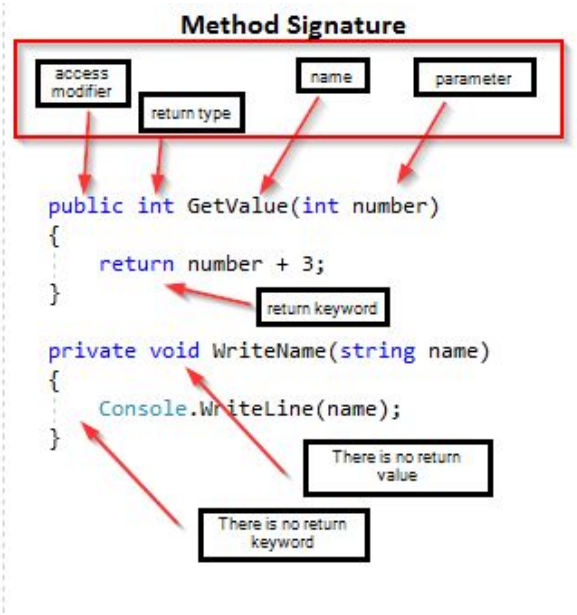
Sarah Coufal, Elly Fernández, Kejing Wang, Ziyao Wang
Deepak Kumar
Principles of Programming Languages
3 December 2020

C# Final Report

Methods



Methods in C# are almost identical to those in Java.

Return Value: types and number

C# method can return void or any data type specified in the signature but cannot return functions. C# neither has built-in support for multiple return values, but the same effect can be achieved by using Array, Tuple and Struct, … e.t.c.

Parameter Association and Default Parameter

C# used to have only positional parameter association. However, from C# 4, we have named arguments and default/optional parameters. Named arguments have the format "name: input value/variable" and they can be in any order. However, named arguments mixed with positional arguments are valid only in the order of formal parameters.

```
// Assume that we have a function as below
Public void PrintInfo(string firstName, string lastName, int age){
        ...
};
// The method can be called in the normal way, by using positional arguments.
PrintInfo("Tom", "Hanks", 64);
// Named arguments can be in any order.
```

```
PrintInfo(age: 64, lastName: "Hanks", firstName: "Tom");
// Named arguments mixed with positional arguments are valid
// as long as they are used in the order of formal parameters
PrintInfo("Tom", "Hanks", age: 64);
```

Optional parameters are defined at the end of the parameter list, after any required parameters. Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used. Any call must provide arguments for all required parameters, but can omit arguments for optional parameters.

```
//example of default/optional parameter
public static void Method(int required, string optionalStr = "default string", int
optionalInt = 10){
    Console.WriteLine($"{required}, {optionalStr}, {optionalInt}.");
}
//no input for default parameters
//output: 3, default string, 10
Method(3);
//change default values provided
//output: 3, a new string, 4
Method(3, "a new String", 4);
//To skip some optional parameters, use named arguments to specify which optional
parameters you want to change.
//output: 3, default string, 4
Method(3, optionalInt: 4);
```

By using the *params* keyword, you can specify a method that takes a variable number of arguments. In the function definition, we use params keyword followed by an array of any type specified by the user. The params + array parameter should be the last parameter of the method definition and no more parameters are permitted after. Params keyword can only be used once in the method.

```
// example of using params keyword
public static void UseParams(string required, params object[] list)
{
    Console.Write(required + ", ");
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + ", ");
    }
    Console.WriteLine();
}
// A params parameter accepts zero or more arguments.
UseParams("required input");
// You can send a comma-separated list of arguments of the specified type.
UseParams("required input", 1, "test");
```

```
// You can pass an array of the type specified in the function signature.
// Here object type allows variable input of different types.
object[] myObjArray = { 2, 'b', "test", "again" };
UseParams("required input", myObjArray);
```

