

Chapitre 4

Coder des classes simples

1 Introduction

Dans les précédents chapitres, on a découvert les bases du langage C#. On y a adopté la posture d'un programmeur PUO, qui **utilise** des usines prédéfinies et leurs objets.

Ce chapitre propose de franchir une étape supplémentaire : découvrir la manière de coder de nouvelles **classes**.

Dans ce but, nous allons toujours utiliser les classes `Ticket` et `MachineATickets`, en présentant la manière de les coder.

2 Code des classes `MachineATickets` et `Ticket`

Voici le code de la classe `MachineATickets`.

```
class MachineATickets
{
    //- Les VARIABLES D'INSTANCES -----
    private int prixTicket;
    private int totalCaisse;
    private int soldeClient;

    //- Les CONSTRUCTEURS -----
    public MachineATickets()
        : this(1000)
    {
    }
    public MachineATickets(int prixTicket)
    {
        //- Traitement -
        this.SetPrix(prixTicket);
        this.totalCaisse = 0;
        this.soldeClient = 0;
    }

    //- Les METHODES -----
    public void AccepteArgent(int montant)
    {
        //- Solution 1 -----
        int[] montantReconnus = new int[]
        { 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 };

        bool montantOk = false;

        for (int cptr = 0; cptr < montantReconnus.Length &&
            montantOk == false; cptr++)
            if (montantReconnus[cptr] == montant)
                montantOk = true;
    }
}
```

```
        if (montantOk == false)
            throw new Exception("Pièce non reconnue ou billet non reconnu");
        //- Fin solution 1-----

        //OU BIEN

        //- Solution 2 -----
        //if (montantReconnus.Contains(montant) == false)
        //    throw new Exception(
        //        "Pièce non reconnue ou billet non reconnu");
        //- Fin solution 2 -----
        this.soldeClient += montant;
    }

    public Ticket DonneTicket()
    {
        //- Déclarations et initialisations des v. LOCALES -
        Ticket retVal = null;

        //- Traitement -
        try
        {
            retVal = this.ConstruireTicketEtActualiserMachine();
        }
        catch(Exception ex)
        {
        }

        return retVal;
    }

    public ArrayList DonneTicket(int nombreDeTickets)
    {
        //- Déclarations et initialisations des v. LOCALES -
        ArrayList retVal = null;
        Ticket unTicket = null;
        int cptTickets;

        //- Traitement -

        if (nombreDeTickets <= 0)
            throw new Exception
                ("Nombre de tickets négatif interdit");

        try
        {
            for (cptTickets = 0; cptTickets < nombreDeTickets; cptTickets++)
            {
                unTicket = this.ConstruireTicketEtActualiserMachine();
                if (cptTickets == 0)
                    retVal = new ArrayList();
                retVal.Add(unTicket);
            }
        }
        catch {}

        return retVal;
    }

    private Ticket ConstruireTicketEtActualiserMachine()
    {
        if (this.soldeClient < this.prixTicket)
            throw new Exception("Solde insuffisant.");

        this.totalCaisse += this.prixTicket;
        this.soldeClient -= this.prixTicket;
    }
}
```

```
        return new Ticket(this.prixTicket);
    }

    public int RendReste()
    {
        //- Traitement -
        int retVal = this.soldeClient;
        this.soldeClient = 0;
        return retVal;
    }

    public int GetPrix()
    {
        //- Traitement -
        return this.prixTicket;
    }

    public void SetPrix(int prixTicket)
    {
        //- Traitement -
        if (prixTicket <= 0)
            throw new Exception("Prix négatif interdit");

        this.prixTicket = prixTicket;
    }

    public string GetChaîne()
    {
        //- Traitement -
        return
            "Prix des tickets : " + (float)this.prixTicket / 100
            + " Euros\nMontants\n"
            + "- Caisse : " + this.totalCaisse / 100.0 + "Euros\n"
            + "- Solde client : " + this.soldeClient / 100.0 + "Euros\n";
    }
}
```

Voici le code de la classe Ticket.

```
class Ticket
{
    //- Les VARIABLES D'INSTANCES -----
    private int prix;

    //- Le CONSTRUCTEUR -----
    public Ticket(int prix)
    {
        if (prix >= 0)
            this.prix = prix;
        else
            throw new Exception("Prix négatif interdit");
    }

    //- Les METHODES -----
    public float GetPrixEnEuros()
    {
        //- Traitement -
        return (float)this.prix / 100;
    }

    public String GetChaîne()
    {
        //- Traitement -
        return "Prix : " + (double)prix / 100
            + " Euros\nBon voyage.\n";
    }
}
```

