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 indirectly use private variables through member functions:

```
NumHider nh=new NumHider();
nh.n=5; nh.m=3; // ERRORS - private
nh.Set(5,3); // legal, the function sets m and n
if(nh.n<10) // ERROR
if(nh.getN()<10) // legal
```

We can never touch `n` and `m`, but we can change and read them 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. The member functions are merely 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. The interface (the public functions) can be like this:

```
class RollerNoRpt { // not done: names of public funcs only
  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.

Now that we like those functions as the interface, we have to write them, along with variables to make them work. We'll save the range, pre-adding 1 to the max, to account for how Random works. We'll have a variable saving the previous number:

```
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: If you call `SetRange(7,2)` by mistake, it fixes it.

`Next()` will use the "roll until it's not a repeat" loop from before, and the private 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); }
}
```

That proves that we can do it, and it's some fun coding. But the important thing is how it's completely hidden. Users type `r1`-dot, see `SetRange` and `Next`, and that's all they need.

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).

We'll give options to set an area starting from either the lower-left corner, or from the center. For example `r1.SetFromCent(6,6,8,4)` puts the rectangle at (6,6), 8 wide and 4 high. Or `r1.SetFromLL(0,0,5,5)` puts the lower-left corner at (0,0), 5 wide and high. Some situations will prefer one or the other.

No matter how we set it, `r1.isIn(p1)` checks whether `p1` is inside. We don't care how it checks, or what variables it uses, as long as it works.

The plan is to store the real positions of all four corners. If we use SetFromCent with x=50 and 8 wide, we'll save 46 and 54 for x min and max:

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

  public void SetFromCent(float xc, float yc, float wide, float high) {
    float halfWide=wide/2, halfHigh=high/2; // go 1/2-way in
    xLeft=xc-halfWide; xRight=xc+halfWide; // each direction
    yLow=yc-halfHigh; yHigh=yc+halfHigh;
  }

  public void SetFromLL(float xLLc, float yLLc, float wide, float high) {
    xLeft=xLLc; yLow=yLLc;  // 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;
  }
}
```

That's all fun code. I especially like how `isIn` is merely two in-between compares this way. But the main thing is how it's hidden. Users only care about `SetFromCent`, `SetFromLL`, and `isIn`. They can't even see the variables.

This next example takes that idea a little further, making a 0-255 color class which purposely hides how Unity Colors use 0-1 (paint programs all use 0-255 for color levels. It's the range all artists know).

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

```
class Color255 { // not done: names of public functions only
  public void Set(int red, int green, int blue) // use 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.

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. If you don't make them private you'll set them directly instead of using the function ("I wrote this class. I know how things work!") and screw it up.
- 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 rewrite it. If we think of a better way to roll dice we can rip out the private variables, and rewrite the guts using new, better ones. As long as `SetRange` and `Next` work the same, it's safe.

Sometimes it's the case that we know exactly what's inside of a class, but we still want to force ourselves to use the interface. Suppose we have a score that should also be displayed in a textBox. We could make a simple class to group them:

```
class Score {
  public int s;
  public GameObject label; // set this to previously created textBox
}
```

It's the user's job to remember to change the label when they change `s`. But at the very least we can make a helpful function for that:

```
class Score {
  public int s;
  public GameObject label;

  public void updateLabel() { // useful label-setting function
    label.GetComponent<Text>().text="Score: "+s;
  }
}
```

Users can add to the score with: `s1.s++;`, then run `s1.updateLabel();` to display it. But someone, somwwhere, will forget to run `updateLabel()`. The score will change, but we won't see it. To avoid that we need one command to do both things at once, which you have to use:

```
class Score {
  private int s; // <- users can't write s1.s++; any more
  private GameObject label; // <- they won't need to use this

  private void updateLabel() { // <- changed to private
    label.GetComponent<Text>().text="Score: "+s1;
  }

  // use these to change the score and auto-fix the label:
  public void ChangeBy(int amt) { s+=amt; updateLabel(); }
  public void Set(int value) { s=value; updateLabel(); }

  public int score() { return s; } // need this to read the score
}
```

To be nice, it's two commands. `s1.Set(5);` seems fine, at first, but `s1.ChangeBy(1);` is an easy way to add 1. The key is that both update the label. You can't get them out-of-synch.

Making `s` private caused a problem: we can't read the score any more. That's what `s1.score()` is for. We'll have to write ugly things like `if(s1.score()>21)`. But always having the score and label change together will be worth it.

The best way to assign the label is a bonus trick. We need to give it a textBox once, at the start, and won't change it again. A neat way to allow that is by putting it in the only constructor:

```
class Score {
  private int s; // <- users can't write s1.s+; any more
  private GameObject label;

  public Score(GameObject theLabel) {
    label=theLabel; // save the textBox link
    s=0; // we may as well do other stuff to make
    updateLabel(); // it look nice
  }
```

It's a neat trick since we need to use `new` anyway, and it's impossible to forget to supply the label:

```
public GameObject scoreTextBox; // dragged into Inspector slot
Score s1; // the class we just wrote
```

