Chapter 29

Member functions

When we make a class, we often write some helpful functions using it. So far, they were just regular functions. This section is about a new trick where we can put class functions directly inside the class.

For example, the `Dog` class now has `setDog(d1,"Spike",2)` and `showDog(d1);`. In our minds, they're `Dog` functions, on `d1`. But they're really just regular functions with `d1` as a regular input.

We're going to rewrite them with the new rules as `d1.set("Spike",2);` and `d1.show();`.

These are called **member functions**. This chapter is really the other half of how to make a class or struct. First we add all the fields we need, then we write some handy member functions inside it.

This might feel like a lot of rules for not much gain – these won't do anything we couldn't do before. But all of the built-in classes use them.

As usual, I'll sneak up to it with an example, a few rules, why we think this is a good idea, then more examples and rules.

Then two notes: classes and structs use these the same way, buy I'll use only classes to make it simpler. And, another word for fields is **member variables** – that way a class has member variables and member functions.

29.1 Example and motivation

Here's a rewrite of `setupDog` as a member function. It's written inside the class. There's no `Dog` input, since it has special rules:

```
class Dog {
  public string name;
  public int age;

  public void setup(string nm, int howOld) {
```

```
    name=nm; age=howOld;
  }
}

// sample use:
Dog d1 = new Dog();
d1.setup("Spot",2);
```

`setup` doesn't have a `Dog` input in the parens. Instead, it requires one `Dog` in front. Inside, it uses just `name` and `age` – it automatically knows that means `d1`'s name and age (more later on this rule.)

It helps to see why we're doing all this work. Some of the advantages of having member functions:

- Putting `d1`-dot in front makes it easier to see this is a `Dog` function when we're reading it.
- It also makes it easier to see `("Spot",2)` are the real inputs. In this case `d1.setup("Spot",2);` makes it look more like an assignment statement, which it mostly is.
- When we're writing code, it's easier to find the `Dog` functions. We can type `d1`-dot and search the pop-up: the same pop-up that shows the fields also shows the member functions.
- We can use shorter function names. As a regular function, `setupDog` seemed good. But as a member function, `setup` is fine, since we know we're inside `Dog`.
- The "using this dog" `name`/`age` shortcut can make the inside of member functions a little nicer.

Part of the idea is that member functions are things the class knows how to do. `d1.setup("spot",2);` is like telling `d1` to set itself up. Like `d1` has it's own, personal `setup` function.

When you're writing the function, you're supposed to feel like you're the `Dog`, doing stuff to yourself. Inside the body, when we use `name` and `age`, it's like saying my name and my age.

Here are the member function rules listed out:

- Member functions are written inside the class. The extra word `public` must be in front, but otherwise they look normal.
- They can only be run by a variable of that type, using variable-dot-function. Like `d1.setup("Spot",2)`.

- The variable that runs them is an automatic input. It doesn't have a name inside the function. Instead you use it's member variables "naked." In a dog member function, just `name` and `age` means the name and age of the dog that called you.
- Member functions, even in structs, are allowed to change the parts of what called them. For example `frog1.doFrogStuff();` could be changing variables inside of `frog1`.
- The order you write them doesn't matter. Member functions can come before, after or in-between member variables.
- Forgetting `public` in front isn't an error, but you won't be able to use the function (it won't appear in the pop-up after `d1`-dot, and more importantly, it gives a *function not found* error if you type it anyway.)
- They aren't allowed to use your global script variables. They can use real globals like `Random.Range` or `Time.time`. This is tricky, but it's a good rule. There's an example later.

Here's a walk-though of `d1.setup("Spot",2);` using the rules. The class again:

```
class Dog {
  public string name;
  public int age;

  public void setup(string nm, int howOld) { name=nm; age=howOld; }
}
```

When we see something like `d1.setup("Spot",2);` we know `d1` is calling its member function `setup`. We'll know to search for it inside the dog class.

To run it, we copy `"Spot"` and `2` as usual. The new rule says to remember we're running it for `d1`.

When we get to `name=nm;`, there's no definition of `name` in the function – it's not an input or a local variable. It's like a global, but inside of `Dog`. This is where the special "running on `d1`" rule kicks in. `name` means "my name" which in this case is `d1`'s name.