Et finalement, voici le code de test et l'affichage obtenu lors de son exécution.

```
static void Main(string[] args)
{
    // Déclarations -----
    MachineATickets maVerviers      = null,
                      maSeraing     = null,
                      maGothamCity  = null;
    Ticket premierTicket            = null;
    ArrayList liasseDeTickets       = null;
    int resteClient                 = 0;

    // Traitement -----

    Console.WriteLine("MODE DE FONCTIONNEMENT NOMINAL.\n" +
                      "===== \n");
    Console.WriteLine("CREER DEUX MACHINES.\n");

    maVerviers = new MachineATickets();
    maSeraing  = new MachineATickets(250);

    Console.WriteLine("ETAT DE LA MACHINE DE VERVIERS :\n\n" +
                      maVerviers.GetChaîne() +
                      "ETAT DE LA MACHINE DE SERAING :\n\n" +
                      maSeraing.GetChaîne());

    Console.WriteLine("METTRE 11 EUROS A SERAING.\n");

    maSeraing.AccepteArgent(1000);
    maSeraing.AccepteArgent(100);

    Console.WriteLine("ETAT DE LA MACHINE DE SERAING :\n\n" +
                      maSeraing.GetChaîne());

    Console.WriteLine("DEMANDER 1 TICKET A SERAING.\n");

    premierTicket = maSeraing.DonneTicket();

    if(premierTicket != null) //Toujours vrai ici
        Console.WriteLine("ETAT DE LA MACHINE DE SERAING :\n\n" +
                          maSeraing.GetChaîne() +
                          "ETAT DU TICKET OBTENU :\n\n" +
                          premierTicket.GetChaîne());
    else
        Console.WriteLine("PAS DE TICKET OBTENU, SOLDE INSUFFISANT.");

    Console.WriteLine("DEMANDER 10 TICKETS A SERAING.\n");

    liasseDeTickets = maSeraing.DonneTicket(10);

    if (liasseDeTickets != null) //Toujours vrai ici
    {
        Console.WriteLine("ETAT DE LA MACHINE DE SERAING:\n\n" +
                          maSeraing.GetChaîne() +
                          "LIASSE DE " +
                          liasseDeTickets.Count +
                          " TICKETS OBTENUE.\nSON CONTENU EST :\n");

        foreach (Ticket ticket in liasseDeTickets)
            Console.WriteLine(ticket.GetChaîne());
    }
    else
        Console.WriteLine("PAS DE LIASSE OBTENUE, SOLDE INSUFFISANT.\n\n");

    Console.WriteLine("PRENDRE SON RESTE.\n");
}
```

```
resteClient = maSeraing.RendReste();

if(resteClient > 0) //Toujours vrai ici
    Console.WriteLine("LE CLIENT RECUPERE " +
        resteClient / 100.0 +
        " EUROS.\n"+"ETAT DE LA MACHINE DE SERAING :\n\n" +
        maSeraing.GetChaîne());
else
    Console.WriteLine("SOLDE CLIENT NUL. " +
        "IL N'Y A PAS DE RESTE A RECUPERER.\n");

Console.WriteLine("POUR ETRE COMPLET." +
    "CHANGER PRIX A VERVIERS de 10 A 6,50 EUROS.\n");
maVerviers.SetPrix(650);

Console.WriteLine("ETAT DE LA MACHINE DE VERVIERS :\n\n" +
    maVerviers.GetChaîne());

Console.WriteLine("PRIX VERVIERS DE " +
    maVerviers.GetPrix() / 100.0 + " EUROS.\n");

Console.WriteLine("MODE DE FONCTIONNEMENT NON NOMINAL.\n" +
    "=====\n\n" +
    "ESSAYER DE CREER MACHINE DEBILE.\n");

try
{
    maGothamCity = new MachineATickets(-69); //Levée d'exception,
                                           //exécution du try
                                           //interrompue
    Console.WriteLine("MACHINE DEBILE CREEE.\n"); //NON exécuté
}
catch(Exception ex)
{
    Console.WriteLine("EXCEPTION INTERCEPTEE, MESSAGE EMBARQUE : " +
        ex.Message + ".\n" +
        "MACHINE NON CREEE, maGothamCity VAUT " +
        ((maGothamCity == null)? "null":
            maGothamCity.ToString())+
        ".\n");
}

Console.WriteLine("ESSAYER D'ENFOURNER UNE PASTILLE " +
    "POUR LA GORGE A VERVIERS.\n");

try
{
    maVerviers.AccepteArgent(123); //Levée d'exception,
    //exécution du try interrompue
    Console.WriteLine("PASTILLE POUR LA GORGE AVALEE.\n"); //NON exécuté
}
catch (Exception ex)
{
    Console.WriteLine("OPERATION REFUSEE, EXCEPTION INTERCEPTEE, " +
        "MESSAGE : " + ex.Message + ".\n\n" +
        "ETAT DE LA MACHINE DE VERVIERS :\n\n" +
        maSeraing.GetChaîne());
}

Console.WriteLine("ESSAYER D'OBTENIR UNE LIASSE DEBILE.\n");

try
{
    liasseDeTickets = null; //DO NOT FORGET,
    //l'ancienne liasse existe toujours
    maVerviers.AccepteArgent(2000);
}
```

```
        liasseDeTickets = maVerviers.DonneTicket(-2); //Levée d'exception,
                                                    //exécution du try
                                                    //interrompue
        Console.WriteLine("LIASSE OBTENUE.\n"); //NON exécuté
    }
    catch (Exception ex)
    {
        Console.WriteLine("EXCEPTION INTERCEPTEE, MESSAGE EMBARQUE : " +
            ex.Message + ".\n" +
            "LIASSE DE TICKETS ET TICKETS NON CREES. " +
            "LA VARIABLE liasseTickets VAUT " +
            ((liasseDeTickets == null) ?
                "null" :
                liasseDeTickets.ToString()) +
            ".\n"); ;
    }
    Console.ReadLine();
}
```

