Generics
in C#

Introduction

- Generic means the general form, not specific.
- In C#, generic means not specific to a particular data type.

- C# allows you to define generic classes, interfaces, fields, methods, properties and events using the type parameter and without the specific data type.
- A type parameter is a placeholder for a particular type specified when invoking the generic member.
- A generic type is declared by specifying a type parameter in an angle brackets after a type name, e.g. `TypeName<T>` where `T` is a type parameter.

Introduction

- With generics, you can create classes and methods that are independent of contained types.
- Instead of overloading methods with the same functionality for different types, you can create just one method.

- Another option to reduce the amount of code is using the `Object` class.
- However, passing types derived from the `Object` class is not type safe.
- Generic methods make use of generic types that are replaced with specific types as needed.
- This allows for type safety: the compiler complains if a specific type is not supported with the generic method.

Performance

- One of the big advantages of generics is performance.
- Using value types with non-generic methods results in *boxing* and *unboxing* when the value type is converted to a reference type, and vice versa.
- Value types are stored on the stack, while reference types are stored on the heap.
- C# classes are reference types; primitive types (`int`, `double`) are value types.
- .NET makes it easy to convert value types to reference types, so you can use a value type everywhere an `object` (which is a reference type) is needed.
- For example, an `int` can be assigned to an `object`.

Performance

- The conversion from a value type to a reference type is known as boxing.
- Boxing occurs automatically if a method requires an `object` as a parameter, and a value type is passed.
- On the other hand, a boxed value type can be converted to a value type by using unboxing.
- With unboxing, the cast operator is required.
- Boxing and Unboxing refer to the process of converting a value type to a reference type and vice versa.

Boxing

- Boxing is the process of converting a value type to a reference type.
- When you box a value type, a new object is created on the heap, and the value of the value type is copied into that object.
- The new object is then treated as a reference type.

```
int intValue = 10;
object boxedObject = intValue;   // Boxing occurs here
```

- In this example, the `int` variable `intValue` is boxed into an `object` variable `boxedObject`.
- Now, `boxedObject` holds a reference to a new object on the heap that contains a copy of the original `int` value.

Unboxing

- Unboxing is the process of converting a reference type to a value type.
- It involves extracting the value from the boxed object and assigning it to a value type variable.

```
object boxedObject = 10;
int intValue = (int)boxedObject; // Unboxing occurs here
```

- In this example, the `object` variable `boxedObject` holds a boxed `int`.
- The value is extracted through unboxing and assigned to the `int` variable `intValue`.

Unboxing

- Note that unboxing can lead to runtime errors if the boxed object is not of the expected type.
- Therefore, you should always perform appropriate type checks to ensure safe unboxing:

```
object boxedObject = 10;

if (boxedObject is int)
{
    int intValue = (int)boxedObject; // Unboxing occurs here
                                     // Do something with intValue
}
```

Boxing and Unboxing

- Boxing and unboxing are easy to use but have a big performance impact, especially when iterating through many items.
- Instead of using objects, a generic method enables you to define the type-safe method when it is used.

Type Safety

- Another feature of generics is type-safety.
- With a non-generic method, if `object`s are used, any type can be passed to a method.
- In some contexts, this may not have relevance.
- It may also result in run-time exceptions.
- Errors should be detected as early as possible.
- With the generic method, the generic type `T` defines what types are allowed.

Example - Non-Generic Method

- Assume we made a method AreEquals which is a non-generic method that accepts two parameters of type object and returns boolean if both values are equal or not.

```
public static bool AreEqual(object value1, object value2)
{
       return value1.Equals(value2);
}
```

- The problem here is that it is not type safe.
- Any type of values can be passed in for comparison.
- Boxing happens as int values are being passed to object.

```
bool result = AreEqual(10, 10);        // this is ok

bool result = AreEqual(10, "hello");   // not relevant
```

Example - Generic Method

- This time, AreEquals is a generic method that accepts two parameters of type T and return boolean if both values are equal or not.

```
public static bool AreEqual<T>(T value1, T value2)
{
       return value1.Equals(value2);
}
```

- This is a type-safe method.
- No boxing happening, as int values are passed to int parameters.

```
bool result = AreEqual<int>(10, 10);          // type-safe, only int
                                              // values will be accepted

bool result = AreEqual<int>(10, "hello");     // Compile-time error
```

Arrays

- If you need to use multiple objects of the same type, you can use an array.
- An array is a data structure that contains a number of elements of the same type.
- An array is a reference type.
- Different ways to create an array of size 4:

```
int[] myArray = new int[4];

int[] myArray = new int[4] {4, 7, 11, 2};

int[] myArray = new int[] {4, 7, 11, 2};

int[] myArray = {4, 7, 11, 2};
```

Accessing Array Elements

- You can access the array elements using an indexer.
- Arrays support only indexers that have integer parameters.
- With the indexer, you pass the element number to access the array.
- The indexer always starts with a value of 0 for the first element.
- Therefore, the highest number you can pass to the indexer is the number of elements minus one, because the index starts at zero.