29.2 More simple examples

Dog examples

Here are some more `Dog` member functions, all with no inputs:

```
class Dog {
  public string name;
  public int age;
```

```
  public string desc() { return "Name: "+name+" "+age+" years old"; }

  public void reset() { name=""; age=0; }

  public bool isAPuppy() { return age<=2; }
}
```

We know they really all have 1 input – the dog that calls them.

`desc()` is the same simple "help print me" function we've written for structs before. We'd call it like `string w = dog2.desc();`. The only funny thing is inside we use `name` and `age` and know it means `dog2`'s name and age.

`reset` is showing the rule how you're also allowed to change them. `pet1.reset();` would blank out the inside of `pet1`.

`isAPuppy` is more interesting: `if(d2.isAPuppy()==true)`. But the inside of the function is pretty boring: we're just looking up the age of who called us.

Again. these would have been fine as normal functions, but are a little nicer rewritten as member functions.

Member functions with inputs use the inputs in the normal way. This checks whether a dog is within an age range:

```
  public bool inAgeRange(int low, int high) {
    return age>=low && age<=high;
  }
```

We could use `if(d2.inAgeRange(2,5))`. The inside is nothing special. 2 and 5 would be copied to `low` and `high` as normal. For the two `age`'s it would just know to use `d2.age`.

This is one of those very minor shortcut functions. We'd only write it if we checked age ranges a lot.

This most complex use is taking another dog as input. The standard example is the `copy(toDog, fromDog)` fucntion, rewritten as a member function:

```
  // inside of Dog:
  void copyFrom(Dog dd) {
    name = dd.name; // <- one has a dot, one doesn't.
    age = dd.age; // <- "age"=my age
  }
}
```

We'd call it like `d1.copyFrom(d2);`. Inside, `d1` is the special dog, while `dd` will point to input `d2`, as normal. `name=dd.name;` means "change my name (`d1`) to the input dog's name (`d2`)."

Mover

Way back in the struct chapter I made something to hold the four variables we need to slowly move and wrap-around. A copy:

```
struct Mover {
  public float val; // the main thing
  public float spd; // amount to change val, each update
  public float min, max; // the limits for val
}
```

Back in that chapter, I had code in Update to hand move `val`. Putting it as a member function would be a little nicer. If you remember, `val` goes up or down by `spd`, wrapping around:

```
struct Mover {
  // same variables

  public bool doMove() {
    bool wrapped=false; // so far
    val+=spd;
    if(val>max) { val=min; wrapped=true; }
    if(val<min) { val=max; wrapped=true; }
    return wrapped;
  }

  public void setRange(int low, int high) {
    if(low>high) low=high; // we may as well check this
    min=low; max=high;
  }
}
```

The moving code is (mostly) the exact same as before, but it's nice how it's right next to the variables it uses. `SetRange` is one of those helpful member functions you may as well throw in there.

Rewritten this way, code to change us from red to black looks really nice:

```
Mover redMove;

void Start() {
  redMove.spd=0.02f;
  redMove.setRange(0.2f, 1.0f); // <- fun way to set min and max
  redMove.val=0.2f;
}

void Update() {
```

```
  redMove.doMove(); // <-same old wrap-around code, but easy to read here
  transform.getComponent<Renderer>().material.color=new Color(redMove.val,0,0);
}
```

The feeling is suppose to be that `redMove` knows how to set it's own range, and knows how to do a move on itself.

I added the `bool` return as a typical "mostly do something, but return info about how it went." We can call `m.doMove();`. which ignores the output. Or we can check it with `if(redMove.doMove()) ...` to do something special on wrap-arounds.

BoardSquare

If you recall, a bit ago we made a simple class to hold data for one square in a game board. That particular one allowed the user to toggle between two colors (I called that selecting a space) and it showed which space you were "on" by making it a little smaller.