Voici l'affichage obtenu lors d'une exécution de cette méthode.

```
MODE DE FONCTIONNEMENT NOMINAL.
=====

CREER DEUX MACHINES.

ETAT DE LA MACHINE DE Verviers :

Prix des tickets : 10 Euros
Montants
- Caisse      : 0 Euros
- Solde client : 0 Euros
ETAT DE LA MACHINE DE Seraing :

Prix des tickets : 2,5 Euros
Montants
- Caisse      : 0 Euros
- Solde client : 0 Euros

METTRE 11 EUROS A SERAING.

ETAT DE LA MACHINE DE SERAING :

Prix des tickets : 2,5 Euros
Montants
- Caisse      : 0 Euros
- Solde client : 0 Euros

DEMANDER 1 TICKET A SERAING.

PAS DE TICKET OBTENU, SOLDE INSUFFISANT.
DEMANDER 10 TICKETS A SERAING.

PAS DE LIASSE OBTENUE, SOLDE INSUFFISANT.

PRENDRE SON RESTE.

SOLDE CLIENT NUL. IL N'Y A PAS DE RESTE A RECUPERER.

POUR ETRE COMPLET.
CHANGER PRIX A Verviers de 10 A 6,50 EUROS.

ETAT DE LA MACHINE DE Verviers :

Prix des tickets : 6,5 Euros
Montants
- Caisse      : 0 Euros
- Solde client : 0 Euros

PRIX Verviers DE 6,5 EUROS.

MODE DE FONCTIONNEMENT NON NOMINAL.
=====

ESSAYER DE CREER MACHINE DEBILE.

EXCEPTION INTERCEPTEE, MESSAGE EMBARQUE : Prix négatif interdit.
MACHINE NON CREEE, LA VARIABLE maGothamCity VAUT null.

ESSAYER D'ENFOURNER UNE PASTILLE POUR LA GORGE A Verviers.

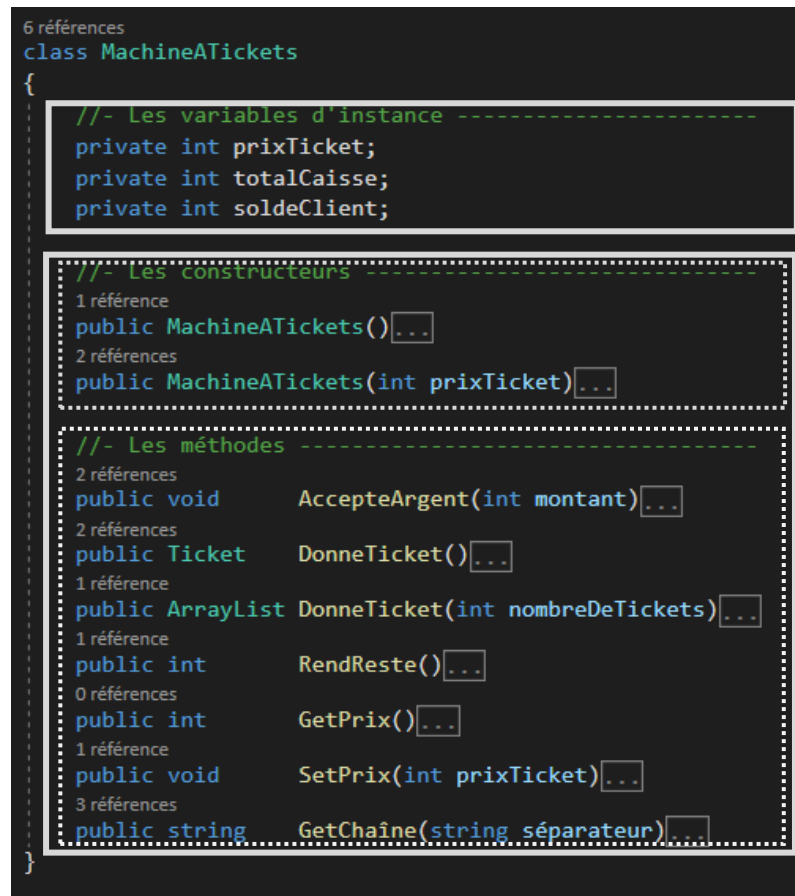
OPERATION REFUSEE, EXCEPTION INTERCEPTEE, MESSAGE : Pièce non reconnue ou billet non reconnu.
```

3 Organisation générale du code d'une classe

Le code de chaque classe est écrit dans son propre fichier.

La définition d'une classe débute par une ligne d'en-tête, qui utilise le mot clé `class` suivi de l'identificateur choisi pour la classe. Le bloc qui suit définit **la structure commune ET le comportement commun** de tous les **FUTURS objets** qui **seront** construits à partir de cette classe.

On recommande est de toujours organiser le code comme suit.



```
6 références
class MachineATickets
{
    //- Les variables d'instance -----
    private int prixTicket;
    private int totalCaisse;
    private int soldeClient;

    //- Les constructeurs -----
    1 référence
    public MachineATickets()...
    2 références
    public MachineATickets(int prixTicket)...

    //- Les méthodes -----
    2 références
    public void AccepteArgent(int montant)...
    2 références
    public Ticket DonneTicket()...
    1 référence
    public ArrayList DonneTicket(int nombreDeTickets)...
    1 référence
    public int RendReste()...
    0 références
    public int GetPrix()...
    1 référence
    public void SetPrix(int prixTicket)...
    3 références
    public string GetChaine(string séparateur)...
}
```

Le rectangle **supérieur** contient les **déclarations des variables d'instances**. Toute future `MachineATickets` contiendra trois sous-objets qui correspondent à ces variables d'instances. Ici, ce sont trois `Int32`.

