Chapter 31

Access modifiers

This chapter finally explains why we have to add `public` in front of all of our struct variables. If you remember, without the `public`, you can't use them, which seems like a crazy rule.

The actual rule is that everything is either `public` or `private`, and if you don't say, they're automatically `private`. But that doesn't explain why we have `private` in the first place.

This chapter is about how `private` variables work, why we'd ever want to use them, and a few similar tricks.

31.1 How `private` variables work

`private` really means that people *outside* of the class aren't allowed to use it. Member functions in the class are allowed to see all the fields. You can think of `private` as employees-only.

Here's an example class with two `private` variables:

```
class NumHider {
  private int n;
  int m; // m is also private, since we didn't write anything in front

  public void Set(int v1, int v2) {
    n=v1; m=v2; // legal. private doesn't apply to us
  }

  public getN() { return n; }
  public getM() { return m; }
}
```

This is not a good class, but it shows how it's possible to use indirectly use private variables through member functions:

```
NumHider nh=new NumHider();
nh.n=5; nh.m=3; // ERRORS
nh.Set(5,3); // legal, sets them how we wanted
if(nh.n<10) // ERROR
if(nh.getN()<10) // legal
```

Again, this is just about the mechanics: it's possible to use variables you can never touch, by going through functions. Later we'll have an example where that's useful.

We can also have `private` member functions. We can't call them, but they aren't useless since other functions can. Here `fixNeg` is private, but `Set` uses it to fix the inputs:

```
class NumHider {
  public void Set(int v1, int v2) {
    n=v1; m=v2;
    fixNeg(); // <- we can call this, user can't
  }

  private void fixNeg() { if(n<0) n=0; if(m<0) m=0; }
}
```

The pop-up hides `private`'s from us, which makes them less annoying. Typing `nh`-dot will only show Set, getN and getM.

31.2 Classes as new types

For our classes so far, we started out knowing the variables. For example, `FullName` started with us wanting one string for first name and another for last name. Turning it into a class was just a nice way to group them. And the member functions are just helpful shortcuts. There's nothing wrong with that, but `private` wasn't made for those.

There's a different way to use classes. Sometimes we start with something we want to make. The variables are just a way to make it happen, and we really don't care about them. We only care about using the member functions.

`private` was invented for this idea – we can use it to say "don't think about the variables – the functions do what you need."

Here's an example. Suppose we want a random number roller that can remember the range, and never rolls the same number twice in a row. We'll start with this plan, describing the functions without writing them yet:

```
class RollerNoRpt {
  public void SetRange(int min, int max);
  public int Next(); // roll number, with no repeat
}
```

Just those functions are all we need for a useful class. We can say: `RollerNoRpt r1; r1.SetRange(5,10);`, and then use `int n=r1.Next();` to get 5-10's with no repeats.

The variables and the function to set the range are probably obvious:

```
class RollerNoRpt {
  private int min, maxp1; // if range is 1-6, we save (1,7)
  private int prevNum; // to avoid same num twice

  public void SetRange(int low, int high) {
    if(low>high) { int tmp=low; low=high; high=tmp; }
    min=low; maxp1=high+1;
    prevNum=low-1; // at the start, there is no previous roll
  }
```

So far, by hiding the variables we've made it impossible to get them backwards. And we're handling the funny rule about `Random.Range` not being able to roll the max without you needing to worry about it.

`Next` will use the "roll until it's not a repeat" loop from before, using those three variables and the helper function `simpleRoll`:

```
  public int Next() {
    if(min+1===maxp1) return min; // if range is 4-4, answer is always 4
    int nn = simpleRoll();
    while(nn==prevNum) nn=simpleRoll();
    prevNum=nn;
    return nn;
  }

    // used only by Next:
    private int simpleRoll() { return Random.Range(min,maxp1); }
}
```

Again, the point is we can use `SetRange` and `Next` without needing to know how the member variables work. We never see `prevNum` and we like it that way. We can't use or even see helper-function `simpleRoll` and that's good, since it can give a repeat.