The old code worked, fine but those seem like things that should be member functions. We can tell a square it's been selected or unselected, or whether we're on it:

```
class BoardSquare {
  public GameObject gg; // same as before
  public int selected;

  public select(bool newSelectStatus) {
    selected=newSelectSatus;
    Color cc=Color.blue;
    if(selected) cc=Color.red;
    gg.GetComponent<Renderer>().material.color = cc;
  }

  public highlight(bool isOn) {
    float sz=1.0f; if(isOn) sz=0.8f; // highlight makes us a little smaller
    Vector3 sz = new Vector3(sz, sz, sz);
    gg.transform.localScale = sz;
  }
}
```

Now `bs.select(true);` selects a square. To move from square `s1` to `s2` we'd use `s1.highlight(false);` and `s2.highlight(true);` In the old program, with the board:

```
if(space key) Board[col][row].select(true);
// move
Board[oldCol][oldRow].highlight(false); // un-highlight old space
Board[col][row].highlight(true);
```

This is a good place to show the last rule, which I promised an example of: member functions can't use globals from your script.

Our main program using boardSquares has globals `row` and `col` and a grid `GG`. The boardSquare class can't see or use those variables. For example, suppose we want a member function that highlights us and un-highlights the old one:

```
// illegal select function:
public moveHere() {
  ... // change my size
  // Now turn off old one:
  GG[col][row].select(false); // <- error, can't see GG, row or col
}
```

This seems like maybe it should be legal, since the class `boardSquare` is inside the big script with those three variables. But remember we can have multiple copies of a script, or no copies – there can be several `row` variables, or none. Each boardSquare would need an extra, invisible link to which copy of the script it counts as being inside of. Blech.

Put another way, classes are always stand-alone. We let you define them inside of a script just to be nice. But it's nice and clean to force them to only use inputs and their own member variables.

29.3 Using your own member functions

The rule about using your own variables without dots also applies to your own member functions. You use them without dots, too.

Here dog's have a `d1.superDesc()` which uses their `desc()` function:

```
class Dog {
  // old desc
  public string desc() { return "Name: "+name+", "+age+" years old"; }

  // new superDesc:
  public string superDesc() {
    string dw = desc(); // <- call "my" desc() function
    return = "<<<< "+ dw +" >>>>>";
  }
}
```

When you call `d1.superDesc()` it runs `desc()`, still on `d1`. It's really the same old rule: member functions don't use a dot to use other things in the class. And it feels natural once you do it a few times.

The rules for calling one of your own member functions work perfectly, with no surprises. For fun, here's a completely fake example function that uses two Dogs, showing how it works even for funny stuff:

```
class Dog {
  // ...
  public string doubleSuperDog(Dog other) {
    string w1 = "dog#1 "+superDesc(); // runs my superDesc
    string w2 = " dog#2 "+other.superDesc(); // runs its superDesc
    return w1 + w2;
  }
}
```

The first call to `superDesc` just knows to use ours, which knows to use our `desc()`. The second `other.superDesc` runs for that dog, which runs `desc()` on the other dog.

The computer remembers which dog you're on, even when you flip back-and-forth.

29.4 Built-in member functions

Most of the pre-made structs and classes have member functions. Now that we know how they work, we can enter variable-dot and look for them.

29.4.1 Unity member functions

`Vector3` has a `Set` member function – we can write `Vector3 pos; pos.Set(-7,2,0);`. It's just a shortcut. For fun, this is how it works:

```
struct Vector3 {
  public float x, y, z;

  public void Set(float new_x, float new_y, float new_z) {
    x=new_x; y=new_y; z=new_z;
  }
}
```

There's nothing special about `new_x`, that's just the name they picked.

Oddly, `Color` doesn't have a `Set` function or anything like it. They just didn't write one.

I used `transform.Translate(1,0,0)` several chapters ago without explaining the dot. Now we know it's a member function. `transform` is a variable (you get it for free in Unity if your script is on a gameObject.) When you type `transform`-dot, you can see field `position` and member function `Translate`.

If you have `GameObject g;`, typing `g`-dot shows a big list, including one named `SetActive(bool);`. That tells us we can use `g.SetActive(false);` to hide it (and again with true to un-hide).

Put another way, we knew there had to be a way to temporarily de-activate a Cube. There is, and it's a member function since it's better than way.