Le rectangle **inférieur** contient les **définitions des constructeurs et des méthodes**. Ce sont les portions de code qui s'exécuteront lorsqu'un message sera envoyé soit à la classe (l'usine), soit à une de instances.

- Les **constructeurs** contiennent le code exécuté quand **la classe** est sollicitée pour construire un nouvel objet.
- Les **méthodes** contiennent le code exécuté quand **un objet est** sollicité pour rendre un des services dont il est capable.

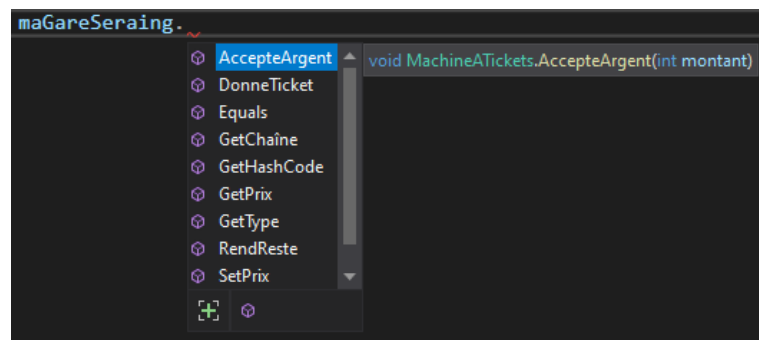
Le code de classe `Ticket` est organisé selon le même schéma.

4 Principe d'encapsulation

4.1 Introduction

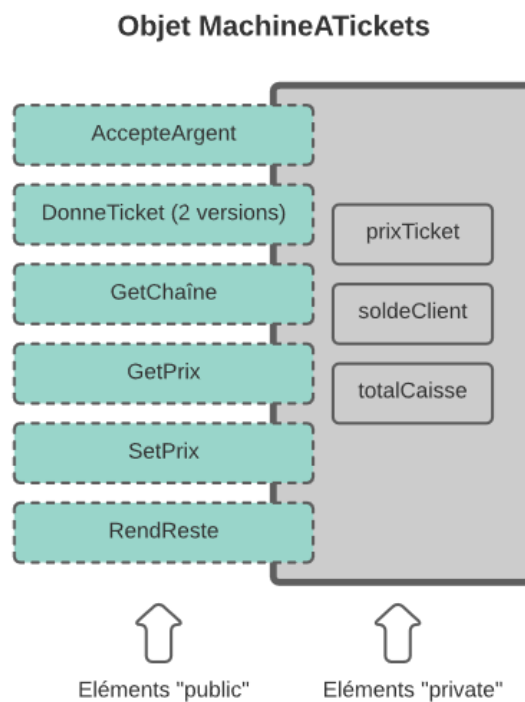
La déclaration de chaque variable d'instance et l'en-tête de chaque méthodes est préfixé de `public` ou `private`.

Pour en comprendre la signification, plaçons-nous à l'extérieur de la classe, par exemple dans `Main`. En appliquant la notation pointée à une variable `MachineATickets`, les services proposés par l'objet sont affichés. Il s'agit de l'**interface publique** de l'objet. **Ces éléments sont ceux qui sont préfixés du mot clé `public` dans le code.**



Les éléments `private` sont bien entendu présents, mais **inaccessibles** depuis l'extérieur. C'est ainsi qu'est mis en œuvre le **principe d'encapsulation**.

Depuis l'extérieur, un objet est donc une « capsule hermétique ». On ne peut interagir avec lui qu'au travers de son interface publique, le reste de sa mécanique étant cachée.



4.2 Bénéfices de l'encapsulation

4.2.1 L'objet préserve la cohérence de son état

Il ne devrait jamais être possible de créer une voiture de 300 tonnes ou un mannequin à trois cuisses de QI négatif. C'est la raison pour laquelle une classe doit garantir que tout objet construit est « cohérent » et le restera.

Pratiquement, cela s'obtient en respectant le principe d'encapsulation : on ne peut interagir avec une classe et ses objets qu'au travers des constructeurs et méthodes de l'**interface publique**. Le reste doit être privé.

Briser le principe d'encapsulation, par exemple en rendant publique `totalCaisse`, c'est s'exposer à ce qu'un code client puisse contenir une ligne stupide comme :

```
| maGareLiège.totalCaisse = -1442;
```

