Module 12

Utiliser les Collections et Construire des types Génériques

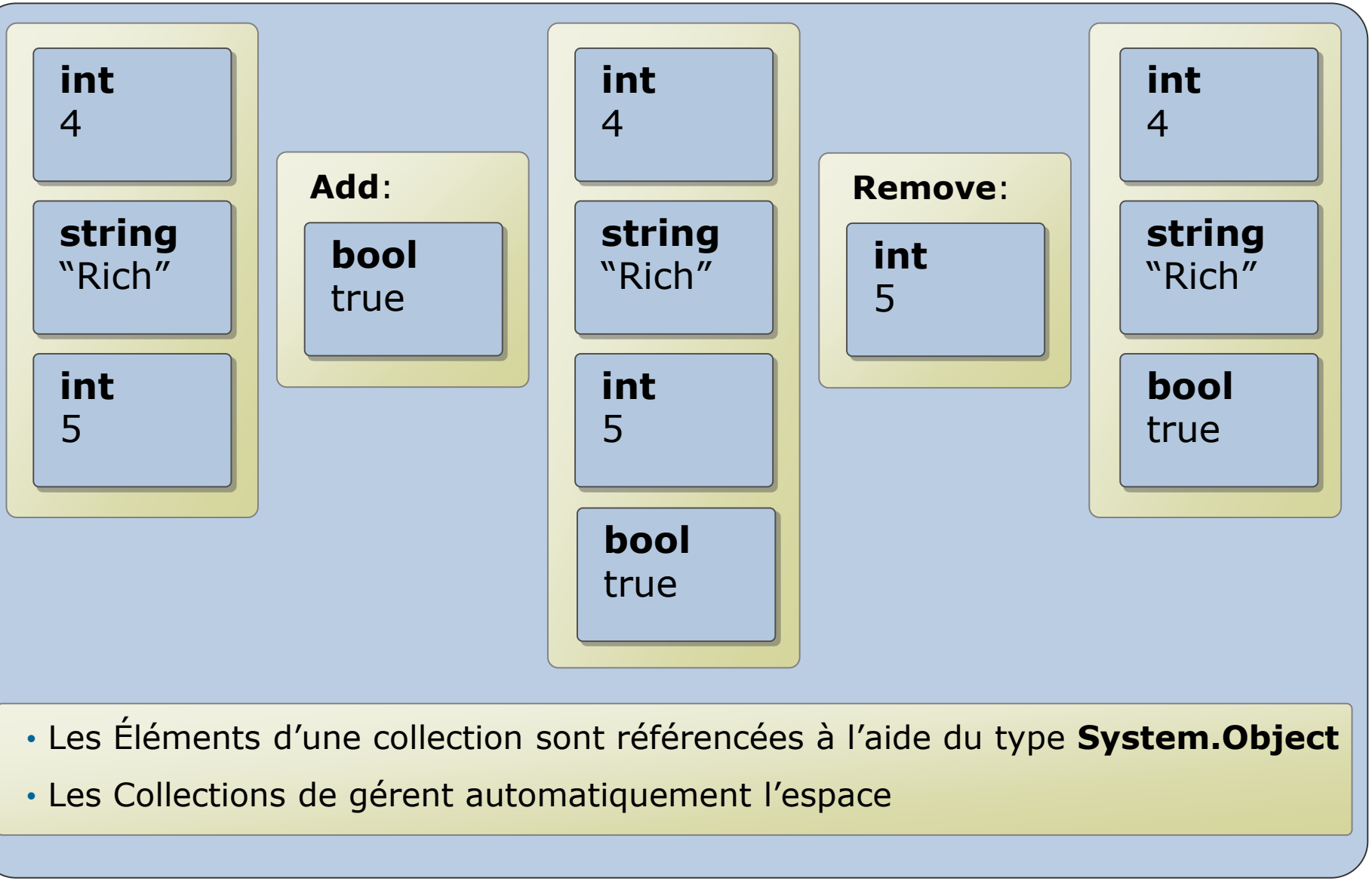
Sommaire

- Utiliser des Collections
- Créer et Utiliser des types Génériques
- Définir des interfaces Génériques et Comprendre la Variance
- Utiliser des Méthodes et des Délégués Génériques

Leçon 1: Utiliser des Collections

- Qu’est-ce qu’une Collection?
- Utiliser des Classes de Type Collection
- Itérer dans une Collection
- Classes de Type Collection Usuelles
- Utiliser des Initialiseurs de Collection

Qu’est-ce qu’une Collection?



- Les Éléments d’une collection sont référencées à l’aide du type **System.Object**
- Les Collections de gérent automatiquement l’espace

Utiliser des Classes de Type Collection

Interfaces ICollection :

CopyTo

GetEnumerator

Count

Implémentée par toutes les classes de type collection

interface IList :

Add

Remove

Implémentée par certaines classes de type collection

Les éléments sont stockés dans les collections en tant qu'objets. Vous devez effectuer un cast des éléments qui proviennent de la collection
Certaines colections fournissent des alternatives comme **Push** et **Pop** à la place de **Add** et **Remove**

```
ArrayList list = new ArrayList();

list.Add(6);

list.Remove(6);

list.RemoveAt(1);

int temp = (int)list[0]);
```

Itérer dans une Collection

Une boucle **foreach** affiche chaque élément de la colleciton

```
foreach(<type> <control_variable> in <collection>)
{
   <foreach_statement_body>
}
```

Pour utilizer une boucle **foreach**, la collection doit exposer un énumérateur.