29.4.2 String member functions

I lied when I wrote that `string` was a basic variable type. You probably figured out that `string` is way more complicated than something like `int`, `float` or `bool`. `string` is really a built-in C# class with a bunch of special rules to make it act like a basic type.

But since it's really a class, it has member functions. As usual, we can type `w`-dot to find them. Lots of them are really fun:

- `w.Contains("a")` is true when `w` has an `'a'` in it. We know this is just an index loop. We've written it already, but it's still a nice shortcut.
- `w.StartsWith("abc");` and `w.EndsWith("abc")` are also simple loops, which we're already written.
- `string w2 = w.SubStr(2,4);` gives a sub-string – start from index 2 and take 4 letters (so positions 2, 3, 4 and 5). `w.Substr(2,2)` on `"catfish"` gives you `"tf"`.
- `string w2 = w.Insert(3,"ABC");` adds `"ABC"` just before index 3 (`"chicken"` would become `"chiABCcken"`) As usual, it returns a changed copy. You'd need to use `w=w.Insert(3,"abc");` to change yourself.
- `string w3 = w.Remove(2,4);` gets rid of 4 characters starting at 2 (it returns the result.)

  A funny thing about this, and the ones before it, are that we know member functions can change themselves. We just prefer returning the changed version and not changing the original, even though we could.
- `w.ToLower()` returns an all-lower case version of `w`. It's used in a clever trick for case-insensitive compares. `if(w.ToLower()=="done")` is true for `"done"`, or `"DONE"` or `"Done"` ....

Those are all simple loops, mostly things we're written before. But it's still nice to have them "in" the computer as easy-to-find member functions. Also, maybe, they used some tricks to run them a little faster than our versions.

29.4.3 List member functions

Back in the `List` chapter, we used `L.Add(6);` to grow List `:` by a box. That's obviously a member function.

There are a few more:

- `L.Clear();` resets the list to size 0.

- `L.RemoveAt(0);` gets rid of the first item (it slides everything else down, using a loop, then subtracts 1 from the list size.)

  It works for any index. `L.RemoveAt(L.Count-1);` gets rid of the last item, which doesn't need a slide-loop at all.

- `L.IndexOf("cat");` searches for that word and returns the index, or -1 if it's not there. `L.Contains("cat");` is the same but worse – it only returns true or false.

- `L.GetRange(2,4));` is like substring. It returns a list of 4 items from `L` starting at index 2.

There are more than that, but that should give the idea.

29.5 `this`

This is an obscure rule which you'll probably never need. But it helps explain the magic member function rules, and it's a common, obscure, rule in many languages, so it's worth seeing. But don't worry about knowing it or using it.

Inside of a member function, you can use `this` to mean "me". For example you could write `this.name` instead of just `name`.

Times when you need it are rare and strange. Suppose there's a real global function taking a dog as input, which our member function needs to run on itself:

```
class Dog {
  public void dogFunc() {
    // oh no! I need to run dogUsingRealGlobalFunc with me as input:
    int n=dogUsingRealGlobalFunc(this);
    ...
  }
}

int dogUsingRealGlobalFunc(Dog d) { ... }
```

There's no good reason to have a global function like that – it would be a member function. But on big projects funny stuff can happen. Sometimes you're stuck using not-quite-right things.

You'll sometimes see a style where every member variable has `this` in front. There's no reason, but it's legal:

```
class ThisAbuse {
  public int a, b;
  // using extra this's for fun:
```

```
  string desc() { return this.a+" "+this.b; }

  void setup(int a, int b) { this.a=a; this.b=b; }
}
```

In the second function, using `this` let us re-use the names in the inputs. It so happens the computer tries to match with parameters first, so `this.a=a;` copies the input `a` into the member variable `a`. But there's no reason to know that or ever use this trick.

29.6 Constructors

These are a special type of member function used to set up a fresh variable. They seem extra strange, but are very useful and common in most languages. And we've been using them already.