4.2.2 Couplage entre l'objet et le code utilisateur

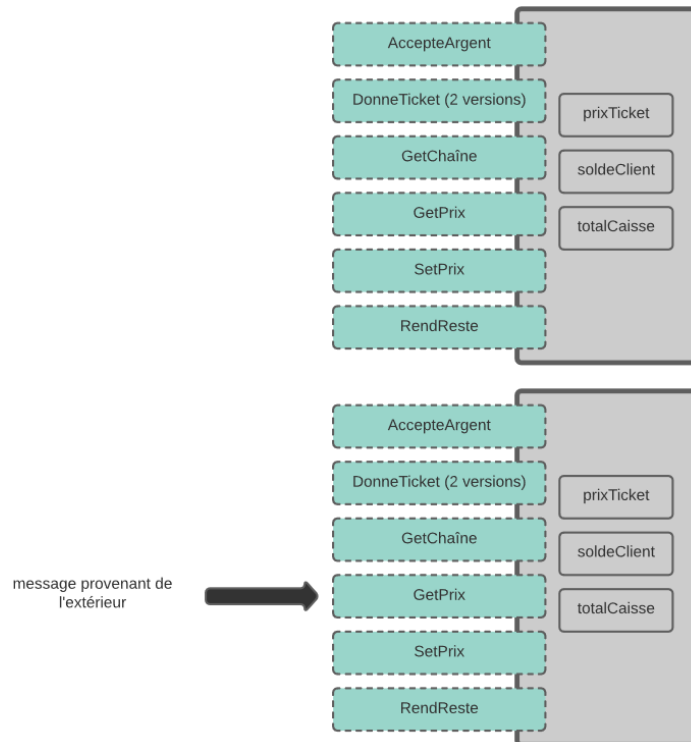
L'encapsulation **diminue le couplage entre les objets et le code utilisateur**.

Prenons l'exemple d'une classe `Sinistre` utilisée par les applications d'une compagnie d'assurances. Au fil du temps, cette classe devra évoluer pour s'adapter aux nouvelles contraintes légales, organisationnelles, commerciales, etc. Et généralement, un programmeur craint de modifier une pièce logicielle existante qui est déjà utilisée. La moindre erreur peut faire vaciller ou même s'écrouler la chaîne applicative existante.

Si le principe d'encapsulation est respecté, cette chaîne applicative peut être vue comme un code PUO qui n'utilise un objet qu'au travers de son interface publique. Le risque existe toujours mais il est jugulé. Le programmeur qui modifie la classe doit seulement veiller à préserver sa façade publique. La tuyauterie interne peut pour son part subir des modifications parfois profondes, sans craindre d'effet de bord inattendu avec les codes utilisateurs.

5 L'objet courant

Supposons que deux objets `MachineATicket` existent. On envoie le message `GetPrix` à la **seconde**. Cet objet qui est la cible de l'envoi s'appelle l'**objet courant**. C'est donc pour lui que le code de la méthode `GetPrix` doit s'exécuter.



Du point de vue du programmeur qui code une méthode, il faut parfois pouvoir désigner l'objet courant dans sa globalité. Pour cela, il dispose du mot clé `this`. C'est **une référence vers l'objet courant**.

ATTENTION. `this` ne peut être utilisé **que DANS** une méthode de classe.

- Dans un constructeur, il désigne **l'objet en cours de construction**.
- Dans une autre méthode, il désigne **l'objet « au sein duquel »** la méthode s'exécute. C'est l'objet qui vient de recevoir le message déclencheur.

ATTENTION. Dans une méthode de classe, **le principe d'encapsulation s'effondre**. Le programmeur peut sans **aucune restriction** accéder à **tous** les éléments (`private` et `public`) de l'objet courant. Il pourra par exemple écrire :

```
| this.soldeClient += 100 ;
```

6 Analyse de MachineATickets

6.1 Méthode AccepteArgent

6.1.1 Généralités

La méthode `AccepteArgent` s'exécute lorsqu'un objet reçoit le message correspondant. Il est accompagné d'un paramètre qui précise le montant à ajouter au solde du client.

Voici une manière de coder cette méthode.

```
| public void AccepteArgent(int montant)
| {
|     //- Déclarations et initialisations des v. LOCALES -
|     int[] montantReconnus = new int[]
|         { 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 };
| }
```

```
bool montantOk = false;

//- Traitement -
for (int cptr = 0; cptr < montantReconnus.Length &&
    montantOk == false; cptr++)
    if (montantReconnus[cptr] == montant)
        montantOk = true;

if (montantOk == false)
    throw new Exception("Pièce non reconnue ou billet non reconnu");

this.soldeClient += montant;
}
```

6.1.2 Utiliser un tableau à un indice

Le montant inséré dans la machine (paramètre `montant`) provient « de l'extérieur ». Il est donc prudent de vérifier qu'il correspond à la valeur d'une pièce ou d'un billet qui existe. Pour cela, on peut utiliser un **tableau à un indice** qui contient les valeurs (en centimes d'€) des pièces et billets acceptables. Pour le déclarer et l'initialiser, on peut écrire :

```
int[] montantsReconnus;
montantsReconnus = new int[]
{ 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 };
```

La déclaration/initialisation est possible en une seule instruction. On peut utiliser :

```
int[] montantsReconnus = new int[]
{ 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 };
```

ou bien :

```
int[] montantsReconnus =
{ 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 };
```

La variable `montantsReconnus` désigne un **objet** de la classe `Array`.

Après instanciation, **un objet tableau ne peut pas être redimensionné**. Mais on peut le remplacer par un autre. Par exemple, pour ne conserver que les billets, on réaffecte `montantsReconnus` :

```
montantsReconnus = new int[]
{ 50, 100, 200, 500, 1000, 2000, 5000 };
```

`montantsReconnus` est une variable locale, qui disparaîtra à la fin de l'exécution de la méthode. L'objet tableau perdra alors l'unique référence qui le désigne et sera atomisé par Samantha.

