DM8XX - Advanced Topics in Programming Languages

Spec#

Jakob Lykke Andersen

IMADA

May 4, 2009

The goal of Spec#

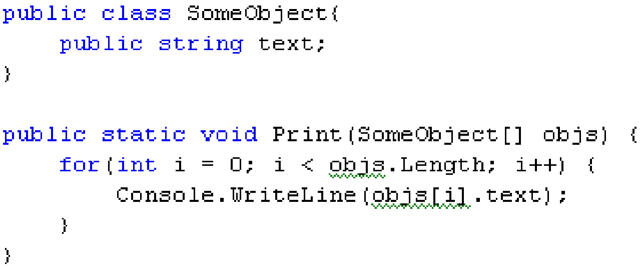
- Help us detect bugs.
- Help us prevent bugs.
- Incorporate some of the specifications into the code.

Spec# in general

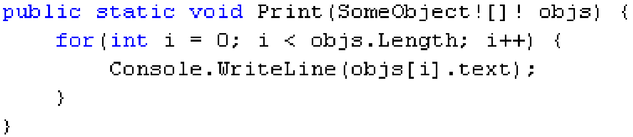
- An extension of C#.
- Runtime checks of contracts.
- Static verification of contracts.
- None-null types, pre- and post-conditions, invariants...

None-null types

- *string a*; $\equiv$ $a \in \{$“all strings”, *null*$\}$
- *string*! *a*; $\equiv$ $a \in \{$“all strings”$\}$



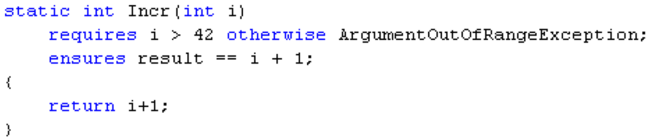
```
public class SomeObject{
    public string text;
}

public static void Print(SomeObject[] objs) {
    for(int i = 0; i < objs.Length; i++) {
        Console.WriteLine(objs[i].text);
    }
}
```



```
public static void Print(SomeObject![]! objs) {
    for(int i = 0; i < objs.Length; i++) {
        Console.WriteLine(objs[i].text);
    }
}
```

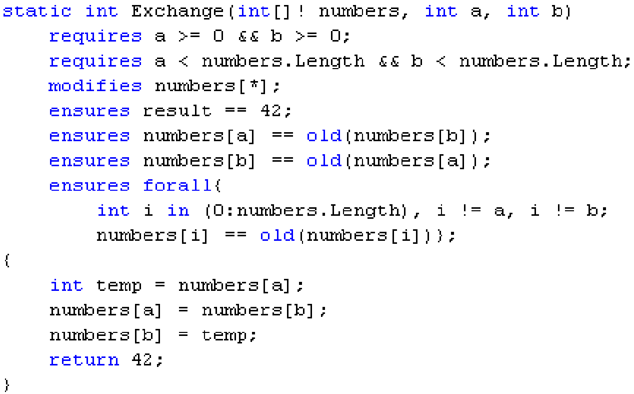
Pre- and post-conditions

- Part of the method signature.
- Pre-condition: *requires B*
- Post-condition: *ensures B*



```
static int Incr(int i)
    requires i > 42 otherwise ArgumentOutOfRangeException;
    ensures result == i + 1;
{
    return i+1;
}
```

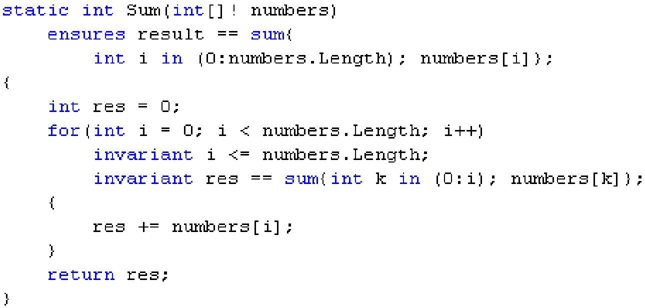
More stuff used in contracts



```
static int Exchange(int[]! numbers, int a, int b)
    requires a >= 0 && b >= 0;
    requires a < numbers.Length && b < numbers.Length;
    modifies numbers[*];
    ensures result == 42;
    ensures numbers[a] == old(numbers[b]);
    ensures numbers[b] == old(numbers[a]);
    ensures forall{
        int i in (0:numbers.Length), i != a, i != b;
        numbers[i] == old(numbers[i])};
{
    int temp = numbers[a];
    numbers[a] = numbers[b];
    numbers[b] = temp;
    return 42;
}
```

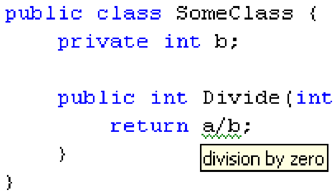
Loop invariants

- Used to help the verifier prove post-conditions.



```
static int Sum(int[]! numbers)
    ensures result == sum(
        int i in (0:numbers.Length); numbers[i]);
{
    int res = 0;
    for(int i = 0; i < numbers.Length; i++)
        invariant i <= numbers.Length;
        invariant res == sum(int k in (0:i); numbers[k]);
    {
        res += numbers[i];
    }
    return res;
}
```

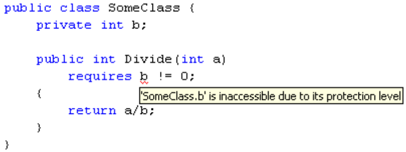
Object invariants



```
public class SomeClass {
    private int b;

    public int Divide(int a) {
        return a/b;
    }
}
```

division by zero

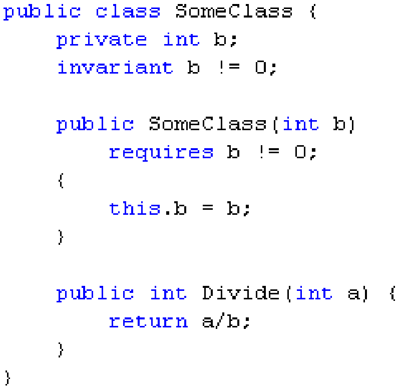
Object invariants



```
public class SomeClass {
    private int b;

    public int Divide(int a)
        requires b != 0;
    {
        return a/b;
    }
}
```

'SomeClass.b' is inaccessible due to its protection level

Object invariants



```
public class SomeClass {
    private int b;
    invariant b != 0;

    public SomeClass(int b)
        requires b != 0;
    {
        this.b = b;
    }

    public int Divide(int a) {
        return a/b;
    }
}
```

Object states

- Object A contains a reference to object B.
- An invariant in object A constrains object B ($invariant\ b.aInt > 0$).

Object states

- Object A contains a reference to object B.
- An invariant in object A constrains object B ($invariant\ b.aInt > 0$).
- What if object C also contains a reference to object B and have $invariant\ b.aInt <= 0$?

Object states

- Object A contains a reference to object B.
- An invariant in object A constrains object B ($invariant\ b.aInt > 0$).
- What if object C also contains a reference to object B and have $invariant\ b.aInt <= 0$?

- Mutable objects.
- Ownership of objects.
- Exposing and packing objects.

Project

A subset of the following:

- Implement some sorting algoritm with invariants and conditions, so the correctness can be proved by the verifier.
- Implement a small library of methods for manipulating strings.
- Implement a not too complex example with object states.