```
void Start() {
  s1 = new Score(scoreTextBox);
  // s1 is created, and has its label
```

This is one typical use of `private`. It's a way to say "I know that you know how to use the variables, and it would probably be fine. But, to be safe, please, please always call the functions instead".

31.3.1 Rewriting classes

A fun way to see the Interface/Implementation idea is to rewrite how a class works inside, without changing what it does.

The `Rectangle` class could be rewritten to secretly store the center and width/height (the old version stored the positions of the corners). This isn't super-exciting:

```
class Rectangle {
  private Vector2 center; // saving the center
  private Vector2 halfSize; // how far it goes in both directions x/y

  public void SetFromCent(float xc, float yc, float wide, float high) {
    center=new Vector2(xc, yc); // save the center they gave us,
    halfSize=new Vector2(wide/2, high/2); // and 1/2 the size. Easy
  }

  public void SetFromLL(float xLLc, float yLLc, float wide, float high) {
    halfSize=new Vector2(wide/2, high/2); // use the size and LLcorner
    center=new Vector2(xLLc, yLLc)+halfSize; // to find the center
  }

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

I think the old way is better. But the point is that both ways – completely different variables – look and act exactly the same to users. That's pretty cool.

The random roller can have a much better rewrite. We probably want it to hit every number before repeating: for 1-6 it could be 4,3,6,1,5,2, then go again.

We can use the shuffle trick to do that. It starts with a hidden list of the scrambled numbers, walks through them as we ask with `Next()`, and rescrambles when we run out:

```
class RandNoRpt {
```

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

  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() { // shuffles the list N
    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) { // past end, reset:
      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]; // shuffle or not, return next # in the list
    cur++;
    return ans;
  }
}
```

The pop-up still shows only `SetRange` and `Next()`. Users know nothing about a list and shuffling, which is great. Those are details they don't need to think about.

31.3.2 Accessors

Hiding a variable to make you use functions is so common that we have a name for the functions: we call them *accessors* or a "getter/setter" pair.

The Score class is a typical example. Everyone knows the class has an integer score variable. We're not trying to keep you from thinking about it. We merely want you use use `s1.Set(4)` to change it. A not-so-good side-effect is having to use `s1.score()` to read it. `Set(n)` and `score()` are its accessors – one to

"get" it, the other to Set it.

It's not great, but we can indirectly use the two functions to do anything = could do before:

```
// using the getter/setters:
s1.ChangeBy(1);
if(s1.score()>10) s1.Set(10);

// how we think of it:
s1.s++;
if(s1.s>10) s1.s=10;
```

We almost always use them to add extra rules about changing a variable – such as also updating the label. A common example is not allowing a variable to be negative. We put a check for that in the Set function:

```
class nUser {
  private int n;
  public int getN() { return n; }
  public void setN(int newVal) {
    if(newVal<0) newVal=0; // copies input into n,
    n=newVal;  // but fixes negative values
  }
}
```

It's a little awkward. `n1.setN(5)` replaces `n1.n=5`. Not too bad. But `n1.n--;` turns into `n1.SetN(n1.getN()-1);`. Ick, but it ensures `n` can't be negative.

There's one more trick getter/setter pairs allow. They can create fake variables. For example, this class stores the distance in feet, but can pretend it uses inches:

```
class FakeInchClass {
  public float feet;

  public float inches() { return feet*12; }
  public void setInches(float newInches) { feet=newInches/12; }
}
```

If you prefer to use this class with inches, you can. Use `n1.setInches(18)` and `n1.inches()` gives you back 18. For real it sets `feet` to 1.5, but so what? And if some other wise-guy sets `n1.feet=5` its fine – `n1.inches()` reads back 60. It works great.

We can do something similar with Rectangle. We already have a function that sets it using the lower-left corner. We can rename it and add one to compute the lower-left corner:

```
class Rectangle {
  public Vector2 center; // let people set this directly
  private Vector2 halfSize;

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

Combined, they allow us to examine and position Rectangles as if lower-left was the actual variable.

31.3.3 Special get/set

The getter/setter idea is so popular that C# has a super-special shortcut. The idea is this: getters and setters are substitutes for `x=n1.n;` (getter) and `n1.n=3;` (setter). What if we change `=`'s so they automatically run the appropriate get or set?

You still have to write both functions, in the class. But you don't have to use that messy syntax to call them. Even something tricky like `n1.n++;` will automatically run your getter for `n`, add 1, then run the setter.

In this legal C# code, `get`, `set` and `value` are magic words, specially for this trick. `n` can't be negative:

```
class NcantBeNegative {
  private int nn;

  public int n { // the fake variable is named n
    get { return nn; } // called for int x = n1.n;
    set { if(value<0) nn=0; else nn=value; } // called for n1.n=4;
  }
}
```

The funny syntax (it looks like a function named `n`, except with no parens for the input) is the special rule. `get` and `set` aren't real function names. `int x=n1.n;` magically calls `get` and `n1.n=3` magically calls `set` with `value` at 3.

The details aren't all that important. This trick is a style thing – if everyone else uses it, so should you. Hopefully it helps show the getter/setter concept.

It's also a fun example of language design. Older languages thought of this trick and rejected it. `n1.n=4;` being a secrete function call seemed too confusing. C# decided, why not?

Many different languages are that way – the features they do or don't have are opinions of the designers.