6.1.3 Compteur de boucle

Valider `montant` revient à vérifier que sa valeur est présente dans `montantsReconnus`. On utilise :

```
for (int cptr = 0; cptr < montantsReconnus.Length
    && montantOk == false; cptr++)
    if (montantsReconnus[cptr] == montant)
        montantOk = true;
```

La variable `cptr` est déclarée directement dans le `for`. Cela implique que sa portée est limitée **au corps de la boucle**. Par exemple, placer l'instruction :

```
Console.WriteLine("Élément détecté en position : " + (cptr - 1));
```

après la boucle produira une erreur de compilation.

Evidemment, on peut toujours déclarer `cptr` « traditionnellement ».

La condition de maintien utilise `montantReconnus.Length` qui donne la taille du tableau. `Length` n'utilise pas de parenthèses. C'est une fois encore une **propriété**.

6.1.4 Lever une exception

On a déjà vu comment intercepter une exception. Mais ici, on est passé de « l'autre côté ». Si le montant est non reconnu, il faut **lever une nouvelle exception**. On utilise l'instruction `throw`.

```
if(montantOk == false)
    throw new Exception("Pièce non reconnue ou billet non reconnu");
```

L'exception est une instance de la classe `Exception`. Le paramètre de son constructeur embarque dans l'objet un texte qui explique la nature de l'erreur. Il pourra être consulté dans le bloc `catch` du code client avec la propriété `Message` de l'objet.

ATTENTION. Quand on code une levée d'exception, on ne dispose généralement d'aucune information sur la nature exacte du client qui l'interceptera. Il peut s'agir d'un code batch, d'une application Windows, d'une application Web, etc.

L'exécution de `throw` **force l'interruption immédiate de l'exécution de la méthode en cours**. Dans l'exemple, si on entre dans le `if`, le code qui le suit ne sera **pas exécuté**.

6.1.5 `this` est-il obligatoire ?

Sauf si une exception est levée, `AccepteArgent` se termine par l'actualisation du solde **de l'objet courant**.

```
| this.soldeClient += montant;
```

Ici, la ligne pourrait aussi s'écrire sans utiliser `this`.

```
| soldeClient += montant;
```

Mais la seconde syntaxe est **confuse**, car les deux variables utilisées sont de **natures TOTALEMENT DIFFÉRENTES**.

- `montant` est une variable **locale**. Son existence est limitée au temps d'exécution de `AccepteArgent`.
- `soldeClient` est une variable **d'instance**. Elle a été créée avec l'objet courant et disparaîtra avec lui.

`montant` pourrait naître et mourir plusieurs fois pour un même objet, si le message `AccepteArgent` lui est envoyé plusieurs fois !

Pour cette raison, dans ce cours, on utilisera SYSTEMATIQUEMENT `this` pour accéder à un élément d'instance¹.

¹ Une alternative courante (qui vient de Python) est de débiter les identificateurs des variables d'instances (et seulement eux) par le caractère « `_` ». Ici, on utiliserait `_soldeClient`. On pourrait alors ne pas utiliser `this` sans pour autant créer d'ambiguïté dans le code. L'affectation ci-dessus s'écrirait :

```
| _soldeClient += montant;
```

6.1.6 Les services des collections prédéfinies

Le tableau est un premier exemple de **collection prédéfinie**. Il en existe d'autres. Et le bon sens le plus élémentaire suggère que tout objet de cette nature devrait être naturellement capable de « répondre » à certaines questions évidentes, par exemple « Contiens tu cette valeur ? ».

Dans le cas d'un tableau, on dispose de la méthode `Contains` qui retourne un booléen. Voici un exemple d'utilisation.

```
public void AccepteArgent(int montant)
{
    //- Déclarations / initialisations -
    int[] montantReconnus = new int[] { 1, 2, 5, 10, 20, 50, 100,
                                         200, 500, 1000, 2000, 5000 };

    //- Traitement -
    if (MontantReconnus.Contains(montant) == false)
        throw new Exception("Pièce non reconnue ou billet non reconnu");

    this.soldeClient += montant;
}
```

Ce code peut aussi s'écrire comme suit.

```
public void AccepteArgent(int montant)
{
    if ((new int[] { 1, 2, 5, 10, 20, 50, 100,
                    200, 500, 1000, 2000, 5000 }).Contains(montant) == false)
        throw new Exception("Pièce non reconnue ou billet non reconnu");

    this.soldeClient += montant;
}
```

6.2 Méthodes DonneTicket

6.2.1 Surcharge de méthode

En « C# », **deux méthodes peuvent avoir le même identificateur**. Rappelez-vous que dans les langages plus anciens comme « C », cette possibilité était exclue.

Mais il existe quand même une restriction : deux méthodes **doivent toujours posséder des signatures différentes**. La « **signature d'une méthode** » correspond à **son prototype amputé du type de retour**.

Ici, deux versions de `DonneTicket` de signatures `DonneTicket()` et `DonneTicket(int)` existent. On parle de **méthode surchargée**. Lors d'un appel, le compilateur identifie l'identité exacte de la méthode à exécuter à partir du **nombre** et **des types** des paramètres effectifs de l'appel.

```
// - La version SANS paramètre de DonneTicket est invoquée
Ticket t1 = maHallGareLiège.DonneTicket();

// - La version AVEC paramètre de DonneTicket est invoquée
Ticket t2 = maHallGareLiège.DonneTicket(4);
```

Après le premier appel, si le solde de `maHallGareLiège` est suffisant, `t1` pointe sur un objet `Ticket`. Sinon, `t1` vaut `null`.