When we use `new Vector3(0,4,0)` and `new Color(1,1,0)` we're calling the constructor. Here's a written constructor for `Cow`, which we could use with `new Cow("Bessy",5,150)`:

```
struct Cow {
  public string name;
  public int age;
  public float wt;

  // a constructor:
  public Cow(string nm, int years, float pounds) {
    name=nm; age=years; wt=pounds;
  }
}
```

The rules:

- The back part of constructors are regular member functions. They take parameters the normal way and run like normal.
- The name of a constructor is always the class name. Above, the class is `Cow`, so the function named `Cow` is automatically a constructor.
- You never write a return type – not even `void`. Leave it blank. It automatically creates and returns a variable of that class.
- You can't run them using variable-dot. They only run using a `new`. In fact, everytime you use `new` it tries to find a constructor.

We can use our new cow constructor in the normal ways:

```
Cow c1=new Cow("Bessie", 8, 425); // with the first new
c1=new Cow("Gertha",6,375); // cheap way to change all
doCowStuff( new cow("Mrs. X", 1,170)); // cheap insta-cow
```

It's common to overload constructors. This lets us also create a Cow with just age and weight, and also a no-input one where the name is always "generic cow":

```
struct Cow {
  // second constructor
  public Cow(int years, float pounds) {
    age=years; wt=pounds;
    name="generic cow";
  }

  // third constructor:
  public Cow() { name="generic cow"; age=0; wt=0; }
}
```

Now we can write `c1=new Cow(4, 410);`, and `c2=new Cow();` and the old `c3=new Cow("A",3,150);`

Constructors are allowed to call other functions. It's common for them to call a `setup` function:

```
struct Cow {

  public Cow(string nm, int years, float pounds) { setup(nm, years, pounds); }
  public Cow(int years, float pounds) { setup("generic cow", years, pounds); }
  public Cow() { setup("",0,0); }
}
```

You never need a constructor. You can hand-set values or hand-run `setup`. But they're very common. Once you get used to them you'll naturally want to write a few as shortcuts.

Horribly confusing C# constructor rules

This is more specific `C#` oddness that doesn't have much to do with programming in general, but since we're learning in `C#`, we may as well see it. Plus it's a good example of strange-for-no-reason rules.

When you write a class, you must use `Frog f1=new Frog();` to create one. If you write your own constructor, you can use `f2=new frog("Kerfnit")`. Well, in `C#` that knocks out the 0-input constructor. `f1=new Frog();` is now an error. You also have to specially write your own 0-input constructor to make that legal

again.

Another oddness is the fake `new` for structs.

Classes need you to use `new` to create the space. Structs already have space – the constructor is just an option, and has a fake `new` in it: if `Lizard` is a struct, you can use `Lizard l1;` and be fine. `l1=new Lizard("lizzie",7);` is just an option, and all it does is change your values. It doesn't create a new lizard (which isn't allowed, anyway.)

29.7 Scope

This is probably obvious: the names of member functions are only for inside the class. It's fine to use the same member function names is different classes, or anywhere else outside the class.

This has three `doStuff`'s, which are fine since they're in different scopes:

```
struct A {
  public void doStuff() { print("A stuff"); }
}

struct B {
  public void doStuff(); { print("B stuff"); }
}

void doStuff() { print("script stuff");}
```

There's never going to be any confusion between `a1.doStuff()`, `b1.doStuff();` and just regular `doStuff();`.

29.8 Style: member vs. non-member

As I mentioned, member functions are just regular functions that we thought looked nicer inside. For some things, it looks nicer to leave them regular.

Compares seem to feel better as normal functions. Here's checking for 2 equal dogs written both ways:

```
class Dog {
  public bool isEqual(Dog d) { return name==d.name && age==d.age; }
}
```

We'd use `if(d1.isEqual(d2))`. That's a little awkward, but not too bad. As a regular function we're comparing two input dogs:

```
bool isEqual(Dog d1, Dog d2) { return d1.name==d2.name && d1.age==d2.age; }
```

This one is `if(isEqual(dog1, dog2))`, which seems a little nicer for a compare.

`Vector3.Distance(v1, v2);` is another one like this. They could have, but didn't, make it a member function: `v1.Distance(v2);`. Since neither point is more special than the other, the first way seems nicer.