Another example, a little simpler: we want a class to check whether an x,y is inside a rectangular area (maybe for a 2D game.) Areas will be set using

width/height, giving either the lower-left corner or the center. Like this (only showing the function headings we want):

```
class Rectangle {
  public void SetFromCent(float xCent, float yCent,
                  float wide, float high)
  public void SetFromLL(float xLLcorner, float yLLcorner,
                  float wide, float high)

  public bool isIn(float x, float y);
}
```

It's probably really storing them one way and converting the other, but we don't care which, as long as it works.

My plan is to store the real positions of all four sides: if we use SetFromLL with x=50 and 8 wide, we'll save 50 and 58. That's neither of the ways we thought, and that's the point. We don't care how you use the variables, as long as it works.

Everything written out:

```
class Rectangle {
  float xLeft, xRight, yLow, yHigh; // private

  public void SetFromCent(float xCent,float yCent,float wide,float high) {
    float halfWide=wide/2, halfHigh=high/2;
    xLeft=xCent-halfWide; xRight=xCent+halfWide;
    yLow=yCent-halfHigh; yHigh=yCent+halfHigh;
  }

  public void SetFromLL(float xLLcorner,float yLLcorner,float wide,float high) {
    xLeft=xLLcorner; yLow=yLLcorner;  // copy these
    xRight=xLeft+wide; yHigh=yLow+high; // compute these
  }

  public bool isIn(float x, float y) {
    return x>=xLeft && x<=xRight && y>=yLow && y<=yHigh;
  }
}
```

Both ways of setting it convert to the real-corners method, and `isIn` does a simple in-between compare on `x` and `y`. But again, we don't need to know this to use it.

This next example takes that idea a little further, making a 0-255 color class (recall many artists prefer 0-255, but the Unity Color class uses 0-1). Users never need to know the real way Unity stores colors:

Here's the outline (it's a little fakey, but it's short and shows the idea):

```
class Color255 {
  public void Set(int red, int green, int blue) // should be 0-255

  public void applyTo(Transform t) // apply the color to this transform
}
```

We can make bright orange with `c1.set(255,128,32);`. Then use `c1.applyTo(cat.transform);` to paint the cat orange. We never see a 0-1 value.

To store them, we'll use a real Unity `Color` variable, and a private helper conversion function:

```
class Color255 {
  private Color c; // a real 0-1 Unity color

  // this expects 0-255 values (which is why they are ints)
  public void set(int red, int green, int blue) {
    c.r=toF(red); c.g=toF(green); c.b=toF(blue);
    c.a=1; // 1=not transparent
  }

  private float toF(int n) { return n/255.0f; } // convert 0-255 into 0-1

  public void applyTo(Transform t) {
    t.GetComponent<Renderer>().material.color=c;
  }
}
```

`toF` is a typical `private` helper function. It's used by `Set` to convert each 0-255 into the real 0-1. We don't want the user to see it, since the entire point of this class is hiding how color is 0-1.

The class only has one variable, which is fine for classes like this. We often call those **wrappers** – the class wraps around the real `Color` struct, making it easier for artists to use.

31.3 Interface/Implementation idea

There are some concepts and terms that go with "use a class to make an idea." They aren't really technical terms, but people use them a lot, so it's nice to hear them, and some can be helpful.

We think of a class as divided into **Interface** and **Implementation**. Interface is the normal English meaning – the part you interact with. For a class, it's the `public` functions. Implementation is how it actually works.

Anyone using the class is a **client**. Clients use the Interface, and don't care about the Implementation.

This is really the same trick we've been using with functions – we only need to know the inputs and output, not exactly how it works. For a class, the inputs and outputs are what you can do with all the public functions.

Another word for the idea is **Information Hiding**. A slightly newer term is **Encapsulation** – the private implementation is encapsulated away from you.

Sometimes we call a class like this an **Object**. That's where the term Object Oriented Programming comes from.

Object is another way of saying we're absolutely not thinking of it as pile of variables.

Here's a list of some regular times people like to use `private` to break into Interface vs. Implementation:

- The class was written by an expert. You don't know how it works inside, and don't want to know. You're glad all that stuff is hidden with `private`.
- You wrote the class, but by tomorrow you'll forget what the variables stand for. You know if `pxPos` was public you'd be dumb enough to assign it 10 without adjusting for the virtual thing-a-majig. So you force yourself to use `setPx(10);`.
- You've got Interns and Jr. Programmers who don't understand that some variables have tricky rules, or go together with other variables. They've never seen the Interface/Implementation idea. Making only the Interface `public` automatically makes them do it the right way, without having to explain anything extra to them.
- We might want to make changes later. This is one of the big ideas. If we keep most things `private`, so no on one else can touch them, we can rewrite the class (to be better, or work with a different device ...) and everything will still work. This is the same idea as rewriting the inside of a function.