Après le second appel , `t2` pointe sur un `ArrayList`. Il s'agit d'un nouveau type de collection prédéfinie. L'objet contient le nombre maximum des tickets demandés dans les limites du solde disponible. Sinon, `t2` vaut `null`.

6.2.2 Méthode de factorisation privée

Une machine doit toujours préserver la cohérence de son état. Lorsqu'elle délivre un ticket, elle doit toujours diminuer le solde du client **et** augmenter le total de sa caisse, ces deux actions étant indissociables.

Pour éviter de dupliquer le code correspondant dans les deux versions de `DonneTicket`, on peut le **factoriser** dans une méthode « logistique ». Cette dernière ne sera utilisée que de manière interne à l'objet sans faire partie de son interface publique. Elle sera donc `private`.

```
private Ticket ConstruireTicketEtActualiserMachine()
{
    if (this.soldeClient < this.prixTicket)
        throw new Exception("Solde insuffisant."); //- Interruption

    this.totalCaisse += this.prixTicket;
    this.soldeClient -= this.prixTicket;

    return new Ticket(this.prixTicket);
}
```

En cas de solde insuffisant, une exception levée. Sinon, le solde est diminué, le total de la caisse est augmenté, l'usine `Ticket` est invoquée pour obtenir un objet `Ticket` qui est ensuite retourné.

La levée d'exception est un choix de conception. Une alternative cohérente serait de retourner la valeur `null`.

6.2.3 Le code des méthodes `DonneTicket`

Voici le code de la première version de `DonneTicket`.

```
public Ticket DonneTicket()
{
    //- Déclarations et initialisations des v. LOCALES -
    Ticket retVal = null;

    //- Traitement -
    try
    {
        retVal = this.ConstruireTicketEtActualiserMachine();
    }
    catch{}

    return retVal;
}
```

Si `ConstruireTicketEtActualiserMachine` lève une exception, elle est interceptée et la valeur `null` est retournée vers le code client. Voici le code de la seconde version de `DonneTicket`.

```
public ArrayList DonneTicket(int nombreDeTickets)
{
    //- Déclarations et initialisations des v. LOCALES -
    ArrayList retVal = null;
    Ticket unTicket = null;
    int cptTickets;
```

```
//- Traitement -
if (nombreDeTickets <= 0)
    throw new Exception
        ("Nombre de tickets négatif interdit");

try
{
    for (cptTickets = 0; cptTickets < nombreDeTickets; cptTickets++)
    {
        unTicket = this.ConstruireTicketEtActualiserMachine();
        if (retVal == null)
            retVal = new ArrayList();
        retVal.Add(unTicket);
    }
}
catch{}

return retVal;
}
```

Si le nombre demandé est négatif, une exception est levée.

On trouve ensuite un bloc `try` qui contient une boucle, dont le code est un peu délicat. Plaçons-nous d'abord durant sa première itération.

- Dans le cas où le solde est insuffisant pour, `ConstruireTicketEtActualiserMachine` lève une exception. Le bloc `try` courant est donc interrompu et `retVal` garde sa valeur d'initialisation `null`.
- Sinon, un objet `ArrayList` est instancié et affecté à `retVal`. Un premier ticket y est ajouté.

Lors de chaque itération suivante, le système essaye d'instancier un ticket supplémentaire. Si le solde résiduel devient insuffisant, une exception est levée par `ConstruireTicketEtActualiserMachine` et la boucle est interrompue. Sinon, le ticket est ajouté dans `retVal`.

Donc, le retour final soit est `null`, soit est un `ArrayList` non vide qui contient le plus possible des tickets demandés en fonction du solde de départ.

Arrêtons-nous brièvement sur `ArrayList` qui est une nouvelle collection prédéfinie.

Array	ArrayList
La taille de la collection doit être précisée lors de l'instanciation. Cette taille ne peut ensuite pas être modifiée.	La taille de la collection ne doit pas être précisée au moment de l'instanciation. La collection se redimensionne lors de chaque ajout (<code>Add</code>) ou suppressions (<code>Remove</code> ou <code>RemoveAt</code>).
La collection est typée . MachineATickets[] tab = new MachineATickets [5] ; Seuls des objets <code>MachineATickets</code> peuvent y être rangés.	La collection est NON typée . ArrayList arl = new ArrayList() ; On peut y ranger des objets de n'importe quel type, et éventuellement de différents type en même temps.
La notation « crochets » extrait un objet typé . On peut écrire : tab[2].AccepteArgent(100) ;	La notation « crochets » extrait un objet NON typé . Pour en récupérer les caractéristiques, on doit forcer la conversion : (MachineATickets) carl[2] .AccepteArgent(100) ;

6.3 Méthodes RendReste, GetPrix et SetPrix

Les méthodes `RendReste`, `GetPrix` et `SetPrix` sont immédiates.

```
public int RendReste()
{
    //- Déclarations et initialisations des v. LOCALES -
    int retVal = this.soldeClient;

    //- Traitement -
    this.soldeClient = 0;
    return retVal;
}

//- Couple ACCESSEUR/MUTATEUR associé à la variable prixTicket
public int GetPrix()
{
    //- Traitement -
    return this.prixTicket;
}

public void SetPrix(int prixTicket)
{
    //- Traitement -
    if (prixTicket <= 0)
        throw new Exception("Prix négatif interdit");
    this.prixTicket = prixTicket;
}
```

Dans `SetPrix`, l'utilisation de `this` est **obligatoire**. En effet, le paramètre `prixTicket` masque la variable d'instance de même nom. Le seul moyen pour accéder à cette à cette variable d'instance est d'utiliser `this.prixTicket`.