Accessing Array Elements

- In the following example, the array myArray is declared and initialized with four integer values.
- The elements can be accessed with indexer values 0, 1, 2, and 3.

```
int[] myArray = new int[] {4, 7, 11, 2};

int v1 = myArray[0];  // read first element

int v2 = myArray[1];  // read second element

myArray[3] = 44;      // change fourth element
```

Array Size is Static

- An array’s size is set at compile-time and is static.
- An array cannot be resized after its size is specified without copying all the elements to a new array.
- If you use a wrong indexer value where it is bigger than the length of the array, an exception of type `IndexOutOfRangeException` is thrown.
- If you don’t know the number of elements in the array, you can use the `Length` property.

```
for (int i = 0; i < myArray.Length; i++)
{
     Console.WriteLine(myArray[i]);
}
```

Array Pros and Cons

- One good thing about array is that it is type-safe.
- If you define an `int` array, you cannot assign a value of other type.
- Otherwise, it’ll result in compile-time error.

```
// Arrays are type-safe
int[] myArray = new int[4];
myArray[0] = 101;
myArray[1] = 102;
myArray[2] = 103;
myArray[3] = 104;

myArray[3] = "hello"; // cannot be done, compile-time error
```

Array Pros and Cons

- One limitation of array is that it is static in size.
- If an array is defined with size 4, then you cannot assign a value to an index beyond 3 of that array.
- If you do, it’ll result in run-time error.

```
// Array size is static
int[] myArray = new int[4];
myArray[0] = 101;
myArray[1] = 102;
myArray[2] = 103;
myArray[3] = 104;

myArray[4] = 105;     // run-time error - IndexOutOfRangeException
```

ArrayList

- ArrayList is an alternative to arrays.
- Items can be added and removed dynamically, and the ArrayList resizes itself.
- ArrayList belongs to System.Collections namespace.
- Add() method can be used to add items.

```
ArrayList myArrayList = new ArrayList();
myArrayList.Add(10);
myArrayList.Add(20);
myArrayList.Add(30);
myArrayList.Add(40);
```

ArrayList Pros and Cons

- ArrayList is not type-safe.
- The Add() method of ArrayList accepts a parameter of type object.
- This means an item of any type can be added to ArrayList, which later could result in run-time exception.

```
ArrayList myArrayList = new ArrayList();
myArrayList.Add(10);
myArrayList.Add(20);
myArrayList.Add(30);
myArrayList.Add(40);

myArrayList.Add("hello"); // could result in run-time error
```

ArrayList Pros and Cons

- Another issue with ArrayList is performance degradation.
- ArrayList stores objects; the Add() method is defined to require an object as a parameter, so an integer type is boxed.
- When a value from an ArrayList is retrieved, unboxing occurs when the object is converted to an integer type.
- While retrieving items from ArrayList, casting is required.
- **Warning:** Do not use ArrayList.

```
ArrayList myArrayList = new ArrayList();

myArrayList.Add(44);      // boxing – convert a value type to a reference type

int num = (int)myArrayList[0];   // unboxing – convert a reference type to
                                 // a value type
```

Exercise

- Write a generic method `PrintArray` that can accept an array of any type.
- It prints the number of items in the array.
- Then, it prints the array items.

Do It Yourself!

- **Exercise: Overloading a Generic Method:**
- Overload generic method PrintArray (from last exercise) so that it takes two additional int arguments: lowIndex and highIndex.
- A call to this method displays only the designated portion of the array.
- Validate lowIndex and highIndex. If either is out of range, or if highIndex is less than or equal to lowIndex, the overloaded PrintArray method should throw an Exception.
- Then modify Main to exercise both versions of PrintArray on arrays intArray, doubleArray and charArray.

```
// Generic method that prints an array
public static void PrintArray<T>(T[] array)
{
    foreach (T i in array)
        Console.Write(i + " ");
    Console.WriteLine();
}
```

Do It Yourself!

- **Exercise: Generic Linear Search Method:**
- Write a generic method `Search`, that searches an array using the linear-search algorithm.
- Method `Search` should compare the search key with each element in its array until the search key is found or until the end of the array is reached.
- If the search key is found, return its location in the array; otherwise, return -1.

- Implement the `Search` method in the `Main` method.
- Randomly generate values for the `int` and `double` arrays.
- Display the generated values, so the user knows what values they can search for.

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Material has been taken from:

- Visual C# 2012: How to Program:
- https://learning.oreilly.com/library/view/visual-c-2012/9780133380170/ch20.html

- Professional C# 7 and .NET Core 2.0:
- https://learning.oreilly.com/library/view/professional-c-7/9781119449270/c05.xhtml