Here's an example of not wanting to touch the variables since they have a relationship you might mess up. I want a score and a screen label displaying it. The score shouldn't go below 0, and the font for the label should adjust based on the number of digits.

We'll combine the score variable and its screen label into a class, with one scoring-change function to do it all at once. Here's the part we care about:

```
class Score {
  // you must supply a textbox when you create it:
  public Score(GameObject textBox); // constructor
  public void ChangeBy(int amt)
  public void Set(int value)
```

```
  public int Val() // get the score
}
```

That seems like it should be plenty to use it. Looking up the score using `s.Val()` is a little bit of a pain, but a small sacrifice.

Making it work is long-ish, but nothing too complicated:

```
class Score {
  private int s; // the score
  private GameObject label; // holds Text display

  // constructor requires you give it pointer to the label:
  public Score(GameObject textBox) { label=textBox; _setScore(0); }

  public int Val() { return s; }

  public void ChangeBy(int amt) { _setScore(s+amt); }

  public void Set(int value) { _setScore(value); }

  private _setScore(int newValue) {
    if(newValue<0) newValue=0; // not less than 0
    s=newValue;
    // write in label, with font size for # of digits:
    label.GetComponent<UnityEngine.UI.Text>().text = ""+s;
    int fSz;
    if(s<=9) fSz=72; // 1 digit
    else if(s<=99) fSz=58; // 2 digits
    else fSz=44; // 3+ digits is smallest font size
    label.GetComponent<UnityEngine.UI.Text>().fontSize = fSz;
  }
}
```

The idea is, we could leave `int s` and the textBox as public. Some things would be a little easier. But somewhere in the program we'd forget to check for below 0, or we'd change the score and forget to update the textBox.

It's better to have to use `Set` or `ChangeBy` so we can never have wrong values.

The start of how we'd use it:

```
// create a GUI Text, position it and drag it here
public GameObject scoreTextBox;

Score player1Score; // <- the class we just wrote

void Start() {
```

```
  player1Score = new Score(scoreTextBox);
   ...
  player1Score.Set(10); // level start with 10 points
  // also auto-displays on the screen
   ...
  if(player1Score.Val()>100) print("win");
}
```

31.3.1 Rewriting classes

I wanted to give a few examples of re-doing member variables, since it helps understand the Interface/Implementation idea.

The `Rectangle` class could store things using center plus size. It would look like this:

```
class Rectangle {
  private Vector2 center; // has dot-x and dot-y
  private Vector2 halfSize; // how far it goes in both directions x/y

  public void SetFromCent(float xCent, float yCent, float wide, float high) {
    center=new Vector2(xCent, yCent);
    halfSize=new Vector2(wide/2, high/2);
  }

  public void SetFromLL(float xLLcorner, float yLLcorner, float wide, float high) {
    halfSize=new Vector2(wide/2, high/2);
    center=new Vector2(xLLcorner, yLLcorner)+halfSize;
  }

  public bool isIn(float x, float y) {
    return x>=center.x-halfSize.x && x<=center.x+halfSize.x &&
        y>=center.y-halfSize.y && y<=center.y+halfSize.y;
  }
}
```

There's no advantage to doing it this way - maybe it was just the first way we thought of. But the point is how having private variables makes it possible. No one was allowed to use the old variables, so this can't break anything; and the functions people use give the same results.

The random roller can have a much better rewrite. Suppose we decided it needs to hit every number once before repeating (random 1-6 could be 4,3,6,1,5,2.)