Le couple de méthodes `GetPrix` et `SetPrix` illustre une nécessité fréquente : disposer dans l'interface publique d'un couple de méthodes pour consulter et modifier une variable d'instance sans briser l'encapsulation. Pour de telles méthodes, on parle d'**accesseur** et de **mutateur**.

6.4 Les constructeurs²

6.4.1 La syntaxe

Un **constructeur** est une méthode qui s'exécute lors de la construction d'un nouvel objet. Il a pour mission d'en créer les sous-objets et de les initialiser.

Tout constructeur **doit** posséder **deux caractéristiques** :

- **porter le même identificateur que la classe.**
- **ne pas posséder de type de retour.**

Il peut y avoir plusieurs constructeurs dans une classes, avec la même restriction d'unicité de signature que pour des méthodes surchargées.

`MachineATickets` possède deux constructeurs.

²

Je pense que la syntaxe des constructeurs est illogique (et pas qu'en C#). Un constructeur est conceptuellement une **méthode d'usine** : il serait plus logique qu'il soit statique (cf. chapitre 5).

- `MachineATickets()` est un **constructeur sans paramètre**.
- `MachineATickets(int)` est un **constructeur avec paramètre(s)**. Il pourrait y en avoir d'autres, sous réserve de respecter l'unicité des signatures.

6.4.2 Constructeur avec paramètre

Débutons avec l'analyse du constructeur avec paramètre.

```
public MachineATickets(int prixTicket)
{
    this.SetPrix(prixTicket);
    this.totalCaisse = 0;
    this.soldeClient = 0;
}
```

Il est utilisé quand on écrit :

```
maSeraing = new MachineATickets(250);
```

Dans le code, la validation du prix et son affectation à la variable d'instance sont délégués à la méthode `SetPrix`. Cette factorisation peut sembler « cosmétique ». Mais dans les cas réels, le code de validation est souvent complexe et sa duplication doit être évitée.

Quand une exception est levée durant l'exécution d'un constructeur, la construction en cours est INTERROMPUE. Autrement dit, aucun objet n'est construit.

Dans notre exemple, cela peut se produire **indirectement**. En effet, le constructeur invoque `SetPrix` qui peut éventuellement lever une exception. Si c'est le cas, vu que le constructeur ne prévoit pas de bloc d'interception, l'exception poursuit sa remontée. **Tout se passe comme si c'était le constructeur lui-même qui levait cette exception.** L'objet n'est **PAS** construit.

6.4.3 Constructeur sans paramètre

Le **constructeur sans paramètre** est invoqué lorsqu'aucun prix n'est précisé.

```
maVerviers = new MachineATickets();
```

Il fixe d'autorité le prix du ticket à 10 €. Voilà comment il pourrait être codé.

```
public MachineATickets()
{
    // - Traitement -
    this.SetPrix(1000);
    this.totalCaisse = 0;
    this.soldeClient = 0;
}
```

Mais cette approche revient à dupliquer la logique du code déjà présent dans le constructeur avec paramètre.

Pour éviter cela, le constructeur sans paramètre doit déléguer le travail au constructeur AVEC paramètre, en lui transmettant la valeur 1000 lors de l'appel.

La syntaxe à utiliser est particulière.

```
public MachineATickets() : this(1000)
{
```

| }

A la fin de l'en-tête, on ajoute `:this`, suivi de parenthèses qui contiennent le(s) paramètre(s) prévus par le constructeur invoqué. Le système choisit le constructeur à exécuter, sur base des mêmes règles que pour des méthodes surchargées.

Ici, le corps du constructeur par défaut est vide. Mais ce n'est pas une obligation. S'il contenait des instructions, elles seraient exécutées **après** le retour d'appel à l'autre constructeur (à un paramètre).

6.4.4 Constructeur sans paramètre implicite

SI ET SEULEMENT SI AUCUN constructeur n'est EXPLICITEMENT codé, le système prévoit un **constructeur sans paramètre IMPLICITE**. Ce dernier attribue à chaque variable d'instance une valeur par **défaut** choisie en fonction de type : 0 pour les nombres, `false` pour les booléens, `null` pour une référence, etc.

Attention. Ce mécanisme d'initialisation par défaut n'est valable que pour les variables d'instances **Par exemple, les variables locales à une méthode ne sont pas initialisées automatiquement.**

6.4.5 Règle à respecter

C# accepte des pratiques « hybrides ». Si elles sont parfois pratiques ou nécessaires, je les déconseille dans un premier temps. Tenez-vous en dans un premier temps au strict respect de la règle suivante.

Toute variable d'instance DOIT être initialisée par un constructeur.

7 Pour ne pas de se faire planter dans le plafond

Si son but est de créer un type neutre, une méthode de classe ne devrait **JAMAIS** contenir une instruction qui la lie à une famille de projets.

Une erreur classique est de polluer le code d'une classe avec une instruction de la forme `Console.WriteLine`. A moins que la classe ne soit strictement réservée qu'à une utilisation en mode console, c'est à **proscrire**.

Ce n'est QU'AU NIVEAU D'UN CODE « PUO » qu'une telle instruction peut se trouver !

