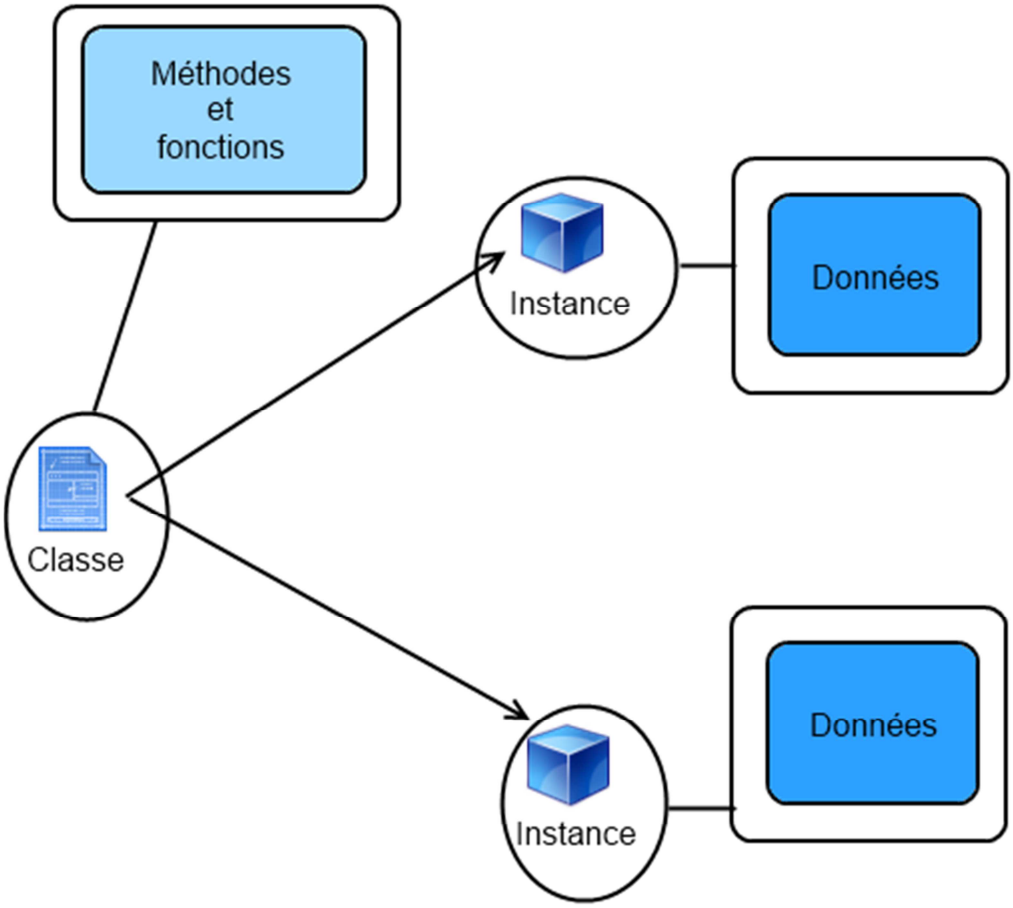
# La sérialisation des objets

## *Introduction à la sérialisation*

Sérialiser un objet, c'est prendre un cliché de son état actuel afin de pouvoir le retrouver plus tard dans les mêmes conditions.




Ce schéma illustre le faite que les instances d'une classe partagent le même code mais pas les données.  Le mécanisme de sérialisation ne se concentre que sur la portion *Données* de nos instances. C'est cette partie qui caractérise un objet et qui le distingue de toutes les autres instances. Sérialiser, c'est donc extraire les données d'une instance en vue de la désérialiser plus tard (à un prochain lancement ou lorsque les données se sont rendues sur un autre poste, par exemple). Sérialiser permet aussi d'échanger des données entre plusieurs langages.

## *Sérialisation binaire*

### Les spécifications de la sérialisation binaire

La sérialisation binaire crée un instantané d'un objet de manière plus compacte que la sérialisation XML.

Pour mettre en place la sérialisation binaire, chaque classe doit avoir l'attribut *Serializable* attaché à celle-ci.

### Comment faire pour sérialiser ?

Nous allons reprendre comme base un projet de jeu RPG.

Voici les différentes classes que nous utiliserons.

```
namespace TPSerialisation
{
    [Serializable]
    public abstract class Personnage
    {
        public string Nom { get; set; }

        public int Niveau { get; set; }

        public int PointVie { get; set; }
    }
}

namespace TPSerialisation
{
    [Serializable]
    public class Magicien : Personnage
    {
        public int PointMagie { get; set; }

        public string NomBaguette { get; set; }
    }
}

namespace TPSerialisation
{
    [Serializable]
    public class Guerrier : Personnage
    {
        public int Force { get; set; }

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

Avec le sérialisateur binaire nous n'avons pas besoin d'avoir des accesseurs en lecture et écriture. On est donc libres de faire ce que l'on veut avec nos règles d'encapsulation parce qu'en effet, elles ne seront pas prises en compte lors du processus de sauvegarder et de chargement. Cela vous permet de créer des classes plus solides, et qui gardent leur intégrité en tout temps.

Et maintenant, la méthode Main qui contient toute notre logique.

