    unNombre = unNombre + 1;
    Console.WriteLine("Après l'augmentation, unNombre = " + unNombre);
}
```

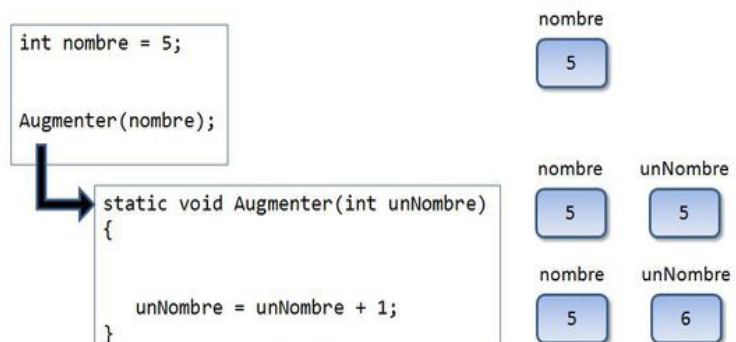
17 sur 29

Résultat de l'exécution

La valeur de l'argument
(un nombre entier) est
copiée dans le
paramètre

Argument et
paramètre
correspondent à des
zones mémoire
différentes

```
file:///C:/Users/Baptiste/Dropbox/Enseignement...
Avant l'appel, nombre = 5
Avant l'augmentation, unNombre = 5
Après l'augmentation, unNombre = 6
Après l'appel, nombre = 5
```



18 sur 29

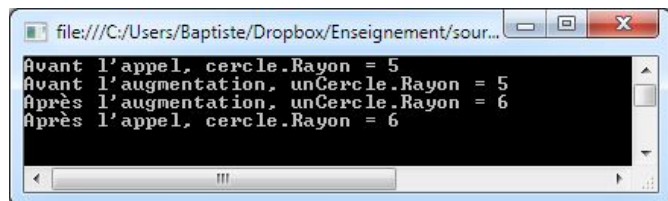
Passage d'un objet en paramètre

```
static void Main(string[] args) {
    Cercle cercle = new Cercle(5);
    Console.WriteLine("Avant l'appel, cercle.Rayon = " + cercle.Rayon);
    AugmenterRayon(cercle);
    Console.WriteLine("Après l'appel, cercle.Rayon = " + cercle.Rayon);
}

static void AugmenterRayon(Cercle unCercle) {
    Console.WriteLine("Avant l'augmentation, unCercle.Rayon = " + unCercle.Rayon);
    unCercle.Rayon = unCercle.Rayon + 1;
    Console.WriteLine("Après l'augmentation, unCercle.Rayon = " + unCercle.Rayon);
}
```

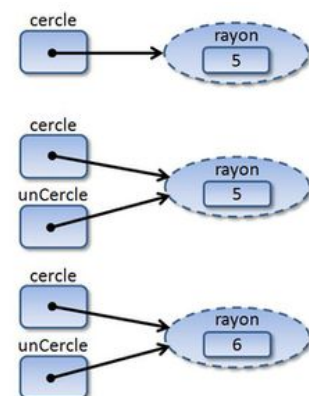
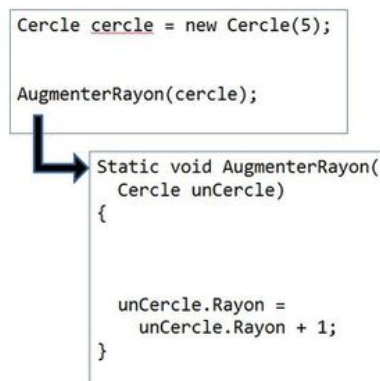
19 sur 29

Résultat de l'exécution



La valeur de l'argument
(une **référence**) est
copiée dans le
paramètre

Argument et
paramètre pointent
vers la même zone
mémoire



20 sur 29

Mode de passage des paramètres en C#

Par défaut, **tous les paramètres sont passés par valeur**

Types valeur : valeur copiée de l'argument vers le paramètre => zones mémoire distinctes

Types référence : référence copiée de l'argument vers le paramètre
=> même zone mémoire

Possibilité de modifier ce comportement avec ref et out

21 sur 29

Gestion des exceptions

Introduction

Exception : évènement qui apparaît pendant le déroulement d'un programme et qui empêche la poursuite normale de son exécution

- BD inaccessible, fichier non trouvé, bug interne...

Gestion des exceptions : technique de gestion des erreurs dans un programme

Avantage : séparation du code applicatif du code de gestion des erreurs

23 sur 29

Syntaxe

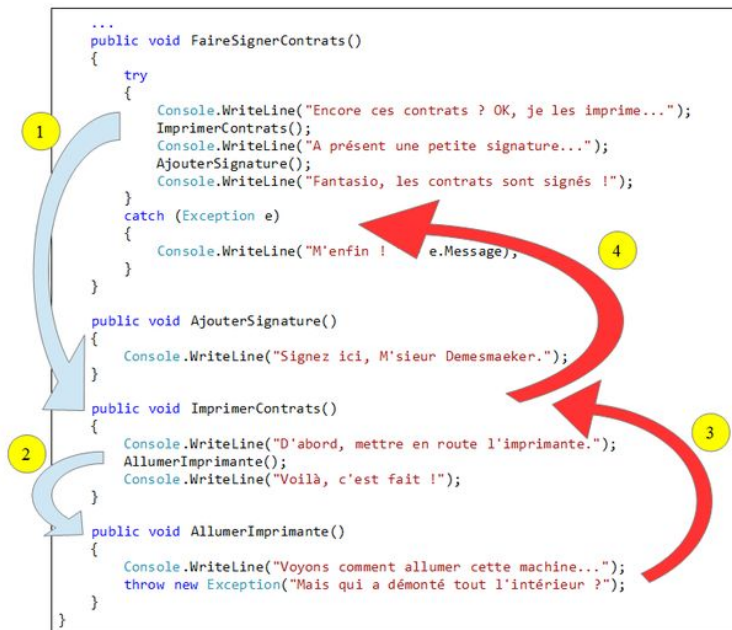
```
try {  
    // code susceptible de lever des exceptions  
}  
catch (Exception e) {  
    // code de gestion de l'exception apparue  
}  
finally {  
    // code exécuté systématiquement  
}  
  
