**LEARN C#**

**The Object of Your Affection**

Your friend is building a new match-making service: The Object of Your Affection or OOYA for short (don’t worry, you still have time to convince them to change the name).



With your new understanding of C# objects and classes, your friend thought you could build a pretty well-organized system of dating profiles.

Your first step? Build a Profile class that allows users to generate profile objects.

The Profile should store the following information:

* User’s name
* User’s age
* User’s city
* User’s country
* User’s pronouns
* User’s hobbies

And this is how users should be able to interact with their own profiles:

* Create a new profile with some information
* Add hobbies
* View profile

Let’s get started!

### The Fields of a Classy Profile

**1.**

Tab over to **Profile.cs** and set up the skeleton of the Profile class.

using System;

namespace DatingProfile

{

}

**2.**

Add the following fields to Profile:

* a string name
* an int age
* a string city
* a string country
* a string pronouns
* a string[] hobbies

We could implement these as properties, but we’ll use fields. Properties are used to:

* validate values
* control external access

Later in this project you’ll see how we achieve the same result with methods.

3. Tab over to **Program.cs**. In Main():

* Instantiate a new Profile object called sam. (Your friend Sam is looking for love.)
* Try to give sam a name: "Sam Drakkila".
* Then try to run the code using dotnet run.

4. Yikes, what was that error message all about? Something to do with Profile.name being private?

Oh that’s right! All the members in a class (including name) are automatically set to private.

To make this more clear for ourselves and others, make the access level explicit: Add private to all the fields you created in Profile.

**5.**

Users should be able to add their profile information in a constructor.

Below the fields, declare a constructor for Profile that allows you to set:

* name
* age
* city
* country
* pronouns (give this a default value of "they/them" just in case it’s ever left blank)

Define the constructor in **Profile.cs** and set the fields to the values passed in. Make sure to also set hobbies to an empty array of strings.

Use this to differentiate parameters from instance fields. For example, this will work:

public Profile(int population)

{ this.population = population; }

But this won’t:

public Profile(int population)

{ population = population; }

**6.**

Time to test your code out!

If you assigned sam a name in **Program.cs** before, remove that line. Where sam is constructed, pass in the following information:

* a name of "Sam Drakkila"
* an age of 30
* city and country of "New York" and "USA"
* pronouns of "he/him"

Then run your code.

**7.**

Nice work! But how can you access profile information once it’s been added?

We could use properties, but we’d like users to see all of the information in a single, formatted string. Time to add a ViewProfile() method.

In **Profile.cs** define a ViewProfile() method under the constructor. It should have:

* public access
* a return type of string
* no parameters

It should return a string containing all of the profile’s information.

8. In Main(), test out the new method on sam and print out the result.

**9.**

You still need to give users a way to set hobbies.

In **Profile.cs**, declare a new method SetHobbies() with:

* public access
* no return value
* a string[] parameter named hobbies

In the method body, set the field this.hobbies equal to the hobbies argument.

**10.**

Great! Go back into ViewProfile() and modify the method so that you display a profile’s hobbies.

(Remember, you can loop through this.hobbies to access each element.)

**11.**

Before you show this all to your friend, be sure to test your work.