```
namespace TPSerialisation
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Personnage> equipe = Charger<List<Personnage>>("data.bin");

            if (equipe == null)
            {
                equipe = new List<Personnage>();

                equipe.Add(new Magicien() { Nom = "Gandalf", Niveau = 12, PointVie =
58, PointMagie = 200, NomBaguette = "Baton" });
                equipe.Add(new Guerrier() { Nom = "Arthur", Niveau = 14, PointVie =
200, Force = 62, NomEpee = "Excalibur" });
            }

            Enregistrer(equipe, "data.bin");
        }

        private static void Enregistrer(object toSave, string path)
        {
            // On utilise la classe BinaryFormatter dans le namespace
System.Runtime.Serialization.Formatters.Binary.
            BinaryFormatter formatter = new BinaryFormatter();

            // La classe BinaryFormatter ne fonctionne qu'avec un flux, et non pas un
TextWriter.
            // Nous allons donc utiliser un FileStream.
            FileStream flux = null;
            try
            {
                // On ouvre le flux en mode création / écrasement de fichier et on
                // donne au flux le droit en écriture seulement.
                flux = new FileStream(path, FileMode.Create, FileAccess.Write);

                // On sérialise
                formatter.Serialize(flux, toSave);

                // On s'assure que le tout soit écrit dans le fichier.
                flux.Flush();
            }
            catch { }
            finally
            {
                // On ferme le flux.
                if (flux != null)
                    flux.Close();
            }
        }
```

```
        private static T Charger<T>(string path)
        {
            BinaryFormatter formatter = new BinaryFormatter();
            FileStream flux = null;
            try
            {
                // On ouvre le fichier en mode lecture seule. De plus, puisqu'on a
sélectionné le mode Open,
                // si le fichier n'existe pas, une exception sera levée.
                flux = new FileStream(path, FileMode.Open, FileAccess.Read);

                return (T)formatter.Deserialize(flux);
            }
            catch
            {
                // On retourne la valeur par défaut du type T.
                return default(T);
            }
            finally
            {
                if (flux != null)
                    flux.Close();
            }

        }

    }
}
```

Nous utiliserons la classe BinaryFormatter pour sérialiser nos objets. Celle-ci doit travailler avec un flux. C'est pour ça qu'on utilisera un FileStream plutôt qu'un StreamWriter, car ce dernier hérite de TextWriter plutôt que de Stream. Pour ouvrir un FileStream, il existe plusieurs surcharges de constructeurs. Celle utilisé danz l'exemple prend trois paramètres et permet d'ouvrir un flux lié à un fichier exactement de la manière dont vous le désirez. D'abord, on spécifie le nom du fichier auquel le flux doit être lié. Ensuite, on spécifie le mode d'ouverture du fichier. Il existe plusieurs énumérations qui permettent de faire exactement ce que l'on veut :

- **Append**. Ce mode ouvre un fichier et "accroche" le résultat à la fin de celui-ci. Si le fichier n'existe pas, il est créé.
- **Create**. Le mode Create spécifie que peu importe ce qui se passe, un nouveau fichier doit être créé. Si un fichier du même nom existe, il sera écrasé. Attention, il faut avoir les permissions d'écriture sur le fichier et dans le répertoire, sinon une exception est levée.
- **CreateNew**. Ce mode spécifie qu'un fichier doit être créé, mais que s'il existe déjà, une exception sera levée. Attention à avoir les droits d'écriture sur ce mode également.
- **Open**. Ce mode est un mode de lecture. Si le fichier n'existe pas ou si le programme n'a pas les droits de lecture, une exception est levée.
- **OpenOrCreate** spécifie que le fichier doit être ouvert, mais que s'il n'existe pas, il doit être créé. Une exception peut être lancée si les droits d'accès ne sont pas valides.

- **Truncate** ouvre un fichier et le réinitialise à 0 octets. Essayer de lire un tel flux est illégal et renverra une exception.

Comme vous le voyez, il y a plusieurs modes d'ouverture. Cependant, en troisième paramètre, on spécifie l'accès que le flux aura au fichier.

- **Write** ne permet que d'écrire dans un fichier. C'est idéal pour notre méthode d'enregistrement.
- **Read** ne permet que la lecture d'un fichier.
- **ReadWrite** permet de faire ce que l'on désire dans le fichier.

Remarquez aussi que l'on utilise une structure sécuritaire afin d'accéder aux fichiers sur le Disque Dur. De plus, en employant un bloc finally, on est certain que peu importe ce qui se passera, le flux sera fermé à la fin de l'opération, que ce soit l'ouverture du flux ou la sérialisation qui échoue.

Vous remarquerez aussi qu'une méthode générique est utilisée pour le chargement. Cela permet de simplifier la lecture des flux. Afin de retourner une valeur par défaut (null dans le cas des types références, mais 0 dans le cas d'un int par exemple), on utilise l'opérateur default.

**Ignorer des champs lors de la sérialisation**

Pour ignorer un ou plusieurs champs lors de la sérialisation, il faudra utiliser l'attribut **NonSerialized**. Il faut savoir que le sérialisateur binaire ne sérialise que les variables membres, et rien d'autre. De plus, comme ce sérialisateur ne prend pas en compte les accesseurs pour faire son boulot. Ainsi certaines informations ne devraient pas être sérialisées, le seront quand même par défaut. C'est là qu'entre en vigueur **NonSerialized**. Voici un exemple de ce qu'il faudra faire pour éviter une sérialisation.

```
namespace TPSerialisation
{
    [Serializable]
    public abstract class Personnage
    {
        [NonSerialized]
        private string _nomFamilier;

        public string Nom { get; set; }

        public string NomFamilier { get { return _nomFamilier; } set { _nomFamilier =
value; } }

        public int Niveau { get; set; }

        public int PointVie { get; set; }
    }
}
```

Notez également que lors de la désérialisation, le constructeur de l'objet n'est pas appelé. On le restaure tel quel ! Ainsi, on ne pourra pas faire de vérifications automatisées afin de restaurer un état particulier. Il faudra programmer une méthode et l'appeler lorsque nécessaire.