// levée d'une nouvelle exception  
throw new Exception("Message d'erreur");
```

24 sur 29

Exceptions et chaîne des appels

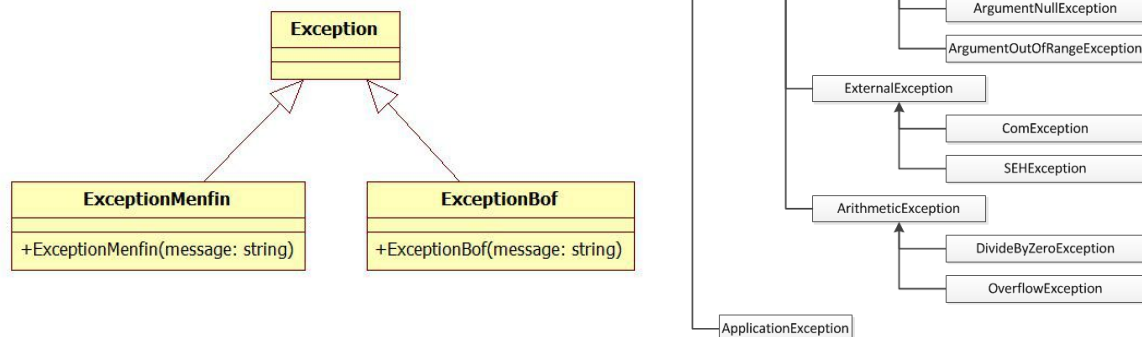
Une exception levée remonte la chaîne des appels dans l'ordre inverse

Fin : interception dans un bloc **catch** ou arrêt brutal du programme



25 sur 29

La hiérarchie des exceptions



26 sur 29

Exceptions et bonnes pratiques (1/3)

Lever une exception uniquement pour signaler qu'une exécution normale n'est plus possible

- Ne pas utiliser les exceptions pour des situations non exceptionnelles (succès d'une recherche, fin d'une itération, etc)

```
i = 0;
trouve = false;
while (!trouve) {
    i++;
    if (i == 10)
        throw new Exception("Fin de la boucle"); // Antipattern
    else // ...
}
```

27 sur 29

Exceptions et bonnes pratiques (2/3)

Intercepter les exceptions uniquement si un traitement approprié est possible (message d'erreur, nouvelle tentative, etc)

- Sinon, la laisser remonter la chaîne des appels vers un meilleur endroit

<pre>try { // ... } catch (Exception e) { // Antipattern : bloc inutile throw e; }</pre>	<pre>try { // ... } catch (Exception) { // Antipattern : exception « avalée » }</pre>
--	---

28 sur 29

Exceptions et bonnes pratiques (3/3)

Bien réfléchir avant de créer ses propres classes d'exception

- Utile pour véhiculer des données spécifiques dans les exceptions
- Toujours inclure le mot *Exception* dans le nom de la classe
- En deçà d'une certaine complexité, utiliser la classe standard `Exception` suffit souvent
- Code de gestion des erreurs << code applicatif