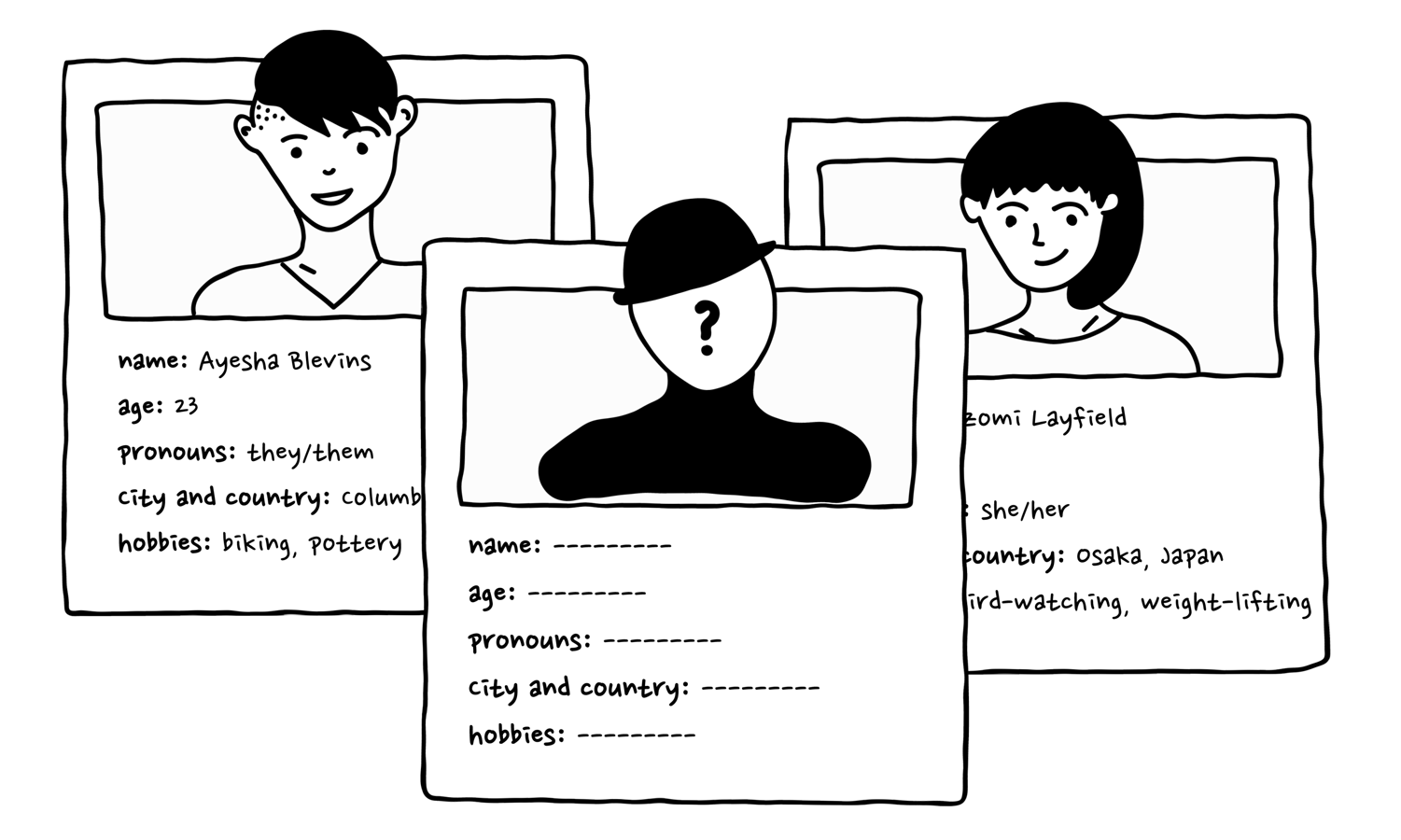
Above where you print sam‘s profile information out, add some hobbies to sam:

* “listening to audiobooks and podcasts”
* “playing rec sports like bowling and kickball”
* “writing a speculative fiction novel”
* “reading advice columns”

Now run your code!

**12.**

Your friend is super impressed with the Profile class you’ve created!



Here are a few suggestions to make the Profile class even better:

* If you call ViewProfile() before calling SetHobbies(), you’ll get an error because the hobbies field isn’t set to any value. Fix the class so that you can call ViewProfile() without adding hobbies.
* Convert the fields into private properties and add validation. For example, users must be at least 18 years of age.
* Some users may create a profile with just a name and age. Use optional parameters or create a constructor overload to handle those issues.

**Avoiding errors** — To avoid the errors in ViewProfile():

1. In the constructor, give hobbies a default value of an array of length 0.
2. In ViewProfile() only append hobbies in if hobbies.Length is greater than 0.

**Adding properties** — A private property is defined like:

private string Name

{

get { /\* omitted \*/ }

set { /\* omitted \*/ }

}

**Partial profiles I** — Optional parameters must have a default value like:

public Recipe(string name = "n/a")

**Partials profiles II** — When overloading constructors, avoid code duplication by using : this(), like:

public Recipe() : this("n/a")