CSE 3902: C# Primer

Justin Holewinski

The Ohio State University

Disclaimer

The following slides are meant to be a very high-level overview of C#; just enough to get you started with Sprint 0!

It is *your* responsibility to read the required chapters from "C# in a Nutshell" and consult additional sources as necessary to get a comfortable understanding of the language!

Variables

Variables are declared starting with their type, or `var`.

```
// Explicit type
int num = 42;

// Implicit (compiler-inferred) type
var num = 420;
```

Basic Types:

- Integral: `int`, `long`, `short`
- Floating-Point: `float`, `double`
- Boolean: `bool`
- Dynamic Strings: `string`

Visibility

Class declarations use *visibility* modifiers to determine who can access classes and their components.

- `public`: Visible everywhere
- `protected`: Visible to derived classes
- `private`: Visible only to itself
- `internal`: Visible everywhere *in the same assembly*

Applies to classes, structs, methods, fields, properties, and more.

Conditional Statements

What is the output of the following?

```
int i = 1 / 2;
if (i > 0)
{
    Console.WriteLine("i is greater than 0");
}
else
{
    Console.WriteLine("i is NOT greater than 0");
}
```

Loops

Loops with explicit conditions

```
for (int i = 0; i < 5; ++i)
{
    Console.WriteLine($"i: {i}");
}
```

```
while (mylist.Length != 0)
{
    /* Process mylist */
}
```

Easy iteration over containers

```
List<string> messages = /* ... */;

foreach (string msg in messages) {
    Console.WriteLine(msg);
}
```

Classes/Structs

Classes and structs are aggregate types that can contain:

- Methods
- Constructors
- Fields
- Properties
- Delegates
- And others ...

```
public class Player
{
    private Vector2 position;

    public Player()
    {
        position = Vector2.Zero;
    }
}
```

Methods

```
public class Player
{
    // Constructor
    public Player() {}

    // Overloads
    public void SetPosition(Vector2 newPos) { /*...*/ }
    public void SetPosition(float x, float y) { /*...*/ }

    // Static Methods
    public static int GetMaxHealth() { /*...*/ }
}
```

Fields

```
public class Player
{
    public int health = 10;

    private ISprite sprite = /* ??? */;

    public static int maxHealth = 100;
}
```

Properties

```
public class Player
{
    public int Health
    {
        get
        {
            return health;
        }

        set
        {
            health = Math.Min(value, MaxHealth);
        }
    }
}
```

Classes

Classes are *reference* types in C#

```
public class Player
{
    public int x = 0;
    public void Move() { x += 1; }
}

// "me" is a reference to the object created by "new Player()"
Player me = new Player();

// "alsoMe" is a reference to the same object
Player alsoMe = me;

// Any side effects of "Move()" will be visible in "me" and "alsoMe"
me.Move();

Debug.Assert(me.x == 1);
Debug.Assert(alsoMe.x == 1);
```

Structs

Structs are *value* types in C#

```
public struct Player
{
    public int x = 0;
    public void Move() { x += 1; }
}

// "me" is the object created by "new Player()"
Player me = new Player();

// "other" is a unique object created by copying "me"
Player other = me;

// Any side effects of "Move()" will be visible in "me" but not "other"
me.Move();

Debug.Assert(me.x == 1);
Debug.Assert(other.x == 0);
```

Parameter Passing

The reference vs value distinction is important for parameter passing

```
public class Player
{
    public Point Position = new Point(0, 0);
}

public void MovePlayer(Player player)
{
    player.Position.X += 10;
}

Player me = new Player();

// "me" will be passed by reference to MovePlayer.
// Changes will be visible in the caller.
MovePlayer(me);

Debug.Assert(me.Position.X == 10);
```

Parameter Passing

```
public struct Position
{
    public int X = 0;
}
public void ChangePosition(Position pos)
{
    pos.X += 10;
}

Position here = new Position();

// "here" will be passed by value (by copy) to ChangePosition.
// Changes will NOT be visible in the caller.
ChangePosition(here);

Debug.Assert(here.X == 0);
```

Parameter Passing

Use `ref` to enable pass-by-reference semantics for `struct` types.

```
public void ChangePosition(ref Position pos)
{
    pos.X += 10;
}

Position here = new Position();

// "here" will be passed by reference to ChangePosition.
// Changes WILL be visible in the caller.
ChangePosition(ref here);

Debug.Assert(here.X == 10);
```