L'interface **ICollection** définit une méthode **GetEnumerator**

```
ArrayList list = new ArrayList();
list.Add(99);
list.Add(10001);
list.Add(25);
...
foreach (int i in list)
{
   Console.WriteLine(i);
}
// Output: 99
//         10001
//         25
```

Classes de Type Collection Usuelles

ArrayList

Collection non triée, semblable à un tableau. Les éléments sont accessibles par index

Queue

Collection first-in, first-out .Utiliser la méthode **Enqueue** à la place de **Add**

Stack

Collection first-in, last-out collection. Utiliser la méthode **Push** à la place de **Add**

Hashtable

Collection de paires clé / valeur. Adapté aux grandes collections

SortedList

Collection de paires clé / valeur. Les éléments sont triés selon la clé

Utiliser des Initialiseurs de Collection

Vous pouvez utilizer la méthode **Add** pour ajouter des éléments à une collection

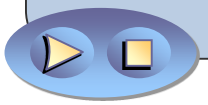
```
ArrayList al = new ArrayList();
al.Add("Value");
al.Add("Another Value");
```

Vous pouvez également utiliser un initialiseur de collection pour ajouter des éléments à une collection, lorsque vous définissez la collection

```
// Assume person1 and person 2 are instantiated
// Person objects.
ArrayList al2 = new ArrayList()
{
    person1,
    person2
};
```

Vous pouvez combiner les initialiseurs de collection avec des initialiseurs d'objets

```
ArrayList al3 = new ArrayList()
{
    new Person() {Name="James", Age =45},
    new Person() {Name="Tom", Age =31}
};
```



Lab A: Using Collections

- Exercise 1: Optimizing a Method by Caching Data

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa$$w0rd

Estimated time: 20 minutes

Leçon 2: Créer et Utiliser des types Génériques

- Qu’est-ce que les Type Génériques?
- Compilation de Types génériques et Sécurité de Type
- Définir un Type Générique Personnalisé
- Ajouter de contraintes aux Types génériques

Qu’est-ce que les Type Génériques?

Un type générique est un type qui spécifie un ou plusieurs paramètres de type

Les paramètres de type sont comme les autres paramètres sauf qu’ils représentent un type, pas une instance d’un type

Paramètres de type sont définis à l’aide de brackets après le nom de la classe

```
public class List<T>
```

Cet exemple de code illustre la définition d’une classe nommée **List** qui prend un seul paramètre de type nommé **T**. Vous pouvez utiliser **T** comme tout autre type de la classe

Compilation de Types génériques et Sécurité de Type

La classe **List<T>** est utilisée plusieurs fois avec différents type de paramètres

```
List<string> names = new List<string>();
names.Add("John");
...
string name = names[0];

List<List<string>> listOfLists = new List<List<string>>();
listOfLists.Add(names);
...
List<string> data = listOfLists[0];
```

Le compilateur génère un équivalent fortement typé de la classe générique, générant les méthodes suivantes

```
public void Add(string item)

...

public void Add(List<string> item)
```

Les classes and méthodes générées par le compilateur, le sont au moment de la compilation de l'application. De ce fait, en tant que développeurs, vous ne pouvez pas utilisez ces versions fortement typées

Définir un Type Générique Personnalisé

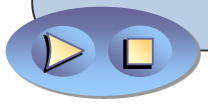
Définissez une classe et spécifiez un paramètre entre les brackets après le nom de la classe

```
class PrintableCollection<TItem>
{
    TItem [] data;
    int index;
    ...

    public void Insert(TItem item)
    {
        ...
        data[index] = item;
        ...
    }
}
```

Utilisez le paramètre de type comme alias pour le type dans les champs et les propriétés

Vous pouvez également utiliser le paramètre de type dans les méthodes



Ajouter de contraintes aux Types génériques

Contrainte	Description
where T: struct	L'argument de type doit être un type par valeur
where T : class	L'argument de type doit être un type par référence
where T : new()	L'argument de type doit posséder un constructeur par défaut
where T : <base class name>	L'argument de type doit être, ou dériver de la classe de base spécifiée
where T : <interface name>	L'argument de type doit être, ou implémenter, l'interface spécifiée
where T : U	L'argument de type fournit pour T doit être ou hériter du type spécifié pour **U**

Leçon 3: Définir des interfaces Génériques et Comprendre la Variance

- Définir des Interfaces Génériques
- Qu’est-ce que l’Invariance?
- Définir et Implémenter une Interface Covariante
- Définir et Implémenter une Interface Contravariante

Définir des Interfaces Génériques

Interface Générique avec Paramètre de Type

Classe Générique qui implémente l'interface générique

```
interface IPrinter<DocumentType>
    where DocumentType : IPrintable
{
    void PrintDocument(DocumentType Document);

    PrintPreview PreviewDocument(DocumentType Document)
}

class Printer<DocumentType> : IPrintable<DocumentType>
    where DocumentType : IPrintable
{
    public void PrintDocument(DocumentType Document)
    {
        // Send document to printer.
        PrintService.Print((IPrintable)Document);
    }

    public PrintPreview PreviewDocument(DocumentType Document)
    {
        // Return a new PrintPreview object.
        return new PrintPreview((IPrintable)Document);
    }
}
```

Qu’est que l’Invariance?