I showed the trick for doing that earlier – make a list of them all, shuffle it, and use them in order. Here's a rewrite doing that. When we set the range, it fills array `N` with the numbers. `Next()` uses `cur` to go through the list, and reshuffles when it uses them all:

```
class RandNoRpt {
  private List<int> N; // all numbers we can roll, mixed up
  private int cur; // index of next number

  public void SetRange(int low, int high ) {
    if(low>high) { int tmp=low; low=high; high=tmp; }
    int rangeSize = high-low+1;
    // fill N with every number we can roll:
    N = new List<int>();
    for(int i=0;i<rangeSize;i++) N.Add(low+i);
    _shuffle();
  }

  private _shuffle() {
    cur=0; // restart at first item in new shuffle
    // this is copied from List chapter:
    for(int i=0; i<N.Count; i++) {
      int ii = Random.Range(0,N.Count);
      int tmp=N[i]; N[i]=N[ii]; N[ii]=tmp;
    }
  }

  int Next() {
    if(cur>=N.Count) {
      int lastNum=N[N.Count-1]; // this can't be first in new one
      _shuffle();
      if(N[0]==lastNum) { int tmp=N[0]; N[0]=N[1]; N[1]=tmp; }
    }
    int ans=N[cur];
    cur++;
    return ans;
  }
}
```

A cool thing here is we aren't even storing the low and high numbers anymore. The way this really works is a lot different than the old way, and nothing like most people might guess. But the Interface (`SetRange` and `Next()`) works the same.

31.3.2 Accessors

Classes often let you use a variable through a pair of functions. For example the Score class is letting you use `s`, but with `Val()` to read it and `Set` to change it.

We call pairs like these Accessors, since they're the only way to access a private variable. Often the names are like in my very first example: `setN` and

`getN`, so they're sometimes called getter/setter pairs.

We often use this trick to create a fake variable. Here `Rectangle` has a getters/setter pair for the lower-left corner, the center and the width:

```
class Rectangle {
  private Vector2 center; // For real the class saves ...
  private Vector2 halfSize; // ... the center

  public Vector2 getLowerLeft() { return center-halfSize; }
  public void setLowerLeft(int x, int y) { center=new Vector2(x,y)+halfSize; }

  public Vector2 getCenter() { return center; }
  public void setCenter(float x, float y) { center=new Vector2(x,y); }

  public float getWidth() { return halfSize.x*2; }
  public float setWidth(float wide) { halfSize.x=wide/2; }
```

Those three pairs make it feel like we have those 3 variables. Currently two are fake and `center` is an actual variable. But a rewrite might change that. If we keep our interface as function pairs, we don't need to worry about how it "really" works.

31.3.3 get/set

C# has a short-cut for using a getter/setter accessor pair. On the one hand it's cutesy, does nothing special and is only in `C#`. But on the other hand Unity uses it in a few places and it's an example of an interface. So it's worth a look, but don't feel like you need to know it.

Here's width from `Rectangle`, rewritten with the shortcut. It creates one function named `width` which can be used both ways:

```
class Rectangle {
  ...
  public float width {
    get { return halfSize.x*2; } // same code as getWidth()
    set { halfSize.x=value/2; } // same code as setWidth
  }
  ...
}
```

The shortcut is we can write `w=r1.width;` and have it automatically run the `get` part, or `t1.width=6;` and have it automatically run the `set`, which would make halfsize's x be 3. As you might guess, set automatically has 1 input, always magically named `value`. In this case, it's 6.

Before, `getWidth` and `setWidth` acted like a fake width variable. This shortcut is the same plan, but now it makes an actual fake `width` variable.

You can tell this trick is being used when you see {`get; set;`} in the dropdown.

Often this trick is abused only to get rid of parens in a 0-input function. Write the `get` part with no `set`. Here's a real example from Unity getting the brightness of a color:

```
class Color {
  public float r, g, b;

  public float grayScale {
     get { return (r+g+b)/3.0f; }
  }
}
```

You'd call it like `float f = c1.grayScale;`, but it really is a function, averaging the 3 color values. The tooltip show you { `get;` }, so you know it's not a variable.

`w.Length` and `L.Count` also both use that trick. They're really functions, like `greyScale`.

Most languages have these *idioms* where everyone writes things in a certain odd way. Using only `get` may have started as a secret handshake – `C#` is one of the only languages with this fake-variable shortcut, so it proves you studied the manual. But now it's just something everyone does, just because (I never add this shortcut to my classes, since `C#` is an alternate language, to me.)