Parameter Passing Schemes: ref, in, out keywords

In terms of parameter passing scheme, C# has both pass-by-value and pass-by-reference. By default, parameters are passed by value unless specified by the keyword *ref*. A method parameter modified by the *ref* keyword indicates that the parameter is passed by reference. The following example shows how to use the *ref* keyword: both the method definition and the calling method must explicitly use the *ref* keyword, but inside function only the variable name is used. In addition, as C# has both value model and reference model for different types and a method parameter can be modified by *ref* regardless of whether it is a value type or a reference type. This gives a lot of flexibility to what users can do because they have all combinations between value/reference model and passing schemes: pass a value type by value, pass a value type by reference, pass a reference type by value, or pass a reference type by reference.
For example, when passing a reference type by reference(passing an array, a reference type in C#, by reference), you can make the variable of the array point to a different array inside of the function. This is something we cannot do in Java.

```
// how to use ref keyword
void Method(ref int arg)
{
    arg = arg + 44;
}
int number = 1;
Method(ref number);


//pass a reference type by reference
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
static void Change(ref int[] pArray)
{
    pArray = new int[5] {-3, -1, -2, -3, -4};
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}

int[] arr = {1, 4, 5};
```

```
System.Console.WriteLine("Inside Main, before calling the method, the first element
is: {0}", arr[0]);
Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first element
is: {0}", arr[0]);
```

The *in*, *out* keywords also cause a parameter to be passed by reference. However, compared to *ref*, they have an emphasis on which role the parameter is playing (*in* parameter or *out* parameter) .

A variable modified by *in* serves as an input parameter. So C# requires that it must be initialized before being passed into the method, and may not be modified inside the method.

```
int arg = 44;
Example(in arg);
void Example(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

A variable modified by *out* serves as an out parameter to store the results of computation during the method. So if the variable is not initialized when it's passed into the method, the method is required to assign a value to the variable before it returns. The method is allowed to modify the variable.

```
int initializeInMethod;
Example(out initializeInMethod);
Console.WriteLine(initializeInMethod);// value is now 44
void Example(out int number)
{
    number = 44;
}
```

Types:

C# is considered as a strongly typed language and variables are statically typed.

Arrays:

Arrays are used to store objects of the same type. You declare an array by stating the type of the elements and then assigning a name to the array:

```
type[] arrayName;
```

There are multiple ways to initialize an array. In this example, an array of 5 integers is declared.

```
int[] array = new int[5];
```

By default, the values of the array are set to zero for numeric arrays. If the elements are reference types, they are set to null. The array element values can then be set using the index that they will be stored in (the index is inside the square brackets):

```
array[0] = 1;
array[1] = 2;
array[2] = 3;
array[3] = 4;
array[4] = 5;
```

Arrays are zero indexed, which means the first index of the array is 0, and the last is n-1. In this example, n=5 so the last index of the array is 4. Alternatively, arrays can be declared and initialized in one statement as follows:

```
int[] array = new int[] {1, 2, 3, 4, 5};
```

To access an element in an array, you use the name of the array followed by square brackets that contain the index of the element. You can also access elements from the end of the array using the ^ operator, where *^0* is equivalent to the length of the array. Using the array from the example above, this code would print the first two elements of the array, 1 and 2.

```
Console.WriteLine(array[0]);
Console.WriteLine(array[1]);
```

This code would print the last two elements of the array, 5 and then 4.

```
Console.WriteLine(array[^1]);
Console.WriteLine(array[^2]);
```

Arrays can be single-dimensional, multidimensional, or jagged (an array of arrays). When an instance of an array is created, the dimensions and length of the array are set and cannot be changed.

The .. operator is used to get an array slice.

```
int[] array2 = array[1..3];
```

The number before the .. operator is the start index (if omitted, the start index is 0), which is inclusive, and the number after the .. operator is the end index (if omitted, it goes to the end of the array), which is exclusive. So the code above will create an array with elements 2 and 3 in it.

Enumeration types:

C# has enumeration types. Enumeration types enable you to define your own type name and literals. To define an enumeration type, use the enum keyword and specify the names of enum members. By default, the associated constant values of enum members are of type int. Can explicitly specify type and the associated constant values.

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

For any enumeration type, there exist explicit conversions between the enumeration type and its underlying integral type. If you cast an enum value to its underlying type, the result is the associated integral value of an enum member.

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output:
Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

Composite Data Types:

Tuples:

A lightweight structure that contains two or more fields whose types are predefined.

Array Types:

```
Dim arrayA( ) As Byte = New Byte(12) {}
Dim arrayB( ) As Byte = New Byte(100) {}
Dim arrayC( ) As Short = New Short(100) {}
Dim arrayD( , ) As Short
Dim arrayE( , ) As Short = New Short(4, 10) {}
```

In the preceding example, array variables arrayA and arrayB are considered to be of the same data type — Byte() — even though they are initialized to different lengths. Variables arrayB and arrayC are not of the same type because their element types are different. Variables arrayC and arrayD are not of the same type because their ranks are different. Variables arrayD and arrayE have the same type — Short(,) — because their ranks and element types are the same, even though arrayD is not yet initialized.

Class Types:

There is no single data type comprising all classes. Although one class can inherit from another class, each is a separate data type. Multiple instances of the same class are of the same data type. If you assign one class instance variable to another, not only do they have the same data type, they point to the same class instance in memory.

Type Checking: type-testing operators and cast expressions

is operator:

The *is* operator checks if the runtime type of an expression result is compatible with a given type. `E is T`. E is an expression that returns a value and T is the name of a type or a type parameter. E cannot be an anonymous method or a lambda expression. The E is T expression returns true if the result of E is non-null and can be converted to type T by a reference conversion, a boxing conversion, or an unboxing conversion; otherwise, it returns false. The is operator doesn't consider user-defined conversions.

```
// Base is a base class, and Derived is a derived class that extends Base)
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

typeof operator:

The argument to the typeof operator must be the name of a type or a type parameter.

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

Cast expression:

A cast expression of the form (T)E performs an explicit conversion of the result of expression E to type T. If no explicit conversion exists from the type of E to type T, a compile-time error occurs.

Generics:

Generics are used to design methods and classes that do not specify type until they are instantiated. This allows programmers to write methods and classes that can be used with any

data type, so code can be reused instead of requiring different code for each type. The following example demonstrates a generic linked-list class:

```
// type parameter T in angle brackets
public class GenericList<T>
{
        // The nested class is also generic on T.
        private class Node
        {
            // T used in non-generic constructor.
            public Node(T t)
            {
                next = null;
                data = t;
            }
        //other variables and methods
        }
   private Node head;

   // constructor
   public GenericList()
   {
       head = null;
   }
   //other variables and methods
}
```

To use the generic list to create a list of integers, the client code would use the following:

```
GenericList<int> list = new GenericList<int>();
```

This code can be used to create a list of any type simply by replacing *int* inside the < > with the given type. By using generics, programmers can write code that is efficient, reusable, and type-safe.

Structs and Classes:

C# has multiple types that are used to make instances or objects: structure types (struct) and classes. A struct is a value type, meaning a variable of struct type holds the actual value of the data. When a struct is assigned to another variable, a new copy is made and changes to one does not affect the other. On the other hand, a class is a reference type, so an object of a class holds a reference to memory, not the actual data. As a result, when an object is assigned to

another variable, changes to one of the variables affect both variables. Here is an example of a struct:

```
public struct Person
{
   public string Name;
   public int Age;
   public Person(string name, int age)
   {
       Name = name;
       Age = age;
   }
}
```

An instance of a struct is created and initialized using the constructor:

```
Person p1 = new Person("Alex", 9);
```

Although the *new* operator is used here, it is not always required when creating a struct instance:

```
Person p2 = p1;
```

In this example, p2 is a new struct object, and changes to p1 will not affect p2 and vice versa.

Classes have constructors, methods, and properties; together these are called class members. There can be multiple constructors for a single class, each one taking a different set of parameters, and if one is not provided, then C# creates one and defaults the object's values to the defaults defined by the language. Constructors are methods with the same name as that of the class, and whose signature includes the method name and parameter list and no return type. Properties are a key part of encapsulation in C#; they provide public ways of reading, writing, and computing values of private fields (and are also known as accessors). The keywords associated with properties are *get*, *set*, and *value*. *Get* and *set* are used to return the property's value and to assign a new value respectively, and *value* with *set* and references the value that the cluine code is assigning to the parameter. Properties can be implemented with the => expression body symbol or auto-implemented. Here is an example of a class:

```
public class Person
{
   // auto-implemented properties for Name and Age
   public string Name { get; set; }
```

```
    public int Age { get; set; }
    // constructor
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}
```

To create an instance of a Person type, you use the constructor which is defined in the Person class.

```
Person person1 = new Person("Leo", 6);
Person person2 = person1;
```

In this example, assigning person1 to person2 causes both variables to refer to the same object. Generally, classes are used for more complex data that will be modified, while structs tend to be used for smaller data types with less complex behavior. Though structs and classes are very similar, there are some limitations of structs. Unlike classes, they do not support inheritance, which is one of the essential properties of object-oriented programming.

Inheritance:

C# and .NET only support single inheritance, and the terminology used is base class for the class that is inherited from and derived for classes that inherit from the base class. Inheritance is transitive, meaning that if there is a base class called Organism that is inherited by the class Mammal, and the class Human inherits from Mammal, Human will also inherit the members of Organism.

```
public class Human : Mammal
//Mammal and Organism fields, properties, methods and events are
//inherited
```

However, there is no inheritance of static constructions, instance constructors, or finalizers. Similarly, inheritance is restricted by access level (internal members are only available to derived classes in the same assembly as the base class). Members of a base class can be overridden by members in their derived classes, but methods in both classes must be declared with special keywords to be overridden (this will be covered in more detail in the encapsulation section). On the other hand, methods that are abstract in the base class, must be overridden in the

non-abstract, directly-inherited derived class. However, if the derived class is also abstract, it can inherit the members without implementing them.

Interfaces:

Interfaces are reference types that define a set of members that must be implemented by all structs and classes that implement said interface. Interfaces provide structs with some of the functionality provided by inheritance for classes and the functionality for multiple inheritance in C# (because a single class or struct can implement multiple interfaces). Interfaces are defined with the keyword *interface* followed by a valid identifier name (convention states that the name begins with a capital I). All classes and structs that implement the interface, must implement all of its members. Here is an example of an interface:

```
// T is a generic
public interface IShapes<T>
   {
       int Area();
   }
```

Here is an example of Triangle implementing IShapes

```
public class Triangle : IShapes<Triangle>
      {
      public int triBase;
      public int height;
      public Triangle(int triBase, int height)
      {
          this.triBase = triBase;
          this.height = height;
      }
      // implementing Area for Triangle
      public int Area()
      {
          return (this.triBase*this.height)/2;
      }
   }
```

Encapsulation:

Classes and structures have modifiers that support encapsulation. As seen in the examples above, class and method headers include an access level keyword that proceeds the class or

method name. The possible access modifiers are *public*: no restriction; *protected*: limited to containing class or derived classes; *internal*: limited to current assembly; *protected internal*: access is limited either to *protected* or *internal* restrictions; *private*: limited to containing type; and *private protected*: limited to containing or derived types within the current assembly. Classes and structs have *private* accessibility by default. Classes can have any of the five possible access modifiers while structs can only be *public*, *private*, or *internal*. Beyond accessibility levels, for derived classes to override members of their base class, the base class must have the keyword *virtual* and associated members in the derived class to be overridden, use the keyword *override* in the member header. Also, as mentioned above, classes and interfaces have properties which are a key part of encapsulation because of how they enable the user to read, write and/or compute private fields.

Here is an example of the method BakeTime for the base class Cake being overridden by the derived class Pancake.

```
public class Cake
   {
       public int mass;
       public Cake(int mass)
       {
           this.mass = mass;
       }
       public virtual int BakeTime()
       {
           return mass/5;
       }
   }
```

```
public class Pancake : Cake
   {
       public int size;
       public Pancake(int mass, int size)
: base(mass)
       {
           this.size = size;
       }
       public override int BakeTime()
       {
           return (mass/size)/10;
       }
   }
}
```

Object-oriented features:

With instances of classes and structs, C# creates objects, through interfaces and allowing members of one class to be pass to another C# enables inheritance, and with accessibility limits and overriding, C# features encapsulation. Thus, C# is an object-oriented language.

Exceptions:

All of the exceptions in C# are derived from *System.Exception* where *System* is the namespace and *Exception* is the name of the class. The *System.Exception* class and thus all exceptions have the read-only *Message* and *InnerException* properties. The *Message* property is a string that has a description of the reason for the exception, and the *InnerException* property either refers to the exception that caused the current exception or null meaning that the exception was not caused by another. This property comes into play when one thrown exception causes another exception to be thrown. In this second thrown exception, the *InnerException* will have a reference to the previous exception.

Exceptions can be thrown either by the compiler in specific circumstances when an operation cannot be completed normally, or an exception can be thrown explicitly with the keyword *throw*. In the first case, there is a group of exception classes called *Common Exception Classes* that shows the different exceptions that can be thrown conditionally in response to common errors like dividing by zero, trying to access an index that does not exist in an array, or when there is a discrepancy between the type of an element trying to be placed into an array and the type of the array. *Throw* must be used with an expression that has a value of type (or a derived type) of *System.Exception*, or the statement must be inside a *catch* block (this indicates that the exception is being re-thrown). The *catch* keyword along with *try* and *finally* are used in exception handling in C#. When a statement within a program throws an exception, the runtime will check to see if the statement that threw the exception is in a *try* block. If so, the *catch* blocks associated with the *try* block will be checked to see if the catch blocks' specified exceptions' types match that thrown. An important note is that an exception is considered to be the same type as any of the classes from which it derives. If a *try* block is not found around the statement that threw the exception, the runtime will check the calling method for a *try* statement, and will continue up the call stack until it finds a *catch* block of the same or compatible type as the exception. A *finally* block is used along with a *try* or *try/catch* block. The purpose of the *finally* block is to close any processes that were opened by the *try* block (like file streams or database connections). As mentioned in the C# documentation: "The statements of a *finally* block are always executed when control leaves a *try* statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a *break*, *continue*, *goto*, *or return* statement, or as a result of propagating an exception out of the *try* statement."

Programmers can create their own exception, and it must derive from *Exception* and it must have at least four constructors: one parameterless, one that sets *Message*, one that sets both *Message* and *InnerException*, and one that serializes the exception--serialization converts the object into bytes to it can be stored or transmitted to memory. Here is an example of a try/catch/finally block. This block will throw the exception because I do not have a file called myFile in the same directory as this program.

```
      System.IO.FileStream file = new FileStream(@"myFile.txt", FileMode.Open,
FileAccess.Write, FileShare.None);
          System.IO.FileInfo fileInfo = new
System.IO.FileInfo("c:\\MyDir\\MyFile.txt");
          try
```

```
        {
            file = fileInfo.OpenWrite();
            file.WriteByte(0x0);
        }
        catch(FileNotFoundException ex)
        {
            Console.WriteLine(ex);
        }
        finally
        {
            if(file != null)
            {
                file.Close();
                Console.WriteLine("File Stream was closed");
            }
        }
```

References

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/passing-reference-type-parameters

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/exceptions/how-to-execute-cleanup-code-using-finally

https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=net-5.0#remarks

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/throw#c-language-specification

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/exceptions

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/statements#the-throw-statement

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/exceptions#common-exception-classes

https://docs.microsoft.com/en-us/dotnet/standard/exceptions/how-to-use-finally-blocks

https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=net-5.0

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual

https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/inheritance

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties

https://docs.microsoft.com/en-us/ef/core/modeling/backing-field?tabs=data-annotations

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/accessibility-levels

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes

https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/intro-to-csharp/object-oriented-programming

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/objects

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum

https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/composite-data-types

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/type-testing-and-cast

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/

http://zetcode.com/lang/csharp/arrays/

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/objects

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/