L’invariance vous empêche de caster des paramètres de type en d’autres types dans la hierarchie de l’héritage

IGenericInterface<string> ne peut pas être castée en **IGenericInterface<Object>** même si elles sont dans la même hiérarchie d’héritage

Sans invariance, vous pourriez caster un string en object. Si le string est utilize comme parameter de méthode, le code pourrait lever des exceptions

Les interfaces Génériques sont invariantes par défaut; les classes génériques sont tout le temps invariantes

Les interfaces Invariantes peuvent être suffisament souples; La covariance et la contravariance peuvent être utilisées pour les paramètres de type appropriés

Définir et Implémenter une Interface Covariante

Vous pouvez specifier le mot clé **out** avec le paramètre de type dans une interface générique si le paramètre de type parameter est toujours utilisé comme type de retour

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

L'utilisation de la covariance, vous permet de réaliser le cast suivant:

```
// Wrapper implements IRetrieveWrapper
IRetrieveWrapper<string> stringWrapper = new Wrapper<string>;

// Without covariance this code would not be legal.
// All strings are objects so a method which returns a string also returns an object.
IRetrieveWrapper<object> objectWrapper  =stringWrapper;
```

Définir et Implémenter une Interface Contravariante

Vous pouvez specifier le mot clé **in** avec le paramètre de type dans une interface générique si le paramètre de type parameter est toujours utilisé comme paramètre de méthode

```
interface ISetWrapper<in T>
{
    void SetData(T item);
}
```

L'utilization de la contravariance, vous permet de réaliser le cast suivant

```
// Wrapper implements ISetWrapper
ISetWrapper<object> objectWrapper = new Wrapper<object>;

// Without contravariance this code would not be legal.
// All strings are objects so a method which accepts an object can also accept a string.
ISetWrapper<string> stringWrapper  = objectWrapper;
```

Leçon 4: Utiliser des Méthodes et des Délégués Génériques

- Qu’est-ce que des Méthodes et des Délégués Générics?
- Utiliser les Délégués Génércis Inclus dans le framework .NET
- Définir une Méthode Générique
- Utiliser des Méthodes Génériques

Qu’est-ce que des Méthodes et des Délégués Générics?

```
void AddToQueue(Report report)
{
   printQueue.Add(report);
}
```

```
void AddToQueue(ReferenceGuide referenceGuide)
{
   printQueue.Add(referenceGuide);
}
```

Au lieu de dupliquer des méthodes avec des paramètres différents, vous pouvez utiliser une méthode générique avec un paramètre de type

```
void AddToQueue<DocumentType>(DocumentType document)
{
   printQueue.Add(report);
}
```

```
delegate void PrintDocumentDelegate<DocumentType>(DocumentType document);
```

Utiliser les Délégués Génércis Inclus dans le framework .NET

Action<T>

Vous pouvez utiliser le délégué **Action** au lieu de déclarer des délégués personnalisés pour les méthodes avec aucun type de retour

Func<T, TResult>

Vous pouvez utiliser le délégué **Func** au lieu de déclarer des délégués personnalisés pour les méthodes avec un type de retour. Le paramètre de type **TResult** représente le type de retour et est toujours le dernier paramètre de type

Le Framework.Net comprend des surcharges pour les délégués **Action** and **Func** comprenant jusqu'à 16 paramètres

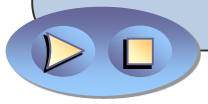
Définir une Méthode Générique

Définir une méthode et spécifier un paramètre de type (ou plusieurs) entre bracket près le nom de la méthode

```
ResultType MyMethod<Parameter1Type, ResultType>(Parameter1Type
param1)
     where ResultType : new()
{
     ResultType result = new ResultType();
     return result;
}
```

Spécifier des contraintes sur les paramètres de type

Utilisez le paramètre de type dans les paramètres de la méthode, le type de retour et le corps de la méthode=n



Utiliser des Méthodes Génériques

Pour appeler une méthode générique, spécifiez le paramètre de type

```
T PerformUpdate<T>(T input)
{
    T output = // Update parameter.
    return output;
}
...

string result =  PerformUpdate<string>("Test");
int result2 = PerformUpdate<int>(1);
```

Si vous appelez une méthode générique, le compilateur génère une méthode concrète pour chaque appel

```
string PerformUpdate(string input) // You cannot call this method.
{
    string output = // Update parameter.
    return output;
}

int PerformUpdate(int input) // You cannot call this method.
{
    int output = // Update parameter.
    return output;
}
```

Atelier Pratique

- Exercice 1:
- Exercice 2:
- Exercice 3: