Contents

Chapter 37

Recursion

This section has only one real rule: a function can call itself. Each time, you get a fresh copy and a fresh set of local variables. That's the whole rule. As you might guess, the real skill is being able to use it to do something useful.

The first part of this chapter is showing the mechanical rules for recursion – no actually useful examples yet. Then next part will be real, but fakey uses of recursion - doing things recursively that we can already do better without it.

Then the last part will be about real problems where recursion is the best way.

Recursion is a tough topic. You have to just "get" it, and even after that it's a tricky thing to do right. But unlike some other hard topics, knowing just a little is useful.

What will happen is: some problem you're trying to solve will naturally break into parts. You'll notice these parts are the same as the original problem, just smaller. And those break into subparts that are the same way, and so on. That's a naturally recursive problem, which means you need recursion to solve it (hopefully by the end of this chapter at least that will make sense.)

Just being able to recognize a recursive problem is a real time-saver, which isn't too hard once you've seen a few examples.

Another funny thing about recursion: you know how with most tricks – like functions, loops and lists – you should have been using them all along? Recursion isn't like that at all. It's pretty much useless except for those few special problems that can't be solved any other way.

37.1 How calling yourself works

A function is allowed to call itself. Just like calling anything else, it waits for the call to end, then continues. The function `crashMe` below is useless, never prints anything and will crash the computer, but it is legal:

```
void crashMe() { crashMe(); print("Arrg"); }
```

When this calls function `crashMe()`, the computer starts running a second fresh copy. The first `crashMe` call is waiting, and the second copy is running. The second copy calls the third copy and waits, and so on.

This makes an infinite number of copies, never getting to line 2 of any of them. A fun fact – each copy takes a little bit of space. So instead of running forever like an infinite loop, it runs out of memory and crashes the entire Unity editor (same as an infinite loop, be sure to save before testing if you changed the Scene.)

Having a chain of `crashMe`'s, all waiting for the next one to finish, isn't the problem. Never stopping is the problem. We can use tricks to make recursion end, just like we use tricks to make loops end.

This next recursive function is still pretty useless, but it will run without crashing:

```
void A(int n) {
  if(n>0) A(n-1); // <- calling ourself
  print(n);
}
```

It works because of the "make a copy of" rule. Each time we call `A`, we get a new copy of it with a new local `n`.

Saying this in a different way: the idea I gave you before about local variables was oversimplified. I made it seem like you could premake one local `n`, which `A` used whenever it was called. The real trick is we really make a fresh local `n` for each call.

Suppose someone calls `A(5)`. It calls `A(4)`, which calls `A(3)` ... down to `A(0)`, which stops because of the `if`. Here's what the call stack looks like after that:

```
 A       A      A       A       A       A
n:5  |  n:4 |  n:3  |  n:2  |  n:1  |  n:0
```

Five copies of `A` are waiting, with their own local `n`'s. The last copy with `n=0` is running. It prints 0 and pops back to `A(1)`. That local `n` is 1. We print that, pop back and print 2 and so on.

A neat thing, pretend you're the first `A(5)`. All you do is call `A(4)`, wait, then print 5. All that growing and shrinking the call stack is handled by the computer. It seems like this is different because the function you're waiting for is you, but it's not.

The whole thing prints 0 1 2 3 4 5, on different lines. To be clear: this isn't a good use of recursion (but it's pretty cool that it works at all.)

If you think back to the old "function calling some other function" examples, it was nice to print some variables just before and just after the call. This standard recursion example does that. We can watch it "go down" and then come back:

```
void B(int n) {
  print("begin: "+n);
  if(n>0) B(n-1); // <- call ourself
  print("DONE: "+n);
}
```

Not even thinking about recursion, we can see `B(3)` will print `begin: 3` first, then some middle stuff, then `DONE: 3` as the last thing. The entire output is:

```
begin: 3 <- first call
begin: 2
begin: 1
begin: 0
DONE: 0 <-finally, we stop calling and start returning
DONE: 1
DONE: 2
DONE: 3 <- leaving first call
```

It's sort of the same logic as when A calls B calls C. The middle part is what `B(2)` does. And the middle of *that* is what `B(1)` does. But even so, yikes!

37.2 Recursive thinking

A recursive function starts with a recursive idea. You have something you can solve by doing a little bit of work to make it the same problem, but smaller. Then you run a copy of yourself solve the smaller version, which repeats the process.

Here's a recursive idea to compute powers of 2: to get 2^n, find the next lower one and double it. For example, 2^4 is two times 2^3. It's recursive since 2^3 is the same problem, but smaller. Another copy of ourself can be used to solve it:

```
int powTwo(int p) {
  if(p<=0) return 1; // 2 to the power of 0 is 1
  else return 2*powTwo(p-1);
}
```

Let's forget about how we can already do this with a loop. Do you see how beautiful that last line is? `return 2*powTwo(p-1)`. It says "call a function to find the next lower power of two, double it, and I'm done."

If `powTwo(3)` computes 8, then `powTwo(4)` will compute 16.

Here's a picture of the largest stack frame if we call `twoPow(4)`. Four functions waiting for the fifth to finish:

```
p:4  |  p:3  |  p:2  |  p:1  |  p:0
```

The fun part is when they start returning things. Each one gets the "one lower" answer, doubles it, and returns that. As the functions pop back, the return values look like the second line:

```
   p:4 | p:3 | p:2 | p:1 | p:0
16 <-  8 <-  4 <-  2 <-  1 <-
```

I think of it as a bunch of guys in a row. Each guy asks the one to the right, then waits. Eventually, they hear the answer, double it, and tell that to the guy who asked them.

You may have realized the "how do I stop" problem is similar to the one we had with loops. So far, we've been counting down to 0, but, like loops, recursion can have all sorts of plans for when to stop.

In this next one I want to pad a string up to a certain length by adding dashes around it. For example `"cow"` padded to 8 would be `"--cow---"`. My recursive plan is this: add a dash on each side and have someone else finish it. And, of course, check if you only need one dash.

The key is how lazy it is. Adding a dash on each end is like a single step of work, and the result is a "more done" version of the same problem. We can hand it off to another version of ourself. The code:

```
string padCenterDashes(string w, int wantSz) {
  print("begin with "+w); // TESTING
  int extraNeeded=wantSz-w.Length;
  if(extraNeeded<=0) return w; // already too big
  if(extraNeeded==1) return w+"-"; // fix and done
  // add one to each side and have my clone finish it:
  return padCenterDashes( "-"+w+"-", wantSz ); // <- recursive call
}
```

Some notes about this:

- A run with `padCenterDashes("cow",8);` calls itself two more times. Once with `"-cow-"`, again with `"--cow--"`. That last one knows to only add a dash to the end.

- Checking this is typical loop-logic. `w` always gets larger (by at least 1), `wantSz` never increases, so this will eventually stop when `w` gets big enough.
- This isn't an example of where we need recursion. It's pretty fake. For real it's easy to have loops add 1/2 the dashes to the front and back. But this way is pretty cool how it avoids math.

You might notice one difference between recursion and loops: when a loop ends, it just ends. Recursion usually ends with an answer. It ends by saying "hey - this input is so easy I can just do it." We call that the **basis step**.

In the power-of-2 functions, the basis step is `p=0`. In this one, there are two basis steps: big enough, and only needs one more.

This next one is another oddball: finding the largest thing in part of a list, using recursion. The plan is to grab the first item, use recursion to find the largest thing in the rest, and compare:

```
int largestInPart(List<int> N, int start, int end) {
  // basis step. if only 1 item, it's the largest:
  if(start==end) return N[start];

  int nFirst=N[start];
  // use my clone to find largest in the rest of the list:
  int nRest=largestInPart(N, start+1, end);
  if(nFirst>nRest) return nFirst;
  else return nRest;
}
```

Like before, this isn't a place where we'd normally use recursion, but it's still neat to see how lazy it is. Each copy only compares 2 numbers.

This shows another funny thing about recursion. This only works since it has `start` as an input. We need it so we can add 1 when we call ourself. This is pretty common. It's also common to add a non-recursive front-end:

```
// front-end. This is not recursive:
int largestInPart(List<int> N) {
  // call the recursive function with the extra inputs:
  return largestInPart(N, 0, N.Count-1);
}
```

For recursive functions we often call that a **driver**. Sometimes it has to set up a lot of variables. And then it often has to decode the answer from the recursive part into what we really wanted. So we thought a fancy name like Driver sounded better.

This last example uses a few tricks and also feels very recursive-y. Suppose we have a sorted list and want to check for a number. Since it's sorted, we can look at the middle number, pick the correct half, and search there. Saying it recursively: to find a number in a list, find which half it has to be in, and search that half.

To cut the list in half, we'll have to use the trick where the recursive part takes `start` and `end`, and a fake front-end driver fills them in:

```
// driver to fill in start/end for the real recursive function:
bool isInList(List<int> A, int num) {
  // may as well check some easy stuff, first:
  if(A.Count==0) return false; // nothing in list
  if(num<A[0]) return false; // everything in list is larger
  if(num>A[A.Count-1]) return false; // everything in list is smaller

  // have to do a real search. run the recursive function:
  return bSearch(A, 0, A.Length-1, num);
}
```

Here's the recursive code, with extra prints:

```
bool bSearch(List<int> A, int start, int end, int num) {
  print("list is "+start+"-"+end);
  // check basis steps: list is size 0 or 1:
  if(start>end || end<0 || start>=A.Count) { // list is empty
    print("size 0 list=not found");
    return false;
  }
  if(start==end) // size 1. Either this is it, or it's not:
    return A[start]==num;

  int mid=(start+end)/2;

  if(num<=A[mid]) { // check first half of list:
    print("searching H1= "+start+"-"+mid);
    return bSearch(A, start, mid, num);
  }
  else { // check second half of list:
    print("searching H2="+(mid+1)+"-"+end);
    return bSearch(A, mid+1, end, num);
  }
}
```

Getting the math correct for the halves is tricky – there are lots of places to have an off-by-one. But those are details. Instead look how beautiful the recursive calls are: `bSearch(A, start, mid, num)` checks the first half and `bSearch(A, mid+1, end, num)` checks the second half.

A new thing is that previously we made exactly the same recursive call every time. Now we choose which one. But the important thing is we're always getting shorter, and closer to being done.

Some notes: 1) This only works if the list is sorted, 2) there's already a built-in BinarySearch in the List class, 3) for real a loop is better: each time would move `start` or `end` and it stops when they touch, 4) it's called Binary Search since it cuts into 2 halves and binary means 2.

There's one more funny thing about this. Suppose we cut it in half 6 times to get down to the smallest list. That means a chain of 6 functions is waiting for us. We know the answer but there's no way to directly send it back.

We return true or false, and everyone else just sends the same answer up the chain. That's what `return bSearch(A, mid+1, end, num)` does – whatever she said, that's my answer to.

It seems wasteful, but there's no other good way.

37.2.1 Tree functions

The computer science term for parents with children, with yet more children, is a tree. A tree is a recursive data structure, which means only recursion is good at navigating one.

Folders with files and more folders inside is a tree. In Unity (or any 3D environment) trees are commonly used to glue objects together.

Suppose we make three long, thin cubes for fingers, positioned against a flat square for a hand. Moving the hand would leave the fingers behind. To glue them to it, drag them in to make them children. You get this picture:

```
hand
  finger1
  finger2
  finger3
```

Now moving or rotating the hand drags the fingers with it. It's a feature. You can move or tilt any of the fingers, and it will follow the hand using that new, adjusted position. For real, animated 3D models have one "skin" and lots of imaginary bones organized in a tree. So you see this sort of thing a lot.

A more realistic tree might start at the body, with 2 hands a head and a tail, which also have several parts:

```
body
    hand1
        finger1
        finger2
    hand2
        finger1
        finger2
        finger3
    head
    tailA
        tailB
            tailC
                tailSpike1
                tailSpike2
```

This is showing how trees are a recursive data structures. `body` is a tree, but `hand1` is also a tree. `head` is a very small tree. The definition of a tree is a parent, with 0 or more children, which are also trees. The definition loops around back to itself.

It sounds like cheating, but it works pretty well: `tailA` has `tailB` inside of it. What are the rules for `tailB`? Well, it is also a tree, so the rules say it has 0 or more child trees in it.

Before using recursion, we need Unity's tree commands. They work off of the `Transform`. `transform.childCount` is an int, which can be 0. `transform.GetChild(n)` gets one child, using the usual 0-based index (three children go from 0-2).

A sample *non-recursive* function to print someone's name and all of their immediate children:

```
void printMeAndKids(Transform tt) {
  print("parent is "+tt.name);
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);
    print("child "+i+" is "+tChild.name);
  }
}
```

The problem is each child could have more children, or not. Some children could have lots of levels of grand-grand-grand-children.

Here's the super-cool part: each child, by the definition, is another tree. To print everything in a tree, we *call the tree-print function*! The recursive function:

```
void treePrint(Transform tt) {
  print(tt.name):
  int childCount=tt.childCount;
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);
    treePrint(tChild); // <-recursive call
  }
}
```

This is so cool. You don't even print the names of your own kids. They're fully trees on their own, so send them to another `treePrint`.

Some notes:

- This is the first problem where recursion is the best solution. Trying this with loops would be a huge mess of nested-nested-nested loops that still wouldn't work on trees deeper than how many loops you used.
- This is the first recursive example that makes multiple recursive calls. That's the main ability that makes recursion useful, and also super complicated.

  A recursive function making one call each time is just a bad way to write a loop (but is still a good way to practice.)
- This still follows normal function calling and stack rules. But, because of the multiple calls, it grows and shrinks. `body` calls `hand1` which calls `finger1`. When finger1 quits we pop back to hand1, as normal. Which calls `finger2`. Then that pops back to hand1, which pops back to body.

  And now body can finally call `hand2`.

  But that's not a new rule. We already knew that function `A` can call `B`, do more stuff, and call `B` again, since it always resumes from where it left off. But yikes does recursion make it seem complicated.
- The basis step (the required thing where it stops calling itself) is 0 children.
- This is the standard way to hit everything in a tree. It's called a Traversal.
- This happens to run down the long way – everything in `hand1` before looking at hand2. Usually we don't care about the order, as long as we hit everything.

  Printing in order of depth (all children, then all grand-children . . .) makes as much sense, but there's no easy tweak to do it. Recursive functions can be delicate – they work the way they work

To show those last two points, this will take any tree made of simple objects and turn them all a color. It's a copy of the same code with "change color" where printing was:

```
void setTreeColor(Transform tt, Color cc) {
  // set our color. Bonus check to be sure we're colorable:
  Renderer rr = tt.GetComponent<Renderer>();
  if(rr!=null) rr.material.color=cc;

  int childCount=tt.childCount;
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);
    setTreeColor(tChild, cc);
  }
}
```

This next example is to show how recursion can be deceptively hard. Suppose instead my plan was to color each child as I saw it, then make a recursive call only if it had children. That “feels” better and seems like it would run faster, but it has a bug:

```
void colorRecur2(Transform tt, Color cc) {
  int childCount=tt.childCount; // this could be 0
  for(int i=0; i<childCount; i++) {
    Transform tChild = tt.GetChild(i);

    Renderer rr = tChild.GetComponent<Renderer>();
    if(rr!=null) rr.material.color=cc;

    if(tChild.childCount>0) colorRecur2(tChild, cc);
  }
}
```

The bug is it won’t color the main parent (body, in my example.) We could add that, but then it would color everything twice. It’s not an error in this case, but for some things it would be.

A classic tree function is counting everything in the entire tree. It’s dastardly how something this short actually works (I’m also using a shortcut for going through every child):

```
int treeCount(Transform tt) {
  int count=1; // me;
  foreach(Transform tChild in tt) // visit every child shortcut
    count+=treeCount(tChild); // <- recursive call
  return count;
}
```

It doesn’t even add `tt.childCount`! It hits every item, in the usual way, and counts them all as 1.

You can test it recursively! On a childless item it returns 1. If we have four normal children, the loop runs 4 times, adding +1 each time, which makes five. Suppose we have two of those things in one parent. The answer is 1 + 5 (call on 1st child) + 5 (call on 2nd child).

That’s the way to figure out tricky recursive stuff. Test it on a small thing. When you try it on something larger, all of the recursive calls will be on something you just tested.

37.2.2 Connected / Flood Fill

Finding all of the connected spaces in a 2D grid turns out to be another recursive problem. The solution works out to: mark this space and find all of the free spaces touching it. For all of those spaces, repeat.

That's often called a Flood Fill. Various problems can be solved with it: checking whether every space on a map is connected, or checking whether a fence keeps in a cow (do a flood fill from the cow. Check it can't reach the road).

We'll run this in Unity, so we can see the results in color. That means we need a bunch of boring grid code. You can skip straight to the recursive part. The grid details are mostly obvious.

Boring grid set-up: each square is either a wall or open. Open spaces have a Mark, used to indicate reachable spaces. We'll color walls blue, open areas white, and marks green.

Here's a simple class to handle one square. `G` will be our permanent link to the real square, which we'll make later:

```
// holds info about one square in the grid
class GridSq {
  public bool isWall;
  public bool isMarked;
  public GameObject G; // pointer to the real square

  public bool isNewFloor() { return !isWall && !isMarked; }

  public void setMark() { isMarked=true; colorMe(); }

  public void colorMe() {
    Color cc;
      if(isWall) cc=Color.blue;
      else if(isMarked) cc=Color.green;
      else cc=Color.white;
      G.GetComponent<Renderer>().material.color=cc;
  }
}
```

The string-picture trick is a cheap way to describe a grid. This one has some interesting differently-shaped areas. The `o`'s stand for walls:

```
string[] GridPic = {
 "      o   ",
 "   oooo   ",
 "          ",
 " o        ",
 "oo    oo o",
 " o    o o ",
 " ooo  o   ",
 " o o  o   "};
```

Then here's the code to make it all on the screen:

```
public GameObject gridCube; // drag a flat 1x1 Cube prefab here

GridSq[,] Grid; // "square" 2D array (since quick to set up)

void makeGrid() {
  Grid=new GridSq[10,8]; // size of that picture
  Vector3 pos; pos.y=0; // used to place the tiles
  for(int x=0; x<10; x++) {
    for(int y=0; y<8; y++) {
      GridSq gs = new GridSq();
      Grid[x,y]=gs;
      gs.isMarked=false;
      GameObject gg = Instantiate(gridCube);
      gs.G=gg;

      pos.x=-5+1.1f*x; // usual positioning
      pos.z=4-1.1f*y; // putting 0 at top to match how GridPic is written
      gg.transform.position=pos;

      gs.isWall = GridPic[y][x]=='o';
      gs.colorMe();
    }
  }
}
```

Nothing special or new there. As usual, lots of trial and error and off-end errors to get it to line up.

The flood fill is another way-too-short-looking recursive function. Remember the plan is: mark this space; find all of the free spaces touching it and repeat:

```
public int xStart, yStart; // set to the coords of the space to check from

void Start() { colorConnected(xStart, yStart); }

void colorConnected(int x, int y) {
  // quit if off-edge or not a legal space:
  if(x<0 || x>=10 || y<0 || y>=8) return;
  GridSq gs = Grid[x,y];
  if(!gs.isNewFloor()) return;

  gs.setMark();

  // try all four neighbors, who try it some more:
  colorConnected(x-1, y);
  colorConnected(x+1, y);
  colorConnected(x, y-1);
  colorConnected(x, y+1);
}
```

If you run this, and give `xStart` and `yStart` various values, you should see the different areas turn green. A fun thing is maybe you're not sure where (0,0) is. Run it with (1,1) and see which part turns green.

Some notes:

- This only works because of the global grid and the marks. When one copy of the function marks a space, everyone knows we've already been there. Globals are a common recursion trick.

  We didn't have to make marks for trees since there's only one way to get to each thing.

- This code always stupidly walks into illegal spaces. When it hits a dead-end, it makes recursive calls to walk into the 3 walls around it. It even tries to walk back the way it came (it notices that space is marked, and quits right away.)

  It might be nicer if the 4 calls at the bottom checked for being open before making the call. But it's so cool that only checking the space we're in right now also works.

- It follows the normal function call / stack rules. Each square checks left, right, down and up, but not all at once. A square walks left, then that square walks left, and so on. Eventually a square hits a dead-end, pops back, and only then do you walk right.

  The starting square will call "walk left" and might wait a very long time before it gets to walking right.

- The order of left/right/down/up isn't important. It will affect the order things are marked, but the end result will still be everywhere we can reach from the starting space.

- If the start is in an open area, and you could watch it run, it looks terrible. It won't grow a nice circle. Instead it will snake back-and-forth in a long, winding path. But it will eventually fill in every little crack, which is what matters.

37.2.3 Maze walk

Finding a path between one space and another is a variant of flood fill. The difference is you stop if you land on the destination, and need some way to remember the path.

The secret to saving the path is remembering how the call stack works. If we're sitting on a space 6 moves from the start, we're at the end of a chain of 6 functions which each took one step.

The plan to save the path is to start with a global empty list. Whenever we take a step (including the start space) we'll add that (x,y) to the end of the list. Whenever give up on a space, we'll take it off. The saved list of steps will always be exactly the steps in the function chain we're on now.

Here's the code, with set-up:

```
List<Vector2> Path; // (x,y)'s from start

public int xStart, yStart, xEnd, yEnd; // set these

void Start() {
  makeGrid(); // the old grid from before
  Path = new List<Vector2>(); // length 0, so far
  walkMaze();
}

void walkMaze() {
    // call the recursive function:
    bool found = _walkMazeR(xStart, yStart);

    // show the path
    // (this is hacky. setMark is a cheap way to turn green
    if(found) {
      for(int i=0; i<Path.Count; i++) {
        int xx=(int)Path[i].x, yy=(int)Path[i].y;
        Grid[xx,yy].setMark(); // turn it green
      }
    }
    else print("no path");
}

bool _walkMazeR(int x, int y) {
  if(x<0 || x>=10 || y<0 || y>=8) return false;
  GridSq gs=Grid[x,y];
  if(!gs.isNewFloor()) return false;

  // this space is free, test walk here and try to keep going:
  Path.Add( new Vector2(x,y) );
  gs.isMarked=true; // NOTE: doesn't color it

  if(x==xEnd && y==yEnd) return true; // found it

  if(_walkMazeR(x-1, y)) return true; // found the exit. Done.
  if(_walkMazeR(x+1, y)) return true;
  if(_walkMazeR(x, y+1)) return true;
  if(_walkMazeR(x, y-1)) return true;

  // it was all dead-ends, so erase this step from the list:
  Path.RemoveAt(Path.Count-1);
  return false;
}
```

Running this, with hand-entered start and end's, will make a green path between them. But it's still not smart – it still follows the same winding path in an open area. But it works great if you add walls to make it more mazey.

A fun thing to do is watch this work in slow motion. This next thing is the same maze walk, except it always shows the current path. It will always show a path of green squares, matching the current chain of function calls:

```
// new, funner maze:
string[] GridPic = {
  " oo o      ",
  "  o ooo oo",
  " oo o o   ",
  " o  o   o ",
  " o oo ooo ",
  "      o oo",
  " oooo   o ",
  "    o o   "};

public int startX, startY, endX, endY; // hand-enter fun spots

void Start() {
  mazeDone=false; // global used by mazeSearch
  StartCoroutine(mazeSearch2(startX, startY));
}

// helper function:
bool isInMaze(int x, int y) { return x>=0 && x<10 && y>=0 && y<8; }

// global needed to say when to quit:
bool mazeDone;

IEnumerator mazeSearch2(int x, int y) {
  GridSq ms=Maze[x,y];
  ms.setMark();
  yield return new WaitForSeconds(0.2f); // small delay
  if(x==endX && y==endY) {
    mazeDone=true; yield break;
  }

  print(x+","+y); // testing

  // try to walk x-1, x+1, y-1, y+1:
  int x2=x-1, y2=y; // current test space

  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone) yield break;
  }

  x2=x+1; y2=y;
  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone) yield break;
  }

  x2=x; y2=y-1;
  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone)  yield break;
  }

  x2=x; y2=y+1;
  if(isInMaze(x2,y2) && Maze[x2,y2].isNewFloor()) {
    yield return StartCoroutine(mazeSearch2(x2,y2));
    if(mazeDone)  yield break;
  }

  // no path from here. Walk back
  // (whomever who called us will try other directions)
  ms.isMarked=false; ms.colorMe(); // erase color
  yield return new WaitForSeconds(0.2f);
}
```

The Unity trick using an IEnumerator to slow it down is sneaky, and this doesn't save the path anymore. But it's one of the nicer ways to see recursion working. If you run it in a more open area, you'll even see how each call tries to follow the same pattern.

37.3 Errors

It's very, very easy to get an infinite recursion by mistake.

This tree function accidentally make the recursive call on itself, instead of on a child, so spins forever:

```
void doTreeStuff(Transform T) {
  ...

  for(int i=0; i<T.childCount) {
    Transform tChild=T.GetChild(i);
      doTreeStuff(T); // <- opps!! used myself by mistake. Meant to use tChild
  }
}
```

In this one, the driver accidentally calls itself, instead of the recursive function:

```
bool findPath(int xStart, int yStart, int xEnd, int yEnd) {
  // set things up, then call the real function:
  for(int x=0;x<10;x++) // clear marks, reset color, etc...
     ...

     findPath(xStart, yStart, xEnd, yEnd); // <-- OOPS!! meant to call findPathRecursive
     // this spins forever
}

bool findPathRecursive( ... ) { ... }
```

Chapter 38

First class functions

The one idea in this chapter is letting us use variables for functions.

For example, suppose `f1` was a variable and we assigned it the round-up function: `f1=Mathf.Ceil;`. Then `f1(3.7f)` would be 4. Since it's a variable, we could change it to round-down: `f1=Mathf.Floor;`. Now the same call `f1(3.7f)` would be 3.

We could even go nuts and say `f1=Mathf.Sqrt;`. I don't know what the square-root of 3.7 is, but `f1(3.7f);` would compute it.

Here's how it looks in working code:

```
System.Func<float,float> f1; // declaring f1
f1 = Mathf.Ceil;
print( f1( 3.7f ) ); // 4 - it runs ceiling function

f1 = Mathf.Floor;
print( f1(3.7f) ); // 3 - runs the floor function

f1 = Mathf.Sqrt;
print( f1(3.7f) ); // 1.92
```

From the declaration, you might have figured out that `f1` is limited to `float` input and `float` output functions. But it's not limited to built-in functions. We can point it at one we wrote, and there's even a rule to make up one on the fly (which I'm not showing you, yet):

```
float randAround(float x) { return Random.Range(f/2, f*1.5f); }

f1=randAround;
float x = f1(10); // random from 5 to 15
x = f1(3); // random from 1.5 to 4.5
```

This is going to be a really useful trick. But, as usual, even though you probably guessed them, we should go over the rules first.

38.1 Rules and examples

These things act like pointers – they don't hold a function, they just aim at them. C# has a special name for them (which I'm saving for later.) Most people call them function pointers.

When you write `f1=Mathf.Ceil;`, you're aiming it at that function. That's why there aren't any `()` parens after `Ceil` – you're not calling the function. We've never left them out before, but we've never had this trick before.

Using them works like other pointers. You automatically follow the variable to the function. If you start with `f1=Mathf.Floor;` then `f1(3.7f);` automatically follows `f1` to the function `Floor`, then runs it the normal way.

38.1.1 Declaring function pointers

This is a longish and semi-boring section. Knowing that `System.Func<float, float, float>` can point to Min or Max, but not square root (since it only has one input) is 90% of what you need. Skim this, quit when you get bored, then come back when you need the details.

The technical term for the type of a function is the *signature*, which is just the input and output types. Like regular pointers, there's no such thing as just a generic function pointer. They need to know the signature of functions they can point to.

This example makes `f2` able to aim at functions with two float inputs and a float return. If you look at the sample call, it's obvious why we can't point it to other types of functions:

```
System.Func<float, float, float> f2; // aim at:  float A(float, float)
f2 = Mathf.Max; // legal
f2 = Random.Range; // legal (picks the (float,float) version
f2 = Mathf.Floor; // ERROR, only 1 input (signatures don't match)
float ff = f2(3.4f, 7.9f); // sample call
```

Another example of the same thing, `f3` is for functions with a string input and output:

```
string makeLonger(string w) { w="x"+w+"y"; return w; }
System.Func<string, string> f3;
f3=makeLonger; // legal
f3=Mathf.Max; // not even close to legal
```

The exact way C# makes signatures for these is a little funny, but not too bad. As we saw, you write `System.Func` with angle-brackets. Inside, you list the input types, in order, then the output type *last* (not first, like in real functions – the output goes last.)

For example, `showCat` takes a Cat and returns a string, so has signature flipped as `Func<Cat, string>`:

```
string showCat(Cat c) { return c.name+" is "+c.age+" years old"; }
System.Func<Cat, string> g1 = showCat; // legal. signatures match
```

For a more oddball example, suppose some functions search a `float` array, starting where you tell it, and return one item:

```
float largest(float[] N, int startIndex) { ... }
float nicest(float[] Nums, int startHere) { ... }
```

The signature (inputs then output last) is `Func<float[], int, float>`. We can use it to make a pointer for them:

```
System.Func<float[], int, float> searchFunc;
searchFunc = largest; // legal, signatures match
searchFunc = nicest; // also legal
float num = searchFunc(NList, 3); // sample call
```

This rule works as-is for functions with no inputs. Put the return type by itself. For example, these two die-rolling functions take no input and return an `int`, so have signature `Func<int>`:

```
int twoD6() { return Random.Range(1,7)+Random.Range(1,7); }
int cheatingCoinFlip() { if(Random.value<0.75f) return 0; else return 1; }

System.Func<int> rollerFunc = twoD6; // legal
rollerFunc = cheatingCoinFlip; // also legal
```

For an odder example. `System.Func<Cat[]>` is a no-input function that returns a Cat array.

Functions with no return values are a little different. Instead of `Func` you write `Action` and just list the input types. Two examples of output-less functions and pointers for them:

```
void move(int x, int y) { ... }
System.Action<int, int> f1 = move;  f1(2,-4);

void makeAllCatsThisOld(Cat[] C, int newAge) {
  for(int i=0;i<C.Length;i++) C[i].age=newAge;
}
System.Action<Cat[], int> catChangeFunc = makeAllCatsThisOld;
```

There's one more special case: no inputs and no outputs, like `void doMove();`. For that, you write Action all by itself, with no angle-brackets: `System.Action f1; f1=doMove; f1();`.

Just so you know, there's nothing special about functions with no return values and the Func/Action split. It just worked out that way. You'd normally write `Func<int,void>` for an `int` input and no output, but `void` isn't legal there. Instead of making it be, they decided to create `Action` for no-output signatures. There's nothing special about the word `Action`. Functions with no return probably have side-effects, which make them action-y.

38.1.2 Using function pointers

Once you have a function pointer, you can do three things with it: point it somewhere, call it, or compare it.

Built-in functions and ones we wrote aren't any different to the computer. One pointer can flip between any functions, as long as the signatures match. The rules for changing where they point are the same as regular pointers. `f1=f2;`, makes `f1` point at the same function as `f2` does. We can even use `f1=null;`

The most interesting thing about using them to call a function is how it looks like a real function call. When you see `func1(5);`, you can't tell right away whether `func1` is a real function or a function pointer. It's similar to how in `c1.age` you don't know if `c1` is a struct or a class. In both cases, C# invisibly follows the pointer for you.

The rest of this is just examples of those rules, and little notes:

This is the standard pointer example, showing how we can aim these where-ever we want. `average` is one of our functions, which has the same `float(float, float)` signature as Min and Max:

```
float average(float n1, float n2) { return (n1+n2)/2.0f; }

System<float, float, float> f1, funcPointer2;

void Start() {
  f1 = Mathf.Max;
  funcPointer2 = f1; // <- copy pointer to pointer
  f1 = Mathf.Min;
  float num = f1(5,8); // runs min, 8
  num = funcPointer2(5,8); // 5, f2 didn't follow f1 when we changed it

  f1=average;
  num = f1(5,8); // 6.5
```

One thing to note is the variables aren't "smart" beyond the signature. What I mean is, Min, Max and Average feel like the same kind of math, but we're allowed to aim at any arbitrary function fitting the signature.

For example, we can aim f1 at Random.Range, or even a useless "always -1" function:

```
float alwaysNeg1(float dummy1, float dummy2) { return -1; }

f1=Mathf.Max; // can go here
f1=Random.Range; // but also here
print( f1(6,9) ); // maybe 7.83?
f1=alwaysNeg1; // useless, but legal
print( f1(6,9) ); // -1
```

The `Random.Range` part is a little interesting, since we know there's a version with two `int`s and another with two `float`s. The trick is, `f1=Random.Range;` has to pick which one we use, using the signature of `f1`. When we come to `f1(6,9)`, we've already chosen the `float,float` version. That's not a big advantage, just kind of neat.

Compare really is the same as with regular pointers. This sets two function pointers and compares them to each other:

```
System.Func<float, float, float> f1, f2;
f1=Mathf.Max; f2=Mathf.Min;
if(f1==f2) print("point to same function"); // false
if(f1==Mathf.Max) print("f1 points to Max"); // true
if(f2==Mathf.Max) print("f2 points to Max"); // false
```

The last two lines are something new. We can compare function pointers directly to functions. The same as with single-equals, `if(f2==Mathf.Max)` isn't running the function. It's just checking where `f2` is aimed.

As usual, one way of using pointers is to always assign them, so you never need to use `null`. But if you want, you can use `null` to mean not assigned yet, empty or invalid. A typical semi-useful example:

```
System.Func<float, float> fixerFunc=null;
if(mode==0) fixerFunc=Mathf.Floor;
else if(mode==1) fixerFunc=Mathf.Ceil;
if(fixerFunc!=null) result=fixerFunc(result);
```

As usual, having `fixerFunc` be `null` isn't an error. Trying to follow a `null` pointer causes the problem. Running `fixerFunc(result)` on a null would give the standard run-time crash with nullReferenceException.

38.2 Uses

This whole section has no new rules – just common ways function-pointers are used.

38.2.1 As a regular variable

Any time you're thinking about making an `int` where 1 stands for function A, 2 for function B, you could make a function pointer instead.

Here's part of a program where our special weapon is either a laser, sonic cannon, plasma blob or nothing. This first version uses an integer to say which one it is (no function pointers, yet):

```
void fireLaser() { ... } // pretend these are written
void fireSonicCannon() { ... }
void firePlasmaBlob() { ... }

int specWeapon = -1; // 0=laser, 1=sonic, 2=plasma, -1=none

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    if(specWeapon==0) fireLaser();
    else if(specWeapon==1) fireSonicCannon();
    else if(specWeapon==2) firePlasmaBlob();
  }

  void loadNextLevel(int lNum) {
    if(lNum==2) {
      specWeapon=0; // laser
      ...
    }
  }
}
```

The key messy part is having to remember the 0,1,2 weapon table. When we press a space, we need `if`s to decode the table. Then, when we change levels, we need to look at the table to assign the correct number for a laser.

We can simplify this by changing `specWeapon` from a look-up `int` to a function pointer:

```
System.Action specWeaponFunc = null; // no special weapon yet
// remember System.Action is the special syntax for no inputs or output

void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    if(specWeaponFunc!=null) specWeaponFunc(); // <- easy call
  }

  void loadNextLevel(int lNum) {
    if(lNum==2) {
      specWeaponFunc=fireLaser; // <= nicer than a 0, and activates pop-up
      ...
    }
  }
}
```

The `if` is gone, and I think `=fireLaser` down below is easier to read than `=0`. Another advantage is when we type `specWeaponFunc=fire`, auto-complete will show us the three possible functions.

38.2.2 Arrays of pointers

If we have several functions, putting them in an array makes it easier to pick one. That's just a standard array trick, but it's worth seeing how we use it.

Some game background: when a character hasn't been moving for a while, we usually play an "idle" animation, like stretching. In this example, we have three of them, and want to pick one at random. Pretend each needs a different function to set it up:

```
void beginStretch() { ... } // pretend this starts stretching
void beginHandsOnHips() { ... }
void beginFlex() { ... }
```

We can put them in an array in the usual way:

```
System.Action[] Idles; // array of function pointers

void Start() {
  // set up the array:
  Idles = new System.Action[3];
  Idles[0]=beginStretch;
  Idles[1]=beginHandsOnHips;
  Idles[2]=beginFlex;
}
```

Now we can run `beginStretch` using `Idles[0]()`. That looks funny, but it's just array rules: `Idles[0]` is a function-pointer, so putting `()` after will run the function.

We already know how to pick a random item from a regular array. This picks a random item and runs it:

```
  IdleActions[Random.Range(0, IdleActions.Length)]();
  // runs beginStretch or one of it's friends, at random
```

It might look nicer broken over three lines:

```
  int iaIndex = Random.Range(0, IdleActions.Length);
  System.Action nextAct = IdleActions[iaIndex];
  nextAct(); // run it
```

You're even allowed to use the array-creation shortcut with functions. Start could just make the array in one line:
`Idles = {beginStretch, beginHandsOnHips, beginFlex};`.

I used simple void/void functions since it was shorter, but it might be nice to see what it looks like when we need the angle-brackets. Suppose the idle functions took a float input and returned a bool. We'd do it like this:

```
System.Func<float, bool>[] Idles;

void Start() {
  Idles = new System.Func<float, bool>[3];
  ...
  bool b = Idles[0](3); // run the 1st idle function, with input "3"
```

38.2.3 User-defined keys

This is another example of an array of function pointers. The idea is, we have three actions, drop, pickup and throw, and we want the player to pick which keyboard key does what (we'll make a menu for that, which I won't show in this example.)

We'll start with just a little struct pairing up the function pointer and current keypress. No array yet:

```
// pretend we have void() functions for drop, pickup and throw

struct KeyActionPair {
  public System.Action theAction; // pointer to drop, pickup or throw
  public char theKey; // key that does it
}
```

Then, the same as the idle action example, we'll make an array pre-filled with these pointers, and also the starting letters:

```
KeyActionPair[] KeyActions; // list of all actions and keys

void Start() {
  KeyActions = new KeyActionPair[3];
  KeyActions[0].theAction=drop; KeyActions[0].theKey= 'd';
  KeyActions[1].theAction=pickup; KeyActions[1].theKey= 'p';
  KeyActions[2].theAction=throw; KeyActions[2].theKey= 't';
}
```

Now `KeyActions[0].theAction()` runs the `drop` function, and `KeyActions[0].theKey=ch;` changes the key we use to drop things (the user menu we're not writing would do that.)

Unity likes us to read keys individually. So, to check during play, we'll loop through the array: check if that key was pressed; if so, run the function next to it:

```
if(Input.anyKeyDown) { // not needed, but can't hurt
  for(int i=0;i<KeyActions.Length;i++) {
    if(Input.GetKeyDown( KeyActions[i].theKey )) {
      KeyActions[i].theAction(); // run the function going with the key
      break;
    }
  }
}
```

If you know constructors, this would be nicer if the `KeyActionPair` struct had a simple 2-input constructor for setting the the pointer and the key together. I left it out to avoid clutter.

38.2.4 Passing functions into functions

The best way to explain this trick is with an example. Here's a regular function to count how many positive numbers are in an array:

```
int arrayCount(int[] A) {
  int count=0;
  for(int i=0;i<A.Length;i++)
    if(A[i]>0) count++;
  return count;
}
```

It would be great if we could send in a replacement for `>0`. Now that we have function pointers, we can. The first step is to rewrite greater-than-0 as a function:

```
bool isPositive(int x) { return x>0; }
```

Notice how this is a `bool(int)` function. That's what `>0` really is. It takes any integer, and tells us whether it likes it. Rewritten, it's a `System.Func<int, bool>` function.

The last step is to rewrite arrayCount to take that as an extra input. There are two changes – the extra input, and using it inside the if:

```
int arrayCount(int[] A, System.Func<int,bool> theTest) {
  int count=0;
  for(int i=0;i<A.Length;i++)
    if(theTest(A[i])) count++;
  return count;
}
```

Like any other parameter, theTest is going to passed in by the caller. For example:

```
int[] N = {4, 0, -3, 78, -11};
int c = arrayCount(N, isPositive); // 2
```

Notice, on the last line, how isPositive is still missing the parens. We're not calling it – we're just passing a pointer. Inside the function, theTest points to isPositive and theTest(A[i]) runs it.

After all that work, we can finally count whatever we want, as long as we write a testing function. This counts how many in the array are even:

```
bool isEven(int num) { return num%2==0; }

  // elsewhere in the program:
  int n=arrayCount(N, isEven);
```

Even though isEven is a function name, it's passed like an normal variable. It's just copied into theTest. Below, in the loop if(theTest(A[i])) is now magically running the isEven function, counting how many things in the array are even.

Positive and even are 1-line functions. This next example is just to use a longer one. There's nothing new, but I think it helps. The function checks if a number is a square (1, 4, 9, 16 . . . ,) using a loop (only read it if you like odd loops):

```
bool isSquare(int num) {
  if(num<0) return false;
  int n=0;
  // keep adding 1. Stop when n-squared hits num, or goes past:
  while(n*n<num) n++;
  if(n*n==num) return true;
  else return false;
}
```

There's nothing special about it, and it just checks one number. `isSquare(81)` is true, `isSquare(22)` is false. The key is that it has a `Func<int,bool>` signature, so we can use it in `arrayCount`:

```
  int[] A={9,5,25,3,2};
  int sqCount = arrayCount(A, isSquare); // 2 (9 and 25)
```

I like this example since it's not special at all. We're calling another function over and over from inside the `arrayCount` loop. But we've always been able to do that. The function we call has another loop in it, making the whole thing a nested loop. But we've also always been able to do that.

The only new thing is being able to plug in any function.

Change each one

A function that changes each item in an array uses pretty much the same idea – we give it a function to use on each item. To make it interesting, I'm going to use an array of `Cat`s.

A simple cat-changing function takes one `Cat` as input and has no output. In other words, they will look like `void(Cat)` or officially `System.Action<Cat>`. Here are two sample functions to change one Cat:

```
void makeOlder(Cat_t c) { c.age++; }

void makeRoyal(Cat_t c) {
  // add "Lord" in front, unless the name already has a Lord in it:
  bool alreadyIsRoyal=false;
  if(c.name.FindFirst("Lord")>=0) alreadyIsRoyal=true;
  if(!alreadyIsRoyal) c.name="Lord "+c.name;
}
```

Our cat-array changing function will take one of these as input, and do it to each thing in the array. This is a little simpler than counting:

```
void changeCats(Cat_t[] C, System.Action<Cat> changer) {
  for(int i=0;i<C.Length;i++) changer(C[i]);
}

  // in the program:
  changeCats( MyCats, makeOlder);
  changeCats( CatList2, makeRoyal);
```

Multiple function inputs

Just to show we can, let's write a function with *two* function inputs. Take the array changer function above, and add a checker. In other words, we'll only make some certain cats royal. Here are two sample cat-checking functions:

```
// two sample cat-testing functions:
bool isHeavy(Cat_t c) { return c.weight>10; }
bool isKitten(Cat_t theCat) { return theCat.age<=2; }
```

These cat-checking functions are like the int-checking ones except cats have more parts we can check.

Now we'll add a cat-checker to the cat-array changing function. There's nothing new here, just a combination of the old checker and changer ideas:

```
// new cat-changer:
void changeSomeCats(Cat_t[] C,
       System.Func<Cat,bool> theTest, System.Action<Cat> theChange) {
  for(int i=0;i<C.Length;i++)
    if(theTest(C[i])==true) theChange(C[i]);
}
```

Notice how the top part is starting to get ugly. We've got a `Func` and an `Action`, and I always get confused how `Func<Cat,bool>` is really `bool Func(Cat)`.

Calling it is nothing special, just put the names of the real functions, in the order it says:

```
// cats 11+ pounds get older:
changeSomeCats(C1, isHeavy, makeOlder);

// young cats become Lords:
changeSomeCats(MyCats, isKitten, makeRoyal);
```

This exact example is a little contrived. But the idea is that function inputs are like any other inputs. Use as many as you need.

Change all children

We can use the idea of a change-function with the visit-all-my-children Unity trick. If you haven't seen it, or have but still don't understand recursion, this will make less sense but is still worth skimming.

As a review, here's the recursive function to touch a gameObject, all of its children and so on. As a stand-in, it just prints the name:

```
void touchAllChildren(Transform t) {
  print( t.name ); // <= test action
  int childCount=t.childCount;
  for(int i=0; i<childCount; i++)
    touchAllChildren(t.GetChild(i)); // <= recursive call
}
```

Printing the name is a stand-in for “do something with `t`”. We’ll change it to calling some transform-using function we plug in. The two changes are the extra input and the first line:

```
void changeAllChildren(Transform t, System.Action<Transform> action) {
  action(t); // <= do whatever thing we plugged in
  int childCount=t.childCount;
  for(int i=0; i<childCount; i++)
    changeAllChildren(t.GetChild(i), action);
}
```

I named my function pointer `action` since I couldn’t think of a better name. Here’s a sample function to turn just one thing red, and a call using it to turn all my children red:

```
void turnRed(Transform t) {
  Renderer rr = t.getComponent<Renderer>();
  if(rr!=null) rr.material.color=Color.red;
}

  // dog and all dog descendants turn red:
  changeAllChildren(dog.transform, turnRed);
```

There’s nothing really special here. This is the same idea as the Cat-changer. If you have some way to look at a bunch of stuff, even if it’s a complicated recursion, you can still use the plug-in function trick. We sometimes say we’re *visiting* each item. How we visit them all is one thing. What we do during the visit is the other, separate part.

38.3 Function pointers as callbacks

Unity’s physics system is an example of *callbacks*, which are usually done using function pointers.

The key idea is you just write `OnCollisionEnter`, with the inputs it requires. Then it’s magically called. You don’t need to check for collisions yourself – you just need to know the system is checking for them and will call you when one happens.

The functions you plug into the system are called *Event Handlers* or *Call-Backs* or *Listeners*. The terms are used interchangeably.
The things the system tracks for you are the *Events*. They’re nothing special – just what the people writing it thought you might want to be told about.

What usually happens is you have some pre-written event system. You have to figure out what the events are, what the function signatures look like, and what the rules are for hooking up your callback functions.

Because of that, some examples below aren't complete. I'm trying to show only the important parts, but maybe not doing a great job. Hopefully at least one of these will make some sense:

Drags and Slides

Suppose we have a pre-written script that tracks screen touches, looking for drags and pinches. Each frame it sees a drag or a pinch, it runs our callback function. The interesting part of the code is:

```
// user plugs callbacks into these 2 variables:
public System.Action<Vector2> dragCallback=null; // input is x&y movement
public System.Action<float> pinchCallback=null; // input is pinch amount

// deep inside the code:
void processDrag( ... ) { // pretend this is called for drags
  ...
  if(dragCallback!=null) {
    Vector2 dragAmt = ... touch math here
    dragCallback(dragAmt); // <-call the function we plugged in
  }
```

`dragCallback` is a function pointer. It's purpose is to be a place the user can plug-in their own functions. This code will never change it. The only place it's used is those last two lines.

Notice how we even use the common rule "`null` is legal, and means do nothing."

This user code has `LRcamSlide` to handle drags, which it plugs in with an assignment statement:

```
// pretend this is a preset link to the script, above:
public fingerTrackScript FT;

void Start() {
  // aim the callbacks at our functions:
  FT.dragCallback=LRcamSlide;
  FT.pinchCallback=null; // ignore pinches
  ...
}

void LRcamSlide(Vector2 amt) { // <- or drag callback
  float sideMove=amt.x*20;
  theCam.position+=new Vector3(amt, 0, 0);
}
```

As far as we see, `LRcamSlide` is magically called on drags.

On purpose `LRcamSlide` only uses `x`, to emphasize the pre-written idea. The touch-tracker is written to be used by everyone. It computes and sends all the info anyone might need: `x` and `y`, in a `Vector2`.

Our script is just one thing using it, and we have to follow the rules, which means taking a `Vector2`.

Here's more imaginary user code. Jumping disables camera movement and enables a cheat for pinching:

```
void beginJump() {
   ...
   FT.dragCallback=null; // can't slide while jumping
   FT.pinchCallback=ammoCheat;
   // cheat code: pinch during jump for full ammo
}

void ammoCheat(float p) { // <- need a float to match the pinch signature
  ammo=999;
}
```

This may not be the best way to do it, but it shows how we can plug and unplug at will.

Unity's UI callbacks

Unity's canvas/UI system uses callbacks. Buttons have a script on them named `Button`, with a `void()` callback. In other words, you can write a no-input, no-output function and tell the button to call it when it's clicked.

This is working code:

```
public UnityEngine.UI.Button btn1; // drag in a Button

// button click function for btn1:
public void testBtnClick() {
  string w=""+Random.Range(1,1000); // sample random #
  // NOTE: buttons have a child named "Text"
  btn1.transform.Find("Text").GetComponent<UnityEngine.UI.Text>().text=w;
}

void Start() {
  // add my testBtnClick function as callback for button1:
  btn1.onClick.AddListener(testBtnClick);
```

When the button is clicked, which Unity will check for us, our `testBtnClick` function is magically called.

The last line looks funny because Buttons don't just have one click callback – they have a list. This is a standard bonus feature that most systems have.

`onClick` is a tiny class holding an array of function pointers. `AddListener` is a shortcut for adding to the end. Notice how we're still using no-parens `testBtnClick`. We're sending you the function, not calling it yet.

It's not easy to figure out that Buttons want a `void()` function. If you look at the tool-tip for `AddListener`, it says the input is a "callback function." Then it says the type is `void UnityAction()`, which means `void()`.

Fun fact: this system means you can add several callbacks to a button. It also means changing to a new one requires `RemoveAllListeners` first.

Redirecting OnCollision

This next example is a simple use of a callback. In Unity, collisions will only ever call a script on the object that was hit. If my game has 5 different color balls, doing different things when they get hit, I need a different script for each color. My ball code is spread out over all those scripts and I don't like that.

Instead, I'll write one script, used on every ball, which redirects the collision to a function in my main script:

```
class genericColliderScript : MonoBehaviour {
  // set this to point to the real collision handler:
  public System.Action<Transform, Collision> callBack;

  void OnCollisionEnter(Collision col) {
    // tell it who you are, and your collision info:
    if(callBack!=null) callBack(transform, col);
  }
}
```

By setting the `callBack` variable, we can tell each block what to do on a collision.

This sample main program has the collision functions for two block colors, and sets the blocks up to use them:

```
void handleRedBlockHit(Transform block, Collision cData) {
  // red blocks just die when hit:
  Destroy(block.gameObject);
}

void handleGreenBlockHit(Transform block, Collision cData) {
  // we get a point when green blocks are hit:
  score++;
}
```

```
void Start() {
  // setup a red  block:
  Transform rb=Instantiate(redBlockPF);
  // have it call the red function above, when hit:
  rb.GetComponent<genericColliderScript>().callBack=handleRedBlockHit;
  // for real we'd position it, etc...

  // bunch of green blocks:
  for(int i=0;i<5;i++) {
    Transform gb=Instantiate(greenBlockPF);
    gb.GetComponent<genericColliderScript>().callBack=handleGreenBlockHit;
  }
```

Now all of our block collision code is together, and can easily use our variables if they need to (like `score` for green ones.) It's not a huge improvement, but being able to organize your code is nice.

38.4 Just cramming in a function

For short things, it's a pain to have to write a separate function. There's a shortcut rule to let you write a nameless mini-function directly. This legal code uses the `countSome` function to count numbers less then 10:

```
print( countSome(A, n => n<10));
```

The `=>` symbol was made up just for these mini-functions. A trick to understanding them is you always know the required types. In this example, `countSome` has to take a `bool(int)` function, so it knows `n` stands for the `int` input. A shortcut lets you leave out the `return`: if you just write math, it knows to automatically return it.

There's a longer version doing the same thing:

```
print( countSome(A, (n)=>{return n<10;} ));
```

The syntax can be a little confusing, but if you get the idea you can look up examples easily enough. The official name for these are **lambda expressions**. That's a real computer science term – when functions were invented, back in the 60's, that's what we called them.

You can use this trick to assign; and you can also have multiple inputs. These completely silly examples show both:

```
System.Func<int,int,int> f1;

f1 = (x,y)=>x*y/10; // nameless (very silly) math function
print( f1(5,7) ); // 3 (35/10)
```

Notice how we're leaving out the curly-braces and the `return` again. It assumes `x*y/10` is the answer. The long way is `(x,y)=>{ return x*y/10; }`. No matter what, we leave out the types. We already wrote they're all `ints` when we declared `f1`.

This one has an `if` and two returns, more like a real function. You could write a very long function this way, but it's probably better to pre-write those, the regular way:

```
// nameless max function:
f2 = (x,y)=>{if(x>y) return x; else return y; };
float n = f2(7,9); // 9
```

As you might guess, the main use for this trick is not having to go off and write all those 1-line functions. For my super cat-changer, we can change the name of all 1-year-old cats to Kitty:

```
changeSomeCats(C, c=>c.age==1, c=>{c.name="Kitty";} );
```

Then, a C# note just in case: if you look these up, there's an obsolete syntax for these using the word `delegate`. Ignore that and you'll find this good way further on.

38.5 Sort plus function input

Most languages have a built-in sort-any-way-you-want, using function pointers. You write a function that says whether two things are in order, send it to the Sort, and it puts the whole list in that order.

Your function always take 2 inputs of the same type. In `C#` -1 means in order and +1 means out-of-order (0 means equal, which is good for non-sorting things.) Here's a working example that sorts Cats by age:

```
int catAgeOrder(Cat c1, Cat c2) {
  if(c1.age<c2.age) return -1; // in order
  if(c1.age>c2.age) return +1; // out of order
  return 0; // equal (won't matter for sorting)
}

public Cat[] C; // pretend this is filled in

  System.Array.Sort(C, catAgeOrder); // sorted by age
```

`System.Array.Sort` is built-in. It has a ton of overloads, but one takes `System.Func<Cat,Cat,int>` as the second input (obviously it keys off of the type of the first input.) It uses the function we gave it to compare, using it to decide when to fix out-of-order. The result will be sorted by age.

This one sorts integers based on absolute value, for example [2, -3, -6, 8, 12, -23] wold count as sorted:

```
public int[] Nums = {8, -3, 2, -23, -6, 12};

  System.Array.Sort(Nums, intAbsOrder); // sorted by "size"

int intAbsOrder(int a, int b) {
  if(a<0) a*=-1; if(b<0) b*=-1; // make both positive if not
  if(a<b) return -1; // in order
  if(a>b) return +1; // out-of-order
  return 0; // equal
}
```

A quick check: `intAbsOrder(5,-6)` should say they're in order, and it does (it sees `5<6` and returns -1.) It should say `(-7,2)` is out of order, and it also does (it sees `7>2` and returns +1).

We can also use the nameless function trick. This sorts from longest name to shortest. I'll skip returning 0, for equals, to save space, and use the `?:` if-shortcut:

```
public Cat[] C; // fill with cats
System.Array.Sort(C, (a,b) => a.name.Length>=b.name.Length?-1:+1 );
```

The nameless function says "if the first name is at least as long as the second name, they're in order."

A sneaky use of a nameless function is to flip another one. The -1/0/+1 answer times -1 reverses it perfectly, sorting backwards:

```
public Cat[] C; // fill with cats

// sorts cats by age, high to low:
System.Array.Sort(C, (a,b)=> -1*catAgeOrder(a,b) );
```

This next part is just a little interesting: we like to build these compare functions into namespaces. For example, `C#` can't compare strings alphabetically using `<`. Instead it has a -1/0/+1-style function named `Compare` using string as a namespace:

```
print( string.Compare("cat", "dog")); // -1
print( string.Compare("cat", "cat")); // 0
print( string.Compare("dog", "cat")); // 1

System.Array.Sort(W, string.Compare); // sort strings by alpha
```

When we have a class that we think we may want to sort, we could write some sort-how functions inside it:

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

  // add possible sort functions:
  static public int compAge(Cat a, Cat b) { ... } // -1/0/+1
  static public int compNameLen(Cat c1, Cat c2) { ... }

}
```

Then would could use `Sort(C, Cat.compName);` to sort by name.

38.6 Built-in array function-using functions

C# has some fun built-in functions using an array and a function pointer. An easy one (which I wrote earlier by hand as `arrayCount`) is counting things we like in an array:

```
// need this at the top:
using System.Linq;

int[] A={4,8,7,1,3,12,5};
// give it a function that likes things more than 6:
int n=A.Count( x=> x>6 ); // 3
```

`Count` is written as an array member function. Putting `using System.Linq` at the top adds it. I used the nameless function shortcut, but it works with anything, for example `A.Count(isPositive)`.

It also works for an array of anything. This counts how many cats have long names:

```
Cat[] C; // pretend this is created
int n=C.Count( c=> c.name.Length>12 );
```

There are two more like this. `Where` returns a shorter list of just what we like. `Select` returns a same-length list of true/false:

```
int[] A={4,8,7,1,3,12,5};
int[] B=A.Where( x=>x>6 ).ToArray(); // B is {8,7,12}

bool[] C=A.Select( x=>x>6 ).ToArray(); // C is {f,T,T,f,f,T,f}
```

Select doesn't seem useful right away, but it's there just in case. The extra `ToArray()` just goes there.

There are two `bool`-returning ones. `All` means "is it true for all of these?" and `Any` means "is it true for any of these":

```
int[] A={4,8,7,1,3,12,5};
// is anything in A more than 10:
bool hasAnyMoreThan10 = A.Any( x=> x>10 ); // true

// is all of A more than 10:
bool areAllMoreThan10 = A.All( x=> x>10 ); // false

// Or use a premade function:
if( A.All( isPositive )) print("everything in A is positive"); // true
```

A semi useful example `if(C.Any( c=> c.name=="" ))` checks whether any of our cats don't have names yet.

If you hate these, the important thing is merely knowing they exist. The computer thinks function pointers are so cool that it adds built-ins using them.

38.7 Older `delegate` syntax

C# has another way to create function pointers. It wouldn't be worth mentioning except it uses a common computer trick – a Typedef. It's also a fun trip into computer language archaeology.

A typedef is a way to make a shortcut definition. A typical one might let you use `intGrid` as a short version of `List<List<int>>` (remember that a List of Lists is how you make a grid.)

`C#` doesn't have that in general, but it has a version that only works for function pointers. This make `chooser` be a shortcut for `system.Func<float, float, float>`:

```
delegate float chooser(float a, float b);

chooser f1; // declare f1 as function pointer
f1=Mathf.Max;  f1(7,3); // this is legal. answer is 3
```

`delegate` is a keyword. It lets the computer know the rest of the line is creating a shortcut. The trick is it means "`chooser` stands for functions that look like this."

That's the way most languages declare function pointers – by making something that looks like the function signature it takes. `C#`'s `System.Func` stuff is more of an oddball.

If you look at the official type for `System.Array.Sort`, it says it takes `Comparison<float>` as the second input (it will have whatever type your array is.) That's another use of the shortcut.

As we know, it takes a `System.Func<float,float,int>` function – compare two floats and return a -1/0/+1 int. They just thought naming the shortcut Comparison was a nice hint.

Chapter 39

Inheritance

After playing around with Unity for just a little while, you probably figured out *component* is the word it uses for things you can add to gameObjects. For example, rigidbodies are in the Component menu. You probably also noticed the Inspector doesn't seem to have preassigned slots for each type – it acts like one list with combined meshes, scripts and colliders, all jammed side-by-side.

You may have looked it up and seen component isn't just a word for humans. There's an actual C# class named `Component`, and the Rigidbody class also counts as a Component. What you see in the Inspector really is a single List with different classes in it. That's illegal, except it's somehow fine because it's also a list of `Component`'s, which everything also magically counts as.

That trick is accomplished using *inheritance.*

There are three parts to inheritance. Part one is the rules for making a class that grows from another one. Part two is making a pointer that can aim at either class. Part three is about using that pointer to call functions in a nice way.

Part one is mostly useless by itself, but we have to know it for part two. I'll write examples, but don't try to figure out how they'd be useful for real. Part two is how Unity accomplishes the single list of different types of things trick. But part three is where you finally see a real example and can understand why we invented inheritance.

39.1 Pre-inheritance example

The simplest, most basic use of inheritance is making two classes which have a lot in common. For example, Cats and Dogs will share name, age and weight.

We can do that pretty well using tricks we already know.

We might split off common animal data into a “helper” class, maybe named `Animal`:

```
class Animal {
  public string name;
  public int age;
  public float wt;
}
```

Now we can make the `Cat` and `Dog` classes using an `Animal` plus specific variables for that one kind. There are no new rules here yet:

```
class Cat {
  public Animal A;
  int cuteness;
}

class Dog {
  public Animal A;
  public float barkVolume; // in decibles
  public string favoriteChewToy;
}
```

The advantages of this idea are probably obvious – it’s basic “don’t write the same thing twice.” But I’ll list them anyway:

- Shorter code (especially if we have lots in common, and create more than just Cat and Dog.)
- Can be easier to read. Once you get used to it, `Animal A;` is the shortest, clearest way of saying a cat has all the basic animal variables.
- It ensures they’re all spelled the same. The weight is automatically `float wt` for every animal. It can’t accidentally be `int wght` for one.
- Easier to add a common stat. Anything you add to `Animal` automatically goes into `Cat` and `Dog`. It also makes sure all they stay in synch.

Another benefit is we can pick out only the `Animal` part. For example getting a pointer to it, or sending it to a function:

```
Animal a1;
// can point to a cat’s or dog’s animal:
a1=c1.A; a1=d1.A;

void printAsTons(Animal a) { print("Tons="+a.wt/2000; }
printAsTons(c1.A); // tons of Cat
printAsTons(d1.A); // tons of Dog
```

There are two sort-of drawbacks. We have to remember to `new` the `Animal`, or else get a null reference error. Not a big deal – we'd just put that in the constructor (this has nothing to do with inheritance, but it is a good constructor example):

```
class Cat {
  public Animal A;
  public int cuteness;
  public Cat() { A=new Animal(); }
}
```

The other annoyance is having to use the `A` for some variables. For a cat, the cuteness is just `c1.cuteness`. But we have to remember the name is `c1.A.name`.

And, to repeat, there's no inheritance yet. This part is showing that we can do a pretty good job having two classes share common data, using the old rules.

39.2 Basic Inheritance

The simplest Inheritance rule makes sharing common variables just a little easier. We start exactly the same as before, by making a class for what they have in common. I'm copying `Animal` here, with no changes:

```
class Animal {
  public string name;
  public int age;
  public float wt;
}
```

Now, the same as before, we want to use this to make `Cat` and `Dog`.

The new Inheritance rule says you put `: Animal` after your name (that's a full colon.) Doing that directly injects everything from Animal into you:

```
class Cat : Animal {
  public int cuteness;
}

class Dog : Animal {
  public float barkVolume;
  public string favoriteChewToy;
}
```

Now Cat and Dog have name, age and wt directly inside them. You don't need to make up an extra name, or use an extra `new`. The `Animal` variables are magically inserted.

To anyone using `Cat`, it looks like a regular class with four variables:

```
Cat c1;
c1.name="Mr. Boots"; // declared in Animal, but used like it was in Cat
c1.age=12; // same
c1.cuteness=4;
```

For some technical terms: we say Cat and Dog inherit from Animal. We'd call Animal the *base class*. It's still just a regular class, but in relation to Cat we'd say it's the base.

We also sometimes say Animal is the *superclass* and Cat is the *sub-class*. That can be confusing, since the subclass has more than the superclass. But everyone uses super and sub that way.

An example with nonsense classes:

```
class Furf { public bool ready; }
class Harhar : Furf { public bool done; }
```

`Harhar` acts like a regular class, with two variables. But `Furf` is still a regular class with just `ready`:

```
Furf f1 = new Furf(); f1.ready=true;
Harhar h1 = new Harhar(); h1.ready=false; h1.done=false;
```

This is a pretty simple rule, so far. Not very tricky, but also not much of an improvement over what we could do without it.

39.3 Intro to Polymorphism

The major advantage of using the official inheritance rule is that a `Cat` also counts as an `Animal`. This is a completely new thing, it's the real reason we invented inheritance, but it takes some explaining why it's so good.

Here's a non-useful example just showing the "counts as" rule. We can make a `Cat` or `Dog`, and use an `Animal` to point to it:

```
Cat c1 = new Cat();
Dog d1 = new Dog();

Animal a1 = c1; // <- animal pointing to a cat. this is legal!
a1.name = "Biggles";

a1=d1; // <- same animal points to a dog. also legal
d1.name="Spot";
```

This isn't just technically legal – it really accomplishes what it looks like. An animal pointer can reach into a Cat to change the name, then do the same thing for a Dog. Because they inherit from it.

The same trick works with function inputs. The Animal-input function from before can now directly take a Cat or Dog:

```
void printAsTons(Animal a) { print("Tons="+a.wt/2000; }

printAsTons( c1 ); // cat input
printAsTons( d1 ); // dog input
```

It's really the same trick. Inside the function, `Animal a` is allowed to also point to a Cat or Dog, since they count as Animals, because they inherit from it.

The trick also works for an array of `Animal`'s. For example, this function finds the longest name in a list of Animals:

```
string longestName(Animal[] A) {
  if(A.Length==0) return "";
  string longest=A[0];
  for(int i=1; i<A.Length; i++)
    if(A[i].name.Length>longest.Length) longest=A[i].name;
  return longest;
}
```

We can run this using an array of Cats or Dogs. It works because each item is an Animal, and Cats and Dogs count as Animals.

The other array trick is using an array of Animals to hold Cats and Dogs, mixed up:

```
Animal[] MyPets = new Animal[3];
MyPets[0] = new Cat(); // putting a Cat into a animal array
MyPets[1] = new Dog(); // and a Dog
MyPets[2] = new Cat(); // another Cat
```

So that's enough about arrays. Let's finish up with restrictions on this trick:

- Being brother and sister types doesn't mean anything. Cat variables can't point to Dogs or vice-versa.
- You're only allow to do stuff based on the type of variable. The real type you're pointing to doesn't matter.

That second rule sounds complicated, but it's obvious once you figure it out. Here's a typical use where `Animal a;` switches between a Cat and Dog:

```
Cat c = new Cat();
Dog d = new Dog();

Animal currentPet=c;

  // switch pets:
  if(currentPet==d) currentPet=c;
  else currentPet=d;

  // use currentPet:
  currentPet.age++;
  if(currentPet.wt>30) ...
```

Using the Animal parts – `name`, `age` and `wt` – is safe, since anything it could point to will have those. But `currentPet.barkVolume` would be an error, since it might not be pointing to a Dog.

`Animal a;` can only do things Animals can do.

Even this is illegal:

```
Animal a = new Cat(); // fine, so far
a.cuteness=6; // ERROR
```

We know `a` is pointing a Cat, but it's still an Animal pointer, so is only allowed to do Animal things.

Functions are the same way:

```
void doAnimalThing(Animal ani) {
  if(ani.cuteness<8) ... // ERROR
  ...
}
```

We can call that with Cats or Dogs, but it's only allowed to do Animal things to it's input.

39.4 Dynamic casting

To review the previous section: you can aim `Animal a1;` at a Cat or Dog, but you can only use it for common Animal stuff. And that's fine most of the time. We can have a list of mixed Cats and Dogs, and an Animal function can find the total weight, or make sure none have the same name.

But sometimes, not as often as you'd think, you want to check what `a1` is really aimed at. And there's one more problem after that. Even if we know `a1` is aimed at a Dog, `a1.favoriteChewToy` is still an error.

One new command solves both problems. `a1 as Dog` checks whether `a1` is a Dog. If not, it returns `null`. If it is, it returns a `Dog` pointer, aimed at `a1`.

An example, then more comments:

```
Animal a1;
 ...
  a1.age++; // works for any animal

  // check for a Dog and do Dog stuff:
  Dog d1 = a1 as Dog;
  if(d1!=null) d1.barkVolume--;

  // ditto, with Cats:
  Cat c1 = a1 as Cat; // is it really a Cat?
  if(c1!=null) c1.cuteness++;
```

`d1 = a1 as Dog` does two things in one. If `d1` is null, it lets us know `a1` wasn't a Dog. That's how the `as` command works. Otherwise it says "now that you know `a1` is a Dog, I'll bet you want to change it? I'll aim `d1` at it, so you can do Dog stuff."

Here's the required silly analogy, from the Seinfeld TV show: suppose you saw someone's phone number on a charity walk mailing list, and wanted to call them for a date. You still have to ask them "can I have your phone number?"

This adds the cuteness of the Cats in a mixed Cat/Dog animal array. It's the same rule, but checking `A[i]` instead of `a1`:

```
int totalCuteness(Animal[] A) {
  int cuteSum=0;
  for(int i=0; i<A.Length; i++) {
    Cat c = A[i] as Cat; // check for a Cat. If it is, c points to it
    if(c!=null) cuteSum+=c.cuteness;
  }
  return cuteSum;
}
```

If we only want to check for the type, we can skip saving the extra pointer. This separately counts how many Cats and Dogs are in an array:

```
void animalChecker(Animal[] A) {
  int cats=0, dogs=0;
  for(int i=0; i<A.Length; i++) {
    if((A[i] as Cat)!=null) cats++;
    else if((A[i] as Dog)!=null) dogs++;
  }
  print(cats+" "+dogs);
}
```

We could have done it the long way. But after a while checking "is it a Dog?" on one line tends to look nicer. Saving a variable you don't need might confuse someone for an extra second.

There's no reason to do this next thing, but if we know `a1` is a Cat, we could use the shortcut to change part of it: `(a1 as Cat).cuteness=5;`. It would crash it wasn't a cat.

But suppose we had a function taking a Cat input, which checked for `null`. Calling `doCatStuff(a1 as Cat)` would be fine.

The technical term for `a1 as Cat` is dynamic cast. It's not exactly a cast, since it doesn't change anything (if it works, it returns the same pointer.) But it changes the pointer type, so it feels cast-y.

Dynamic is a general term for anything involving inheritance that you have to look up while you're running. The opposite is *static*, which means things the compile can do ahead of time.

As you might guess, this only works with inheritance. `int n = f as int;` doesn't even make any sense (just use `int n = (int)f;`).

39.5 Inheritance and functions

You also inherit functions from the base class. It works the usual way – they count as being directly inside of you. Here's a quick, boring example. Cat has two functions: `cStats()` from itself, and it inherits `aStats` from Animal:

```
class Animal {
  ...
  public string aStats() { return "name: "+name+" "+age+" yrs old"; }
}

class Cat : Animal {
  ...
  public string cStats() {
    return "cute: "+cuteness;
  }
}
```

We'd use both of these as if they were inside `Cat`, without having to know which used inheritance:

```
Cat c1 = new Cat();
string w1 = c1.aStats(); // inherited, but called normally
string w2 = c1.cStats(); // normal, and called normally
```

In short, inheriting functions from the base class works exactly how you'd expect, with no new rules or surprises.

There's one new rule: you're allowed to cover up a function with another one. For example, Animals have a standard cost, Dogs use it, but Cats are on sale:

```
class Animal {
  public int cost() { return age*3; } // just any formula for cost
}

class Cat : Animal {
  public int cost() {
    int price = base.cost(); // use cost from Animal
    if(age>1) price/=2; // sale on adult cats
    return price;
  }
}

// Dogs don't have another cost()
```

This works out pretty well. Most types of animals would probably inherit the basic `cost()` from Animal. But some can cover it up with their own. That's usually called overriding.

From inside of the class you're allowed to use the covered-up one: `base.cost()` looks it up. The way Cat.cost looks up the basic Animal cost and tweaks it is typical.

Constructors and base constructors

Skip this section if you don't like constructors. Otherwise this is a neat example of the `base` trick. If you inherit from something, you probably want to use its constructor to help with yours, so a special rule makes it easier.

Here a standard Animal is 2 years old. A standard Cat uses that, then some extra Cat stuff:

```
class Animal {
  ...
  public Animal() { age=2; } // constructor
}

class Cat : Animal {
  ...
  public Cat() : base() { cuteness=5; } // also sets age to 2
}
```

Putting the colon-base in-between like that is a completely special rule, just for this. I like it since it shows how much we love the "borrow stuff from the thing I inherit from" trick.

39.6 dynamic dispatch

This is the rule that makes inheritance and polymorphism worth it.

Suppose we have an Animal pointer aimed at a Cat and call `a1.cost();`. Animals and Cats each have their own cost function, so which do we use?

```
Cat c1 = new Cat();
Animal a1 = c1;

a1.cost(); // run the one in Animal, or in Cat?
```

If we want the right price, we should base it on the real type. In other words, we should do the extra work to check the real thing `a1` points to, and run that version of `cost`.

That extra look-up, each time it runs the line, is called Dynamic Dispatch. Dispatch is another word for a function call, and dynamic is about how we don't know it ahead of time.

To double-check this way is how we want, this finds the total cost of Animals in a list full of Cats and Dogs:

```
int totalCost(Animal[] A) {
  int cost=0;
  for(int i=0;i<A.Length;i++)
    cost += A[i].cost();
  return cost;
}
```

Without dynamic dispatch, this wouldn't care about which type of Animals we're trying to buy, and give wrong costs.

It also shows off the dynamic part. Each time the loop runs, `A[i].cost()`, might run a different function. Imagine we have many more Animals besides Dogs and Cats. That's a lot of work we get, for free.

This is also why we almost never need to hand-check the real type using `as Cat`. We usually have dynamic functions do all of that for us.

Various languages have different rules to enable this. Java does it automatically. In `C#` you need to add extra words. `virtual` goes in front of the function in the base class, and `override` goes in front of all the ones hiding it:

```
class Animal {
  ...
  virtual public string stats() { return "animal"; }
}
class Cat : Animal {
  ...
  override public string stats() { return "cat"; }
}
```

If we don't add both things, `virtual` and `override` in the correct spots, we get either errors or the "other" behavior – `a1.cost()` always calls the Animal version. That way is easier for the computer, but almost always gives results we don't want.

Some notes:

We still can't call non-animal functions this way. For example, if only Dogs have a `needsWalk()` function, you still can't call `a1.needWalk()`. It has to be a function that's in Animal, and then covered up in a subclass.

It also only makes sense if we start with an Animal. For example `Cat c1;` can only ever point to a Cat. There's only one possible cost function it can call, so this trick does nothing for us.

Chapter 40

More Inheritance

The last section was all the basic ideas about inheritance and polymorphism. This section has a few examples, reasoning, and little rules to tweak things.

40.1 A story about dynamic dispatch

This section doesn't have any rules or tricks. You may have noticed the rules for making `a1.cost()` work properly on a Cat or Dog are very odd. This section explains why. It if wasn't bugging you before, skipping this part is safe.

First, to review: if we call `a1.cost()`, and `a1` is pointing to a Cat, we should call the Cat `cost` function. For a second, it might seem as if it should always call the Animal version, because of `a1`, but we quickly realize that would give wrong answers.

So why does `C#` try so hard to make us do it the wrong way?

The two things to know are some languages are very interested in being just a tiny bit faster. And inheritance was one of the last computer tricks we invented.

Compilers like to do as much work as possible ahead of time. It turns out that if `a1.cost()` keys off how `a1` is an Animal, it can preload exactly where to jump for the call. In fact, if `cost()` is short, it might copy the code directly into your function (that's called in-lining.)

Looking up the real type is a lot more work, and you can never use inlining. The computer really has to look up the real type, look that up in a table, check for override's . . . before it knows where to jump.

In an old-style language where this was just invented, like C++, we couldn't just change the commands around. We needed to make up keyword `virtual` to enable it, and we called this new way a virtual function.

We also liked it since speed is important to C++. The general rule is to make you work a little harder to do things that run slower.

In contrast, Java is a much newer language, emphasizing safe and easy over speed. They simplified things so covered-up functions always run the correct way. There aren't any extra keywords or terms, and you don't need to learn the difference.

The way `C#` does it is a neat story about how languages get designed. They decided that since `C#` is more of a general-purpose language, it should have the option for non-virtual functions. That might make some programs run a tiny bit faster.

They should have made virtual the normal way, and have you add extra words to run the bad way. But `C#` was new and they didn't want to scare people away. C++ was the best-known language with both options, so they copied the funny "bad way, unless you add the words to make it good" rules.

40.2 Privacy

If you recall, `private` is a way to help organize things – a way to hide variables from outsiders who shouldn't be using them anyway. So far, the one privacy rule is about what people outside the class can see.

With inheritance, we need more: when you inherit from a class, what can you see from it? We'll grow the privacy rules for that:

- `private` means only you. Things that inherit from you can't see your private stuff. As usual, they have them. But they can't see or directly use them.
- `public` is unchanged – everyone, anywhere can see it.
- A new keyword `protected` means "public if you inherit, private to everyone else." As usual, it can be used on variables or functions.

For an example of each, imagine Animas have a standard cost multiplier based on royalty. But not all animals use it all the time. Our plan is to give Animal a "helper" function computing it. Real animal sub-classes can use it as needed.

And then, we'll give our helper a helper function. Here's what they look like. Note the `protected` and `private` in front:

```
class Animal {
  // helper to compute the cost:
  protected float _getNameCostFactor() {
    if(_nobleCheck("Lord",true)) return 2.0f; // starts with
    if(_nobleCheck("Sir",true)) return 1.4f;
    if(_nobleCheck("Esq",false)) return 1.1f; // ends with
    return 1.0f
  }

  // helper for the function above:
  private bool _nobleCheck(string w, bool checkAtStart) {
    if(checkAtStart) return w.StartsWith(w));
    else return w.EndsWith(w));
  }
}
```

If we type `a1`-dot or `c1`-dot, neither of these will be in the pop-up. That's good. They only exist to help compute `cost()`, which is the function outsiders are suppose to call.

Inside of a `Goat` we can't see the second one, which is also good, since it's only a helper for the first. But we can use the first, in the normal way:

```
class Goat : Animal {
  public override cost() {
    float rm=1.0f;
    if(age>=4) rm=_getNameCostFactor(); // <= using helper from Animal
    // NOTE: the other one, _nobleCheck, wasn't in the pop-up!!
    ... // do more cost stuff
  }
}
```

As usual, if you're not sure, making everything `public` is safe. The only bad thing is your pop-ups will be more cluttered.

The interesting thing about `protected` is how we're thinking of ways to keep the program as smaller parts, where each part can only see what it needs to.

40.2.1 abstract base class

In the Animal, Cat, Dog example, I only made Cats and Dogs. I could have used `new Animal()`, but it wouldn't make any sense (a generic Animal?)

A base class you never plan to create is common. Adding `abstract` in front of the name makes it illegal to create one. It's another one of those rules that does nothing except limit us, and make the program easier to read:

```
abstract class Animal { ... } // new Animal() is now an error

class Cat : Animal { ... } // no change in Cat

Animal a1 = new Cat(); // legal
a1 = new Animal(); // ERROR
```

The way the rule works is tricky. We can still declare `Animal a1;` which will point to a Cat or Dog. We can write functions with Animal inputs, which for real always take Cats or Dogs. We can still have `List<Animal>`, full of Cats and Dogs.

The end result is if we use `new Animal()` by mistake, and that's always a mistake, we get a helpful error. It's also a nice note for someone reading the program.

Abstract base classes are common. It seems like you should always do it. But here's a slightly fake example when you wouldn't:

```
class Hammer { // not abstract. Since we could have normal hammers
  // add hammer stats
}

class ElectricHammer : Hammer {
  // add stats about plug-ins and batteries needed
}
```

In this case, we could have `Hammer h1;` which could point to an actual Hammer or an Electric one. A function like `poundNails(Hammer h)` could take either type.

40.3 Inheritance chains

You're allowed to inherit from a class that inherits from something else. You don't have to list them all – you automatically get everything it gets. In this example, FlyingCats has everything from Cats and Animals:

```
class Cat : Animal { ... }

class FlyingCat : Cat {
  public float airSpeed;
}
```

As usual, everything inside of you looks the same. A FlyingCat has name, age, cuteness and airSpeed, all looking like normal member vars. Likewise it has every function from Animal and Cat (except `private` ones.)

Everything else works the same. FlyingCat's can override functions from Cat. They can override functions in Animal that Cat didn't override. `Animal a1;` can point to a FlyingCat.

One new thing: `Cat c1;` can now also point to a FlyingCat. But we probably won't use that trick.

40.4 Interfaces

Suppose we wrote a base class that had only do-nothing functions. For example, this is for something that might break each time you use it:

```
public class Breakable {
  public virtual checkForUseBreak() {} // call after each use
  public virtual bool isBroken() { return true; } // dummy answer
}
```

Clearly, just this is useless. But suppose a few classes inherit from it and override those two functions. They all count as Breakables and can share Breakable-using functions. For example, `n=getActualUses(popper, 10)` checks whether you can use it 10 times, or if it breaks early:

```
int getActualUses(Breakable b, int timesWanted) {
  int useAmt=0;
  while(!b.isBroken() && useAmt<timesWanted) {
    b.checkForUseBreak();
    useAmt++;
  }
  return useAmt;
}
```

First, note that we could run this normal function with an actual Breakable input. It would always say 0, since the useless `isBroken()` function always says true. But it would run.

Next remember this is our old dynamic dispatch trick. The purpose of `getActualUses` is to run with subclasses of Breakable, using *their* 2 functions.

For example Hammer and Pen could be Breakables:

```
class Hammer : Breakable {
  // Hammers have a 2% chance to break each use:
  bool broke=false;
  public override checkForUseBreak() {
    if(Random.value<0.02f) broke=true;
  }
  public override isBroken() { return broke; }
  // rest of hammer stuff goes here
}

class Pen : Breakable {
  // Pens have ink for 20 uses:
  int usesLeft=20;
  public override checkForUseBreak() { usesLeft--; }
  public override isBroken() { return usesLeft<=0; }
  // rest of pen stuff goes here
}
```

Something functionally useless we inherit, which only "registers" some of our functions, is often called a *contract*. Bt inheriting you promise to write those functions, and in return you now count as a Breakable.

Since it's just a set of functions, we sometimes say you're inheriting another *interface*.

Some languages have an official way to write a a contract-only class. `C#`'s way is pretty typical. You use the word `interface` instead of class, you're not allowed to declare any variables or write bodies for any functions, but you don't have to write abstract, public or virtual:

```
interface Breakable { // counts as abstract (new Breakable() is an error)
  void checkForUseBreak(); // counts as public and virtual
  bool isBroken(); // sub-classes must write these
}
```

Now, when you write `class Hammer : Breakable` you really are promising to write those functions. You get an error if you don't. But otherwise it works the same way. You can declare Breakable variables, or write functions taking a Breakable, and it's understood they'll take sub-classes of Breakable.

Let's jump back a bit. What if we tossed all of this stuff? Say a function with input `q` used `q.age` and called `q.isAdult()`. How about we let you call it with any class that has those two things?

That works – some languages allow it (it's sometimes called *duck typing*, after the joke "it it walks and talks like a duck, it's a duck.") The drawback is it can be hard to know what things a function needs you to have. Using interfaces, a function taking a Breakable is really just saying "you need these two functions to run me."

Back to using interfaces, the main point is being able to have several of them. Suppose we have another interface for things that can move:

```
interface Movable {
  void setTarget(Vector3 pos);
  void move();
  // more move-related funtions
}
```

To make it more interesting, suppose we have a Machine base class, that works about the same way as Animal. We could have real classes that inherit from Animal or Machine, and also get Breakable and Movable interfaces:

```
class Car : Machine , Movable, Breakable { ... }
class Shredder : Machine { ... }
class Bird : Animal, Movable { ... }
class Turtle : Animal, Breakable { ... }
```

Say we have `List<Movable> M;` of things wandering around the map. We'd be able to add Cars, Birds and Turtles. We could call `M[i].setTarget(v);` and `M[i].move();`.

if we really needed to we could check whether any of them break:

```
Breakable bb = M[i] as Breakable;
if(bb!=null) {
  if(bb.isBroken()) { ... } // crash? remove from list?
}
```

This stuff takes practice. It's confusing since a bunch of things blur together. Base classes and interfaces are different ideas, but we inherit from both and use variables the same way for both. The "sub-class counts as" rule is the same for both.

It's one of those things that isn't useful until you have a big, messy program and you're thinking "these 3 classes are similar, but different. I wish I had a way to tell the computer how they all have this certain part in common."

40.5 Built-in interfaces

`C#` has a built in interface for sorting that might help explain the idea.

You might remember that we can run a built-in sort by passing in a 2-item compare function. This is a different one.

The interface requires you to write one compare member function:

```
interface IComparable<T> {
  int CompareTo(T t); // Ex: n = a1.CompareTo(a2);
}
```

If you inherit from this, you promise to provide a `CompareTo` function. Here's the Cat class using it:

```
class Cat : Animal, IComparable<Cat> { // <= added IComparable
  // the required function:
  int IComparable<Cat>.CompareTo(Cat c2) { // yikes!
    // sort by age, using standard -1/0/+1 rules:
    if(age<c2.age) return -1;
    if(age==c2.age) return 0;
    return +1;
  }
  // rest of regular Cat stuff here
}
```

The important thing about this is we now officially have `c1.CompareTo(c2);` that will compare two Cats. We count as an IComparable.

The other built-in sort function uses it. It looks like `Sort(IComparable<T>[] A)`. That looks horrible, but means an array of any class with a `CompareTo` function. Which our Cats now have.

`System.Array.Sort(Cats);` now sorts cats by age.

Some notes:

- I left out lots of specific `C#` details. For example figuring out `IComparable<Cat>.CompareTo` was the way it wanted me to write it. Ouch. But if you understand the basic idea, you can look up those details.
- To repeat, the Sort that takes a second compare-function input is completely different. It can sort any way you want, but you always have to tell it how.
- This way is nice if there's one main way you'd want to sort. Writing it is a pain, but then anyone can use `Sort(Cats)` without knowing anything about how it works.
- This is also an example of overloading magic. `Sort(Cats)` uses the interface method, while `Sort(Cats, catNameLongestFirstComp)` uses the completely different function-pointer method. But it feels like one Sort function, with options.

40.6 Misc examples

40.6.1 `Component` class

Unity3D uses a semi-typical system for making things: everything is a GameObject, and you add sub-parts to make it act the way you want.

The sub-parts are all different, but we want to put them all in one list. To do that, we use the base-class trick. Everything inherits from `Component`, and gameObject's have one List of Component's.

Component holds what everything has in common, which seems like it would be nothing. But everything in the list needs a link to what it's in. So:

```
class Component { // lots of things inherit from this
  public GameObject gameObject; // link to what's it's on
  public Transform transform; // ditto
}
```

This is actually the way our scripts have those two variables (more, later).

If you remember, `Rigidbody` lets something move by itself. It inherits from `Component`:

```
class Rigidbody : Component { // <- inherits
  public Vector3 velocity;
   ... // lots more
}
```

Nothing special here. Rigidbodies get 2 extra variables for free and count as Components.

The same way a List of Animals can hold Cats and Dogs, a GameObject's `List<Component>` can hold Rigidbodies.

Here's a function to find the first Rigidbody if there is one, in a List of Components:

```
Rigidbody getFirstRB(List<Component> C) {
  for(int i=0; i<C.Count; i++) {
    Rigidbody rb = C[i] as Rigidbody; // dynamic casting test
    if(rb!=null) return rb;
  }
  return null;
}
```

40.6.2 Scripts and inheritance

The first line of a script looks like `class testScriptA : Monobehaviour {`. All of our scripts inherit from that class. But `Monobehaviour` inherits from `Behaviour` and that inherit's from `Component`:

```
yourScript -> Monobehaviour -> Behaviour -> Component
```

That's just simple chain inheritance. It means all scripts are Components. They can go into the list.

The `Behaviour` subclass only has `bool enabled;` in it. That's not much, but it's a real example of how just one thing can be fine – don't be stingy with subclasses if you need them.

Other things beside scripts can be disabled, so they also inherit from `Behaviour`. We could do this to skip disabled scripts, lights and so on:

```
// handle all Components
for(int i=0; i<C.Count; i++) {
  // check for a disabled behaviour and skip:
  Behaviour bb = C[i] as Behaviour;
  if(bb!=null && bb.enabled==false) continue;
  // do stuff with C[i]:
}
```

The class Monobehaviour is mostly a tag to let you know it's a script. It doesn't have anything really useful in it, but we can use `C[i] as Monobehaviour` to check for scripts.

So you know, `Mono` is the name of the framework running scripts.

40.6.3 Image subtree example

Unity has two classes to put a 2D picture in the screen: `RawImage` is basic, and `Image` has options to stretch parts of it.

We'd like to make these as interchangable as possible, and they both have a color and a material (what picture they use.) So Unity sets them up with a common base class (which eventually inherits from Component):

```
                    Image (extra stats about resizing percents)
                   /
MaskableGraphic --
   - Color         \
   - Material       RawImage (basic picture)
```

When your scripts needs a link to any picture, do this:

```
public MaskableGraphic catPicture; // really an Image or RawImage

  catPicture.color=Color.White; // legal for both types

  // Using base class trick to work on either:
  void turnRed(MaskableGraphic mg, Color cc) { mg.Color=cc; }
```

We can easily swap in either type, and change the color and picture. We can't easily change the resizing values from the more complex Image type, but we usually don't want to.

40.7 General inheritance advice

One of the things people tend to do is, for no reason, is make subclasses that look like a classification system from school. For example:

```
class Tool { ... }
class HandleTool : Tool { ... }
class SqueezeTool : Tool { ... }
class Wrench : SqueezeTool { ... }
class Hammer : HandleTool { ... }

                / Hammer
    / HandleTool - Screwdriver
Tool
    \ SqueezeTool - Wrench
                \ Scissors
```

But will we ever want `Tool t1;` that can point to any tool, and nothing else? It seems more likely we'll want a variable like `HandItem rightHand;` that can point to any weapon or tool. Or we'll want `List<InvetoryItem>` that can hold every tool, potion, spellbook ....

Will we ever want to call `t1.apply(target)` and have to work for tools, but nothing else? It might make more sense to have `apply()` work for everything, so you can try to apply a potion, or some food (or applying a weapon would try to smash it?)

And, the same way, what use is `SqueezeTool`? Do we need `SqueezeTool s1;` that can only point to a SqueezeTool? Will we ever write a function that only takes a `SqueezeTool` as input?

And then, are there really any common stats that all tools have? They probably have a weight, price and size, but every item in the game has that.

As a counter-example, it might be useful to make a base Pickup class that types of Pickups inherit from:

```
class Pickup {
  public float timeout; // how long until it vanishes
  Vector3 location;

  // everyone will override this:
  virtual public bool use() { return false; }
}

class HealthPickup : Pickup {
  public int healthGain;

  // try to give health to player:
  public override bool use() {
    int needed=100-player.health;
    if(healthGain>needed) // apply some, leave the rest
     ...
  }
}
```

We can use the `List<Pickup> P;` trick to hold all types of Pickups. When we grab one we can use virtual functions to make `P[i].use()` do whatever that type of pickup does.

For many simple things, we don't need inheritance at all – variables are fine. For example, what if we have types of animals, but otherwise they all act the same:

```
class Animal {
  public float tailLen; // -99 means no tail
  public float cuteness; // -99 means no cuteness
  public enum aType {cat, dog, rabbit, snake}
  public aType theType;
}
```

Maybe you can see how this is pushing against the edge. Each animal has a few useless stats (spiders always have tail=-99). Subclasses let us add only what we need. Maybe we need a few special `if`'s for certain animals – maybe lizard tails grow as they age.

Inheritance was actually invented this way. Things like this slowly grow more complicated as you add animals: you get more special-use variables and more `if`'s for one animal type. Sub-classes are a nice way to split those out.

Sometimes you have lots of sub-parts that various things will need in combinations. Inheritance is terrible at that. But that's fine. You can make classes for the sub-parts and use them the normal way:

```
class ElectricHammer {
  public InventoryItem itm; // cost, weight ...
  public UseRestrictions; // what level you have to be to use it
  public PowerSupply ps; // info on batteries and plug-ins
  public PowerSupply ps_backup; // nil = no back-up

  public float secsToPoundNail; // <- finally stats about this hammer
}
```

Notice how there are 2 possible power supplies. Inheritance can't even do that.

Later, you might want these in an inventory list, and want a link to things you hold in your hands. So `ElectricHammer` might inherit from a class you made `HandItem` which inherits from `InventoryItem`. But the other parts would still be just regular member variables.

Another common bit of inheritance advice is how using `as` a lot often means you should write a virtual function. From the Pickup example, our first try might look like this:

```
HealthPickup hlt = p1 as HealthPickup;
if(hlt!=null) hlt.addHealth();
AmmoPickup amo = p1 as AmmoPickup;
if(amo!=null) amo.AddAmmo();
```

This is exactly what having `use()` in the base Pickup class, and then overriding it, is for. Then those lines become simply `p1.use()`.

Chapter 41

Templates

This section is about a shortcut letting us use a variable for a type. We can declare `T val;` and have `T` be a type to be filled-in later. It's used in things like `List<int>` or Unity's `GetComponent<Rigidbody>()`.

There are two versions of it: using it in a class or in a function.

You probably won't use this shortcut in your own code. But there are built-ins using it, and it's nice to understand how they work.

41.1 Template functions

To start with, here's a function that creates an array filled with one number. There's nothing special about it, yet:

```
float[] makeArray(int size, float value) {
  float[] Result = new float[size];
  for(int i=0;i<Result.Length;i++) Result[i]=value;
  return Result;
}

// sample call:
float[] N = makeArray(10, 1.23f); // [1.23, 1.23, 1.23 ... ]
```

An interesting thing is we're not using the "numberness" of the second input. It could be a string or int with no changes.

The template trick lets us actually do that. After the function name, add `<T>`. The angle brackets are required, but the `T` is any name you pick. All of the other `T`'s are filled in for you:

```
T[] makeArray<T>(int size, T value) {
  T[] Result = new T[size];
  for(int i=0;i<Result.Length;i++) Result[i]=value;
  return Result;
}
```

We can call it with `makeArray<string>(3,"tree");`. That turns all the `T`'s into string's. Where it says `T value` for the input, that's now `string value`. And the `T[]` for the return type is now `string[]`. The answer is `[tree, tree, tree]`. Other sample calls:

```
float[] N = makeArray<float>(10, 1.23f); // same as the 1st example
string[] Fives = makeArray<int>(8, 5); // [5,5,5,5,5,5,5,5]
Cat[] CatList = makeArray<Cat>(3,c); // [c, c, c]
```

Note that the last one, with the cats, will have every entry pointing to that one `c`, since that's how `=` works with pointers.

A way to think of them is that the compiler pre-makes the ones you need. If your program has `makeArray<string>` anywhere, the compiler creates that function from the template, with all the `T`'s filled in with `string`. In other words, this really is a template, used to create several copies of itself with various types.

In `C#` template functions are very limited. Other languages allow more. This, which isn't legal in C#, would check whether 3 values are the same:

```
bool allSame<T>(T a, T b, T c) {
  if(a==b && b==c) return true;
  return false;
}

// sample uses:
bool b1 = allSame<string>("cat", "cat", "dog");
if( allSame<float>(F[0], F[1], F[2]) ) ... // compare three floats from array
```

Notice how it has three inputs, but there's one `<T>`. That's on purpose - `(T a, T b, T c)` says they're three things of the same type – the single type you gave it.

These next two just show the rules. This one requires a type, but never uses it, which is legal but pointless:

```
void sayMoo<T>() { print("moo"); }

// sample calls (both print moo):
sayMoo<Cat>(); // moo
sayMoo<float>(); // moo
```

You can have more than one type - put them in the angle brackets with commas. As usual, if you think of a descriptive name, depending on what the function does, use that. I picked just `T1` and `T2` for this:

```
void printStuff<T1,T2>(T1 a, T2, b, T1 c) {
  print(a+" "+b+" "+c);
}
```

This says the first and last have to be the same types, since they both use `T1`. the middle one could be the same, or could be different. Ex:

```
printStuff<float, string>(3.0f, "goat", 9.3f);
printStuff<string, float>("a", 8, "b");
printStuff<int, int>(1,2,3); // <- T1 same as T2 is legal
```

The loosest form of template functions, which is very not legal in `C#`, is where you use whatever member functions you want, and can use any class that has them.

This takes a list of any type and tries to use `c1.lessThan(c2)` to find the small ones. Any class with a lessThan member function can use it:

```
List<itemType> getSmallItems<itemType>(List<itemType> L, itemType maxVal) {
  List<itemType> Ans = new List<itemType>();
  for(int i=0; i<L.Count; i++) {
    itemType itm=L[i];
    if(itm.lessThan(maxVal)) // using dot-lessThan
      Ans.Add(itm);
  }
  return Ans
}
```

If you called tried to call this with a List of `Cat`, it wouldn't compile. But if you added `bool lessThan(Cat c)` as a member function, it would.

This is a substitute for interfaces and inheritance. You just pick some member functions, like `lessThan` or `a.CompareTo(b)` and write template functions using them. Anyone who wants to use them can add those functions to their class.

41.1.1 Implicit types

If the compiler can guess it, you can often call a template function with the `<T>` left out. For example, in `makeArray(3, 5.6f)` the computer can tell that you meant `makeArray<float>` since you used `5.6f`.

This means that overloads and template functions can blur together.

Suppose you see `doThing("gorilla")` and `doThing(6)`. That could be overloading (they wrote a string and int version of the function.) Or it could be a template, `thing<T>(T a)`, being called with the shortcut (the calls were really `thing<string>("gorilla")` and `thing<int>(6)`).

Often we purposely mix overloads and templates with the same name. Not to be confusing, but because it seemed like the best way to make it feel like one function, usable by lots of different types.

Programming-wise, sometimes you need to remember the real way. You may call `doThing(c1);` and get an error. The long way should work: `doThing<Cat>(c1);`.

41.2 Template classes

Classes with variable types work roughly the same way. You add `<T>` in the definition, which the user must fill in with a type. Then everywhere inside the class you can use `T`.

41.2.1 Tuple examples

This makes a simple class that can hold two items of any type. As usual, the words that stand for the types don't matter, but it's traditional to use a capital letter if you can't think of anything else:

```
class Pair<S, T> {
  public S val1;
  public T val2;
}
```

This can make any 2-item-holding class by filling in the types:

```
Pair<string, int> p1;
  ...
p1.val1="toad"; p1.val2=5; // a string and an int

Pair<float, float> ff;
 ...
ff.val1=3.2f; ff.val2=2.9f; // both are floats
```

I left out the part where we `new` them. You have to use the full name of the type: `p1 = new Pair<string, int>();` and `ff = new Pair<float, float>();`.

But that shows how the computer thinks these are different. `Pair` isn't a class, yet. It can be used to make lots of classes, with the `<>`'s.

`Pair` is useful. It's commonly called a tuple. It's in lots of languages as a quick way to group two values.

Suppose we want to get the integer and fractional part of a number. We can return them both using a `Pair<int,float>`. This is a regular function:

```
Pair<int,float> getIntAndFraction(float f) {
  // do the math:
  int wholeNum=(int)f;
  float fraction=f-n;
  // pack in into a Pair:
  Pair<int,float> p = new Pair<int,float>();
  p.val1=wholeNum; p.val2=fraction;
  return p;
}
```

Now we could call `Pair<int, float> nn = getIntAndFraction(4.72f);`. The result would be `nn.val1` is 4 and `nn.val2` is 0.72.

Notice how that is not a template function. It runs using only one exact type (which happens to be a template class).

Sometimes we use tuples as a shortcut for writing a real class. Suppose we want a list of things like: 2 cats, 6 dogs, and so on. We could write a class with name and amount, but `Pair<string,int>` is fine.

We could make an array of them:

```
// An array where each entry is like: (Cat, 2)
Pair<string,int>[] AniCount = new Pair<string,int>[6];
AniCount[0] = new Pair<string, int>();

AniCount[0].val1="ferret"; // <- first item is (ferret, 7)
AniCount[0].val2=7;
```

As a time-saver and neat example, we could make a function that creates pairs. This is a template function, which creates a template class:

```
Pair<S,T> makePair<S,T>(S v1, T v2) {
  Pair<S,T> pp = new Pair<S,T>();
  pp.val1=v1; pp.val2=v2;
  return pp;
}
```

The key to reading it is `makePair<S,T>`. The angle-brackets after the function name are, as usual, the one place we pick the types. Everything else is filled in. But it makes sense when you see it in use:

```
Pair<int, int> p1 = makePair<int,int>(6,9);
// p1.val1 is 6, p1,val2 is 9

Pair<float,string> p2 = makePair<float, string>(4.1f, "cat");
```

The types we write in the brackets determine the required types of the inputs, and the type of the returned Pair.

Since we're giving it the types as input, we get to use implicit types in the call: `makePair("cat", 2);` is a legal shortcut. Here's more of the animal list:

```
AniCount[1] = makePair("cat",2);
AniCount[2] = makePair("rat",12);

print( AniCount[1].val1 ); // cat
```

Backing up, this is a typical case where each part is simple, but the whole thing is confusing since we're doing so much as once: we need a `new`, but `makePair` does it for us; we're using the implied types shortcut in `makePair`, and we're putting a template class in a array.

This next normal function isn't anything new – just more practice. It tells us how many there are of a certain animal:

```
int getAnimalCount(Pair<string,int>[] A, string animal) {
  for(int i=0; i<A.Length; i++)
    if(A[i].val1==animal) return A[i].val2;
  return 0; // couldn't find, means there are 0 of that animal
}
```

This is a pretty simple function. If we had a real class then the `if` would look nicer, like: `if(A[i].name==animal) return A[i].count;`. But with tuples we're always stuck using non-descriptive names like `val1` and `val2`.

41.2.2 More oddball examples

You're allowed to nest template classes. This makes a very crude triple class by using a Pair as the first thing in a pair:

```
Pair<Pair<string,int>,string> Z = new Pair<Pair<string,int>,string>();

Z
------------------
|val1:| val1: frog
|     | val2: 7
|     ---------
|val2: toes
------------------
```

The construction and the picture is no different from any other nested struct, like a Cow holding a Color. But it looks extra bad because of the terrible names. And don't think of it as a Pair inside of a Pair – instead it's a simple class made from Pair, used to make a more complex type of Pair.

Lines using it (to finish the picture):

```
Z.val1.val1="frog";
Z.val1.val2=7;
Z.val2="toes";
```

For real, we'd probably not do this, using either a special class (with better variable names and useful member functions.) Or we'd at least have `Triple<string,int,string>`. But maybe we need to frequently split out the first two, so having them in a Pair is handy.

When you see lots of nested template classes, it can be difficult to know whether there was a good reason to do it.

Here's another simple function using a Pair to return two values. You give it a letter and it searches your array for the first word with it. It returns the index of that word *and* the index in the word where the letter was:

```
Pair<int, int> findWithLetter(string[] S, letter ch) {
  Pair<int,int> Answer = new Pair<int,int>();
  Answer.val1=-1; // not found
  for(int i=0; i<S.Length; i++) {
    int pos=S[i].IndexOf(ch);
    if(pos>=0) { Answer.val1=i; Answer.val2=pos; break; }
  }
  return Answer;
}
```

Some sample calls, to explain how it works:

```
string[] S1 = {"goat", "pony", "rat"};
Pair<int,int> Ans = findWithLetter(S1,'t'); // returns [0,3]
Ans = findWithLetter(S1,'n'); // returns [1,2]
```

Notice how this function only finds letters in lists of strings. It's not a template function.

Member functions in template classes are allowed to use the pre-set types. They don't need extra `<>`'s after the name. Here's a working `set` function for Pair:

```
class Pair<S, T> {
  public S val1; // no change
  public T val2;

  public void set(S v1, T v2) { val1=v1; val2=v2; }
}
```

`set` says that you have to use it with the types you said you were going to use. Some sample use:

```
Pair<string,int> hh = new Pair<string,int>();
hh.set("hamster",4); // <- hh knows it takes string and int

Pair<float, float> ff = new Pair<float,float>();
ff.set(4, 9.876f); // <- ff knows it takes 2 floats
```

All-in-all: template classes have you set the types once, in the declaration. Everywhere inside, even member functions, you can use the “type variables”. They’ll be filled in with the real types.

Moving on, suppose we want to sideways combine two lists. We want [3, 8, 9] combined with [cat, owl, ant] to get a single list with [(3,cat), (8,owl), (9,ant)].

The input lists can be of any types, which means we need a template function. Obviously the new list will use a Pair to make the items. It will get its types from the original 2 arrays.

Here’s the side-by-side array combiner. Remember that we only fill in `<S,T>` after `zipArrays`. The rest of the `S`’s and `T`’s are copied from those:

```
Pair<S,T>[] zipArrays<S,T>(S[] L1, T[] L2) {
  // use length of shorter array:
  int len=L1.Length; if(L2.Length<L1.Length) len=L2.Length;
  Pair<S,T>[] Result = new Pair<S,T>[len]; // result array
  for(int i=0;i<len;i++) {
    Result[i]=new Pair<S,T>();
    Result[i].val1=L1[i]; // copy from the side-by-side array boxes ...
    Result[i].val2=L2[i]; // ...into val1 and vals2
  }
  return Result;
}
```

To review, that’s a template function, which uses it’s inputs to define a specific template class. To make it worse, it can guess `<S,T>` from the array inputs. A sample call:

```
int[] N={3,8,9};
string[] W={"cat","owl","ant"};

Pair<int,string> NW = zipArrays(N, W); // really zipArrays<int,string>(N,W)
print( NW[2].val2 ); // ant
```

Despite how messy it looks, it’s pretty cool to be able to jam just any two arrays side-by-side like that.

This final function is the backwards version of that (well, sort of). It takes an array of some Pair and gives you the first part, cut out from it (it could turn `NW`, above, back into `N`):

```
// input is a Pair array. Output is a simple array of all val1's:
S[] ripOutFirst<S,T>(Pair<S,T> anArray) {
  S[] Ans = new S[anArray.Length];
  for(int i=0; i<anArray.Length; i++)
    Ans[i] = anArray[i].val1; // only 1st values go into the answer
  return Ans;
}
```

Hopefully that first line makes a little sense now. The input is a Pair array, of any types. The output is a simple array, using part 1 of the Pair (since the S's match, not the T's).

41.3 Templates and container classes

A class that just holds items, acting as a nicer or better array, is usually called a *container* class. We've used `List<int>` before. It's just an ordinary template container class.

Here's part of the `List` class written out:

```
class List<T> {
  T[] theArray; <- the entire point of this class is a nice array front-end
  int size=0; // how much of the array we're really using

  public void Insert(int index, T newItem) { ... }
  public int IndexOf(T item) { ... }
  public T ElementAt(int index) { ... } // <- I made this one up
  public void RemoveAt(int index) { }
  ...
}
```

Nothing here is new. `List<T>` says there's no class named `List` – you need to supply a type to complete it.

Inside we know the `T`'s are all copies of the one you declared it with: `T[] theArray;` is an array of string's if you declared `List<string> W;`.

The member functions use the regular template class rule: they know the `T` from when you made it. When you read them, every `T` means "whatever type is in the List."

Another fun created template container class is a Map, which `C#` calls a dictionary. It takes two types. First, an example:

```
Dictionary<string, float> AniWt = new dictionary<string,float>();
AniWt["zebra"] = 5.5f;
AniWt["wombat"] = 3.25f;

print( AniWt["zebra"] ); // 5.5
```

If you read the tooltip, it says `<TKey, TValue>`. Those names are a hint: we can use the first type as a look-up (keys is another word for that), to get the stored values.

From our old rules, we know the two types are allowed to be the same, so we can map animals to the sounds they make:

```
Dictionary<string, string> AniSound= new dictionary<string,string>();
AniSound["duck"] = "quack";
AniCount["cow"] = "moo";

// example of a real look-up:
if(AniCount.ContainsKey("unicorn"))
  print( "Unicorn says " + AniSound["unicorn"] );
else
  print("Unicorn is silent");
```

Maps are strange containers. They don't remember the order you added things, and are terrible if you want to look at everything with a loop. Their only use is for look-ups.

Even if you have int's for the look-up, Maps are good when you have numbers from all over, but not too many. Suppose we have a few hundred ships, with ID numbers from negative to positive a billion:

```
Dictionary<int,string> D = new Dictionary<int,string>();
D[4]="frigate"; // so far an array could do this
D[100278]="yacht";
D[-1093]="pinnace";

// look up an ID:
string shipName="none";
if(D.ContainsKey(shipID)) shipName=D[shipID];
```

That last line is a quick, relatively simple way to check a ship ID. In this case, a Dictionary is better than an array.

Template container classes can have nested templates. This converts the previous "array of name/count" example into a List of Pairs:

```
List<Pair<string,int>> ACount = new List<Pair<string,int>>();
ACount.Add( makePair("cat",2) ); //<- can re-use makePair
ACount.Add( makePair("ferret",7) );
```

You could read `List<Pair<string,int>>` as "a list of string-int pairs".

Here's a more complicated, but semi-common, example of nested template classes. It lets you look up any word and get a list of numbers for it:

```
Dictionary<string,List<int>> AA = new Dictionary<string,List<int>>();

// Make a simple [1,3,8] list:
List<int> ff = new List<int>(); ff.Add(1); ff.Add(3); ff.Add(8);

AA["farm"]=ff; // "farm" is [1,3,8]
```

We can even play compound type tricks with this. `AA["farm"].Count` is 3, and `AA["farm"][2]` is 8 (the last thing in the list).

41.4 Unity template functions

We've been using `GetComponent<Renderer>()` in Unity3D as the first part of color changing. That's obviously a template function. A review of it in use:

```
Renderer rr = GetComponent<Renderer>();
rr.material.color = Color.red;

// with a script we wrote:
catMoveScript cs = cat1.GetComponent<catMoveScript>(); // look-up
c1.name="Soxs"; // now use it
```

It works in a strange way. There's a secret list of `Componenet`'s. Pretend it's named `C`. Component is a base class, and Renderer, scripts and anything else we might search for, is a subtype.

`GetComponent<T>` search the list, checking for the subtype you asked for:

```
// template function: no inputs, returns type you asked for, or null:
T GetComponent<T>() {
  // look at every item in component list:
  foreach(Component c in C) {
    // is it the correct type:
    T cc = c as T; // dynamic cast
    if(cc!=null) return cc;
  }
  return null;
}
```

This is the first template function we've seen that also does stuff with inheritance.

The other common Unity function using a template is Instantiate. A review, this uses Instantiate with 3 different types:

```
public GameObject treePrefab;
public Transform rockPrefab;
public Rigidbody fallingRockPrefab; // hooked up through the rigidbody

void Start() {
  GameObject newTree = Instantiate(treePrefab);
  Transform newRock = Instantiate(rockPrefab);
  RigidBody newFallrock = Instantiate(fallingRockPrefab);
}
```

The way it does this is with a template for the type, used implicitly. The heading for Instantiate:

```
T Instantiate<T>(T copyMe) { ... }
```

When you call, `Instantiate(rockPrefab)`, which is a Transform, the system figures out you meant to use `Instantiate<Transform>`, which means it knows to return a `Transform`. There's more sneaky stuff going on inside, but templates are how it's magically able to return the type you gave it.

Chapter 42

Linked Lists

A linked list is an alternative to an array. It stores a list of items, and can't do anything an array can't do. We care about them because they're very fast at inserting & removing items, from anywhere, and arrays are very, very slow at that.

We don't usually care about a little speed here and there, but choosing the wrong type of list (array or linked list) can give a x1000 slowdown. So linked lists are one of the basic data structure all programmers learn.

Linked lists use pointers, as real pointers, a lot. Playing with linked lists to get better with pointers is very traditional. This chapter also has some fun functions doing that.

A note: we've also seen `C#`'s `List<int>` class. If you remember, that's just a class holding an array. Some languages have lots of alternate data types for storing lists. Deep down, they're always either an array, or a linked list.

42.1 Simple linked list

An array makes a single chunk of memory with each item next to the other. We can find `A[5]` because it's exactly 5 ints past `A[0]`. We can just jump to it using math.

A linked list doesn't bother putting them in order. The items are scattered around, with each one pointing to the next. We call the item plus the pointer a **node** (but you know it's just a small class.)

A linked list named `L1` holding 2, 7, 4, 9 could look like this. The lines stand for pointers:

```
        9-->null
         \
L1--> 2   4
       \ /
        7
```

I moved them around to emphasize how we need to use the pointers to find which item is next.

Each node has two things – the value and the pointer – so we have to make a class:

```
class IntNode {
  public int val;
  public IntNode next; // pointer to next one, or null for last
}
```

The picture above has 4 `intNode`'s, and `L1` is of type `intNode` (but it's only a pointer to the first one.)

Our picture written out as IntNodes would look like this:

```
     --------       --------       -------       -------
L1->|val next|  /->|val next|  /->|val next|  /->|val next|  /->null
    | 2    o-|-/   | 7    o-|-/   | 4    o-|-/   | 9    o-|-/
     --------       --------       --------       --------
```

Backing up a little, if you look at the second field, `public IntNode next;`, you might think "wait – we have an IntNode inside of another IntNode?" Well, no.

If you remember from the pointer chapter, a quirk of C# (and Java) is there are two ideas of pointer variables. The "normal" way is where we own it, it's inside of us, and we have to create it with `new`. The other is a real pointer: we don't create it, we just use it to point to what someone else made.

`next` is being used as a real pointer. Each IntNode "knows about" the one after it, using `next`.

This is also the first good example of how `new` creates "nameless" heap objects. Every time before this, we would have needed `L2`, `L3` and `L4` variables to point to the other three actual nodes.

Now we think of `L1` as being the whole list. The 4 intNode's feel more like just floating things with no specific names. For the 9 node, none of our declared variables points to it, and that's fine. The only way to find it is by following the chain.

42.1.1 Linked List loops

For now, let's just pretend we can make the chain of nodes somehow, and jump straight to using them.

A standard linked list loop starts a pointer on the first item (called the *head*) and follows it one node at a time until we go off the edge with `null`.

This loop prints the list. For example `printList(L1);`:

```
void printList(IntNode head) {
  IntNode p = head; // loop variable. Aim at 1st item
  while(p!=null) { // not past the end
    print(p.val);
    p=p.next; // go to next node
  }
}
```

It walks `p` to point at each item, until it goes off the end. We can rewrite it as a `for` loop:

```
for(IntNode p=head; p!=null; p=p.next)
  print(p.val);
```

This isn't even abusive – it's a perfect use of a `for`. In the parens we can clearly see `p` is the loop variable, then where it starts, when it stops and how it moves. Down below we can focus on only what we do as `p` hits each node.

Here's a quick review of pointers and how they work in that loop. Nothing new:

- Remember that pointers don't "reach through" other pointers. `p=head;` makes `p` point to the first item. When we change `p` later, it won't affect `head`.
- Likewise, `p=p.next;` simply makes `p` move down one link in the chain. In this picture, `p` is the node with 7 and `p.next` is the node with 4 (it's really the arrow from 7 to 4, but what matters is it's pointing to the 4):

  ```
  L1--> 2--> 7--> 4--> 9-->null
             A
             |
             p
  ```

  After `p=p.next;`, `p` is aimed at the node with 4. It just slides down one node.
- `p` will eventually go past the last node and be `null`. That's fine. The loop checks for `null`, which is what you're supposed to do.

  If you think about it, regular array loops also go past the end, so this is the same.

We can use a pointer loop to count how many items are in the list. It's not much more complicated than printing them:

```
int getLength(IntNode head) {
  int len=0;
  for(IntNode p=head; p!=null; p=p.next) len++;
  return len;
}
```

As a quick check, suppose `L1` is `null`. When we call `myLen=getLength(L1);` then `p` starts as null, the loop won't run, and it returns 0, which is correct.

Another simple pointer loop is finding the largest item. As usual, I'll start out with the 1st as the largest, then have the loop start at the second:

```
int largest(IntNode head) {
  if(head==null) return -9999; // list is empty
  int largest=head.val; // 1st item is largest, for now
  for(IntNode p=head.next; p!=null; p=p.next) // loop from 2nd item
    if(p.val>largest) largest=p.val;
  return largest;
}
```

Notice the way the `for` loop starts at the second item: `intNode p=head.next`. This is roughly the same as `int i=1;` in an array loop.

Checking whether something is in the list is another simple pointer loop. For fun I'll use the semi-slimy shortcut where we reuse the input as the loop variable (that's why the 1st part of the `for` is blank):

```
bool isIn(IntNode head, int findMe) {
  for(;head!=null; head=head.next)
    if(head.val==findMe) return true;
  return false;
}
```

As we know, instead of saying true/false if something is in an array, we'd rather return the position, or -1. For a linked list, positions aren't as useful. We'd rather return a pointer to the node, or `null`. This revised item-finding loop does that:

```
IntNode findNodeWithVal(IntNode head, int findMe) {
  for(IntNode p=head; p!=null; p=p.next)
    if(p.val==findMe) return p;
  return null;
}
```

If you don't trust the return-from-middle trick we can rewrite (this is just playing with loop conditions, which is always fun):

```
IntNode findNodeWithVal(IntNode head, int findMe) {
  for(IntNode p=head; p!=null && p.val!=findMe; p=p.next) {}
  return p; // null == not found
}
```

The {}'s after the `for` loop say what they do: walk through the loop until you find the value or `null`, and for each item, do nothing.

The thing linked lists are terrible at, is jumping to a certain position. In an array, `A[5]` jumps straight to the fifth item. Finding the Nth item in a linked list requires a loop.

It's a little tricky since we're moving and counting and need to check for falling off the end (if the list isn't that long). Also, instead of returning just the Nth number, we'd rather return a pointer to that node:

```
IntNode getNodeAtIndex(IntNode head, int wantIndex) {
  int i=0;
  IntNode p=head;
  while(i<wantIndex && p!=null) { p=p.next; i++; }
  return p;
  // NOTE: if index was too big, this returns null, which is the right answer
}
```

We'd use it like `IntNode pp = getNodeAtIndex(L1,4);`. To get the value, we'd use `pp.val`.

Counting how many 7's are in a linked-list could be another "check every box" loop. But there's a much more fun way.

The plan is to use `findNodeWithVal` over and over, always starting 1 past where the last seven was, until we run out of list:

```
// count how many 7's there are in L1:
int count=0;
IntNode p=L1; // start looking at start of the list
while(p!=null) {
  p=findNodeWithVal(p, 7); // moves p to next 7 node, or null
  if(p!=null) { // found one
    count++; // count it
    p=p.next; // next search starts at next node
  }
}
```

This is a loop within a loop: `findNodeWithVal` skips to one 7, and the loop we wrote makes it keep finding more. We've done this before with arrays. With them, we used `int i;` and `i+1` to go to the next. With a linked list the same thing is `p` and `p.next`.

Fun fact: if we forgot `p=p.next;` at the end, this would be an infinite loop, finding the same 7 over and over.

42.1.2 Insert/Remove

Everything above would have been easier with an array. Linked Lists big advantage is inserting and removing items.

With an array, almost every add/remove requires a loop to slide everything to fill the empty spot. With a linked list, we can easily insert and remove from anywhere, as long as we have a pointer to a nearby node.

To insert, we splice it in by changing 2 pointers Here's a picture adding 55 to the start of our old list:

```
L1  2--> 7--> 4--> 9-->null
  \ |
   55
```

If it bothers you that memory is really numbered and is more like a line, here's a more memory-accurate picture of how splicing in 55 might look:

```
   --------------------
  /                    \
L1  2--> 7--> 4--> 9<>  55
     \                /
      -----------------
```

Following those long arrows is a pain for us, but it's not work at all for the computer. Following any arrow is exactly as easy as any other.

The code is what you see in the picture: create a new node, point it to the former first item, make the head point to the new item. This takes the start of the list as a reference parameter (since the function needs to change it):

```
void insertToFront(ref IntNode head, int newVal) {
  IntNode newNode = new IntNode(); newNode.val=newVal; // make the node
  // hook it up:
  newNode.next=head; // new one points at old 1st item
  head=newNode; // make this first item
}
```

Calling `insertToFront(ref L1, 55);` would make our new picture.

Hooking it up is a little tricky. If we flipped the order of the last two lines, it wouldn't work (`head=newNode` would lose our only link to the list.)

Another way to hook it up would with an extra pointer:

```
// alternate hook-up code:
  IntNode oldFirst = head; // saved 1st item
  head=newNode; // point to new 1st item...
  newNode.next=oldFirst; // ...which points to old 1st item
}
```

One last thing to check – does this work inserting into an empty list? `L1` would start out as `null`. Then it would point to the 55, which would point to `null`, so it works (in the intro, when I wrote these were good practice with pointers, this is what I meant).

We can now finally create that 2, 7, 4, 9 list we've been using. Since we're adding to the front, we have to add them in reverse order:

```
IntNode L1=null;
insertToFront(9); // L1-->9
insertToFront(4); // L1-->4--9
insertToFront(7); // L1-->7-->4-->9
insertToFront(2); // L1-->2-->7-->4-->9
```

Inserting in any other position requires us to have the node before it. For example, inserting 15 after the 7 involves changing the `next` pointer from 7:

```
L1--> 2--> 7  4--> 9-->null
            \ |
             15
```

But besides that it's the same 2-pointer dance:

```
void insertAfterNode(IntNode p, int newVal) {
  IntNode newNode=new IntNode(); newNode.val=newVal;
  newNode.next=p.next; // new node points to the one after p
  p.next=newNode; // p points to the new node
}
```

`newNode.next = p.next` is probably a big "yikes!" To read it, `newNode.next=` means we're changing where 15 points. `=p.next` is the new target, which is the node after 7. The whole line means "make the 15 point to the 4" (no one has ever understood that on the first try, just so you know. I haven't shown you a single new rule, but this is in the advanced chapters for a reason.)

To use it, and this is pretty cool, we could run our function that finds the 7. It returns a pointer to that node, which is exactly what insert needs:

```
// add a 15 after the 1st 7:
IntNode p = findNodeWithVal(L1, 7);
if(p!=null) insertAfterNode(p, 15);
```

For fun, here's a hacky way to add 15 to the end of the list, using our previous functions. Find the length, subtract 1, find that node, and insert after it. We'd never do this for real, but it's a nice exercise:

```
// insert 15 to end of L1:
int len=getLength(L1);
if(len==0) insertToFront(ref L1, 15);
else {
  IntNode last=getNodeAtIndex(len-1);
  insertAfterNode(last, 15);
}
```

This is still messing around: we could write a function that does every step by hand:

```
void insertAtEnd(ref IntNode head, int newVal) {
  // pre-make the new node:
  IntNode newNode=new IntNode(); newNode.val=newVal;
  newNode.next=null; // since it's the last item

  if(head==null) { head=newNode; return; }
  // loop stops where we're _about_ to go off the edge:
  IntNode p=head;
  while(p.next!=null) // <- if NEXT item is null. Tricky
    p=p.next;
  // p is last node. Add us after it:
  p.next=nn;
}
```

`while(p.next!=null)` is one of those extra-sneaky pointer linked-list tricks. When the next item is `null`, it means we're on the last item.

Another trick for finding the last item is adding a trailing variable. Let `p` fall of the edge, as normal:

```
// find last item using a trailer:
IntNode p=head;
IntNode pPrev=null; // trails behind p by 1 node
while(p!=null) {
  pPrev=p; // old value of p, just before moving it
  p=p.next;
}
// now p is off edge and pPrev is last node
```

You may be thinking that the thing before `i` in an array is `i-1`, which is so much easier. But all of this extra work with linked lists is worth it for the speed if we need to frequently insert and remove from anywhere.

42.1.3 Removing

Removing an item is like inserting it. We just reroute the pointer around the item to remove. Like inserting, the first item is a special case, since we have to change `L1`.

Here's what it looks like to remove the first item of of 2, 7, 4, 9:

```
L1    2--> 7--> 4--> 9-->null
  \       /
   -------
```

It looks odd that 2 still points to 7. We could set it to `null`, but it won't matter. The key things are the list now starts with 7, then 4 then 9; and there's no way to get to 2 anymore. Eventually the garbage collector will remove it.

The code looks like this:

```
void removeFirst(ref IntNode head) {
  if(head==null) return; // there isn't a 1st item
  head=head.next;
}
```

As a check: if there's only 1 item, this would make `head` point to `null`, which is correct.

Removing anything else requires the node in front of it. We again route the pointer around

```
void removeAfter(IntNode beforeNode) {
  if(beforeNode.next==null) return; // nothing after us to remove
  beforeNode.next = beforeNode.next.next;
}
```

`beforeNode.next.next` isn't a joke, or magic. It's the pointer to 2 nodes after you. Read it as if it had parens: `(beforeNode.next).next` If `beforeNode` is the 2, it jumps to the 7, then checks the `next` in there, which is an arrow to the 2.

But, again, yikes! Experienced coders are able to sight-read this, but it takes lots of practice.

This last one is a loop to remove all negative numbers from a linked list. Since we can only remove the next number, the loop checks whether the next number is negative. As usual, removing the first is a special case:

```
void removeAllNegatives(ref IntNode head) {
  if(head==null) return; // no items in list
  // removing 1st is special case, since we have to change head:
  // also, there might be several negatives at the start, so we need a loop!!:
```

```
  while(head.val<0) {
    head=head.next; // skip past = remove
    if(head==null) return; // whole list was negative numbers
  }
  // check non-first items, looking 1 ahead:
  IntNode p=head; // NOTE: we know this is not null
  while(p.next!=null) { // while not last item
    if(p.next.val<0) p.next=p.next.next; // cut out next item
    else p=p.next; // move normally to next item
  }
}
```

There are so many way this could go wrong. But it's typical linked-list pointer math.

The sneakist part is that if we remove something, we *don't* move forward. The removal puts a fresh item in front of us. Also moving would jump past it – we'd have a bug when there were 2 negative numbers in a row.

42.2 Misc

For testing, it would be nice to make a linked list with 0 to 9. This does it:

```
IntNode makeSeqList(int len) { // list from 0 to len-1
  IntNode head=null;
  for(int i=len-1;i>=0;i--) // backwards, since adding to front
    insertToFront(ref head, i); // inserts 9, 8 ... 0
  return head;
}
```

`IntNode L2 = makeSeqList(10);` runs it.

It might also be nice to convert an array into a linked list. We can go through the array backwards, adding each to the front:

```
IntNode arrayToLinkList(int[] A) {
  IntNode head=null;
  for(int i=A.Length-1; i>=0; i--)
    insertToFront(ref head, A[i]);
  return head;
}
```

`IntNode L3=arrayToLinkList(new int[]{2,7,4,9})` would run it (or use an array you've already made).

Suppose we didn't know adding to the front is fast and the end is slow. We could write array-to-list this way, which seems simpler:

```
IntNode arrayToLinkList2(int[] A) {
  IntNode head=null;
  for(int i=0; i<A.Length; i++) // simple forward array loop
    insertAtEnd(ref head, A[i]); // gah -- this is another loop
  return head;
}
```

If `A` has 100 items, which isn't big at all, this takes 1+2+3 ... + 100 steps – – roughly 5,000, instead of only 100 if we used front-insert.

Another plan, which isn't as good but seems more obvious, is to make a Linked List of empty nodes, then copy `A` into them:

```
IntNode arrayToLinkList3(int[] A) {
  // make list with all 0's, same length as A:
  IntList head=null;
  for(int i=0;i<A.Length;i++) insertToFront(ref head, 0);
  // now copy values:
  IntNode p=head; int i=0; // march these side-by-side
  while(i<A.Length) {
    p.val=A[i];
    i++; p=p.next; // 1 step ahead in both
  }
}
```

Reversing a linked list is another one that has a nice way if we think about linked lists, and some clumsy ways if we try to copy the array way of thinking.

Array thinking is to swap values from the first and last, then second and second-to-last, as so on:

```
void reverseArrayStyle(IntNode head) {
  int len=getLength(head);
  // position of the 2 to swap. Move these inward, together:
  int i1=0, i2=len-1;
  while(i1<i2) {
    IntNode n1=getNodeAtIndex(head, i1); // <- loop
    IntNode n2=getNodeAtIndex(head, i2); // ditto
    // standard swap:
    int temp=n1.val; n1.val=n2.val; n2.val=temp;
    i1++; i2--;
  }
}
```

That works, and the logic is clear – we're used to it from arrays. But it's a nested loop. 10,000 steps to reverse a length 100 list.

The much faster version uses linked-list thinking to rearrange the nodes (which is impossible for an array). Remove nodes from the front, and add it to the front of a new list. The new one is backwards, because that's how insert-to-front works:

```
void reverse(ref IntNode head) { // head will change
  IntNode R=null; // temp holder for reverse list
  IntNode p=head; // loop variable
  while(p!=null) {
    IntNode savedNext=p.next;
    p.next=R; R=p; // insert p into the front of R
    p=savedNext; // next in the starting array
  }
  head=R; // change original list pointer to fixed list
}
```

That's another linked-list loop which is much trickier than how short it is.

If you recall, we often don't like functions that change us. We prefer ones that return a copy with things the way we wanted. This returns a copy of `L`, but reversed (this is our first list-copying function):

```
IntNode makeReversedCopy(IntNode L) {
  IntNode A=null; // answer (L reversed)
  for(IntNode p=head; p!=null; p=p.next) // standard list loop
    insertToFront(ref A, p.val);
  return A;
}
```

Another list-y plan is to reverse every arrow. We'll go through front-to-back, using a trailing pointer, flipping arrows as we go (the arrow from `prev` to `p` is flipped to go from `p` back to `prev`). This isn't as good, and is too complicated:

```
void reverse3(ref IntNode head) {
  IntNode p=head; // loop variable
  IntNode prev=null; // trails behind p
  while(p!=null) {
    IntNode savedNext=p.next; // save next node
    p.next=prev; // switch p to point to the node behind it
    // now move the loop ahead a node:
    prev=p;
    p=savedNext;
  }
  head=prev; // when p is off end, prev is last node
}
```

This seems harder to read than the insert-to-front method, and isn't any faster. But it's fine for an exercise.

We can be more extreme with tearing apart a linked list. For example, we can split the list into even/odd:

```
IntNode odds=null, evens=null; // heads of 2 lists
IntNode p=L1;  // L1 is the original list
for(p!=null) {
  IntNode savedNext=p.next;
  if(p.val%2==0) { p.next=evens; evens=p; }
  else  { p.next=odds; odds=p; }
  p=savedNext;
}
// problem: both lists were backwards, since we front-added. Fix it:
reverse2(ref evens); reverse2(ref odds);
```

42.3 Special cases

You may have noticed that about half of each chunk of code is dealing with the list being empty, and handling the first node, which is a special case.

That's typical of lots of things.

The trick is writing a good function is ignoring special cases at first. For these, assume you've for a nice, long list and pretend the loop is about halfway through. Write code to keep going from there.

Then think about to start the loop. Then worry about empty lists, only length 1 lists and any other funny stuff you can think of.

42.4 Doubly-linked lists

The final form of a real linked-list is giving each node an extra pointer to the node in front of it. This is known as a Double-Linked List.

We usually call the second pointer `prev`, for *previous*. The new Node class looks like this:

```
class IntNode {
  public int val;
  public IntNode prev; // node in front of us. null==1st node
  public IntNode next;
}
```

Here's the new picture. The top row is `next`, the bottom is `prev`. Since we can walk backwards now, we also save a pointer to the end, named `tail`:

```
head-->| |-->| |-->| |-->| |-->| |-->null
       |2|   |7|   |4|   |9|   |3|
null<--| |<--| |<--| |<--| |<--| |<--tail
```

An advantage is being able to remove a Node based on its node pointer. Another is being able to go through the list backwards, and a minor one is being able to insert to either side of a Node.

But this won't change the basic linked list nature: insert/remove is fast; but searching for item N is slow.

We also like to make a nice class to hold the head and tail, plus the length, since why not. Then we can write everything as member functions. The new linked list class:

```
class IntList {
  public IntNode head=null, tail=null;
  public int len=0; // member functions need to keep this updated
  // insert, remove ... as member functions:
  // remember they can all see head, tail and len
}
```

Now we use `IntList L1 = new IntList();` to make one, and `L1.frontAdd(6);` to insert a new item.

Here's the new add-to-front member function. Setting up both forward pointers is the same; and it also need to set up the two backwards pointers:

```
public void frontAdd(int newVal) {
  IntNode newNode=new IntNode(); newNode.val=newVal;
  newNode.next=head; newNode.prev=null;
  head=newNode;
  // the old first node needs to point back to us:
  if(len>0) newNode.next.prev=newNode;
  else tail=newNode;
  len++;
}
```

`newNode.next.prev=newNode;` is another brain-teaser. Reading it with parens: `(newNode.next).prev=`. It finds the node after the new one, and changes its `prev` pointer to aim back to the new one. This is one of the most confusing lines in pointer math.

Also notice how inserting to an empty list is a special case (there's no node to point backwards to the new one, but the main `tail` should point to it.)

Since we have the tail, we can easily add 1 new item to the back. The code is (obviously) the same as adding to the front, except reversed:

```
public void backAdd(int newVal) {
  // if list is empty, it's easier to reuse frontAdd:
  if(len==0) { frontAdd(newVal); return; }

  IntNode newNode=new IntNode(); newNode.val=newVal;
  newNode.prev=tail; newNode.next=null;
  tail=newNode;
  newNode.prev.next=newNode; // previous last node points to us
  len++;
}
```

Now, with confidence, we can use `L1.backAdd(7);`.

This next one is just fun. Finding an item at a particular index is still a slow loop, but we can start from the back, if that would be faster:

```
// this is a member function:
public IntNode nodeAtIndex(int index) {
  if(index<0 || index>=len) return null; // off edge
  IntNode p=null;
  if(index<len/2) { // 1st half - search from front:
    p=head; // this is the same loop as before:
    for(int i=0;i<index;i++) p=p.next;
  }
  else { // search from back:
     p=tail;
    for(int i=len-1;i>index;i--) p=p.prev; // backwards using prev pointers
  }
  return p;
}
```

Hopefully the indexing for the backwards walk looks OK. It's using the actual index of that item, which is why it subtracts. If we want to find item 36 in a length 50 list, the loop counts backwards from 49 until it hits 36.

Now that we know both nodes around us, we can remove a Node we know about (none of that needing the previous item nonsense, from before.)

To cut out the 4 in this picture, we route the `next` arrow around it, as usual, and also the `prev` arrow. Also the same as before, the arrows from 4 don't matter, since we can't get to 4 anymore:

```
                 ---->--
                /       \
head-->| |-->| |   | |-->| |-->| |-->null
       |2|   |7|   |4|   |9|   |3|
null<--| |<--| |<--| |   | |<--| |<--tail
                \       /
                 --<----
```

Suppose `nn` is pointing to the 4, these lines change the arrows to cut it out:

```
nn.prev.next=nn.next; // the node behind us points to node past us
nn.next.prev=nn.prev; // node past us points to node behind us
```

They are both different versions of the hardest thing about pointers.

The real remove member function needs to worry about special cases, and adjusting the `len` variable:

```
public void removeNode(IntNode p) {
  if(p==null) return; // may as well check this
  // node in front of us goes around. Special case if first item:
  if(p!=head) p.prev.next=p.next;
  else head=p.next;
  // node after us goes around. Special case if last item:
  if(p!=tail) p.next.prev=p.prev;
  else tail=p.prev;
  len--;
}
```

As I wrote, doubly-linked lists let us insert after or before a Node. Here's after:

```
public void insertAfter(IntNode p, int newVal) {
  if(p==tail) { backAdd(newVal); return; } // after p is the end

  IntNode nn=new IntNode(); nn.val=newVal;
  nn.next=p.next; nn.prev=p; // our arrows
  p.next=nn;
  nn.next.prev=nn;
  len++;
}
```

To write `insertBefore` we can reuse `insertAfter`:

```
public void insertBefore(IntNode p, int newVal) {
  if(p==head) { frontAdd(newVal); return; }
  insertAfter(p.prev, newVal); // before this == after previous
}
```

I'm skipping frontRemove and backRemove. They're more of the same. Searching for a particular item is also no change.

42.5 An any-type linked-list, using templates

If you remember, the built-in "nice array" looks like `List<int> L=new List<int>();`. using templates. This section doesn't have any new linked-list stuff in it – only using templates to make them nicer.

Our Node class now holds type `T`:

```
class Node<T> {
  public T val; // T is whatever type the list holds
  public Node<T> next, prev;
}
```

Remember the `<T>` in `Node<T>` means we have to supply a type, and the `T val;` inside says to replace `T` with int or string or whatever they supplied.

The base list class also gains a `<T>`. Notice how it uses `T` to define the Node's it uses:

```
class LinkList<T> {
  public Node<T> head=null, tail=null;
  int len=0;

  public frontAdd(T newVal) {
    Node<T> nn = new Node<T>(); nn.val=newVal;
    nn.next=head; head=nn;
    ...
  }
}
```

We'd use `LinkList<string> L1 = new LinkList<string>();`. Then `L1.frontAdd("frog");`.
Member function `frontAdd` gets to use `(T newVal)` as the input since it remembers whatever `<T>` we used when we created `L1`. That way `L2.frontAdd(6.4f);` could be legal, since it uses `L2`'s value for `<T>`. This is all template sneakiness.

Everything else is the same, except they use `T`'s instead of `int`'s. I think this is pretty neat, since we can hand-code, without too much trouble, an actually useful linked list class.

42.6 Built in Linked List

As you'd guess, `C#` has a built-in Linked List class. The variables are all private, but we use the functions the same way. For example:

```
LinkedList<string> L1 = new LinkedList<string>(); // empty
// add to front or back:
```

```
L1.AddFirst("bb"); L1.AddFirst("aa");
L1.AddLast("ccc"); // aa, bb, ccc

// searching returns a NODE pointer:
LinkedListNode<string> nn = L1.Find("bb");
string w = nn.Value; // Value instead of just val

// can add items before or after a node:
L1.AddBefore(nn,"a2"); // aa, a2, bb, ccc
L1.AddAfter(nn,"a4"); // aa, a2, bb, a4, ccc

// remove takes a node as input:
L1.Remove(nn); // aa, a2, a4, ccc
```

You can't see the head or tail directly, but you can use `L1.First` and `L1.Last` to get those pointers (to Nodes). Those names are kind of interesting. Someone who took a course would prefer `head` and `tail`, since it reminds them it's a linked list. But that would confuse people who didn't take a course.

Since everything is private, you can't directly change the pointers, but, for example, you can still tear a list apart using built-in commands. This pulls all the odd nodes out of `L2`, moving them into another one:

```
LinkedList<int> Odds = new LinkedList<int>();
LinkedListNode<int> p = L2.First; // loop pointer
while(p!=null) {
  LinkedListNode<int> savedNext = p.Next;
  if(p.Value%2==1) {
    L2.Remove(p);
    Odds.AddLast(p); // has a version to add a node
  }
  p=savedNext;
}
```

This sort of thing is rare. I like it since you can use the official linked-lists with simple Add and Remove commands. But if you know how they really work, the system gives you access to the internal nodes and lets you shift them around.

42.7 Array implementation of linked lists

This section isn't very useful, but it's more practice for thinking about how linked lists work and more playing with array indexes. It is a real thing, but it's only used in very specific circumstances.

The trick is, we pre-make an array with all of nodes we'll ever use. Each `next` pointer will be the index of the next item. The `head` pointer is the index of the first item.

This negates a big feature of linked lists – we won't be able to grow it all we want. But we can still do the other nice stuff: insert and remove from anywhere.

The nodes will look like this:

```
struct Node {
  public string val; // using strings to be less confusing
  public int next; // index in array of next item. 0 to length-1
}
```

Then we'd make `Node[] AllNodes = new Node[10];`. In our minds, this is ten unused nodes.

Here's a picture of a list with: aaa, bbb, ccc, ddd, eee:

```
L1: 3

   0     1     2     3     4     5     6     7     8
 ----- ----- ----- ----- ----- ----- ----- ----- -----
| ddd |     | ccc | aaa |     | bbb |     | eee |     |
|  7  |     |  0  |  5  |     |  2  |     | -1  |     |
 ----- ----- ----- ----- ----- ----- ----- ----- -----
```

Notice how our head, `L3`, points to box 3, with `aaa`. That box points to box 5, with `bbb`. At the end, box 7 uses -1 for null.

A loop to print them out looks a lot like a standard pointer loop. `p` will jump through 3, 5, 2, 0, 7 then -1:

```
for(int p=L1; p!=-1; p=AllNodes[p].next)
  print(AllNodes[p].val);
```

Inserting to the front is similar. Pretend we have a function to get an unused node:

```
int nn = getUnusedIndex(); // 0-9 of a free node
AllNodes[nn].next=L1; L1=nn;
```

Deleting the node after `p` should also look similar:

```
int p2 = AllNodes[p].next;
int p3 = AllNodes[AllNodes[p].next].next; // ugg
AllNode[p].next=p3; // skip past
AllNodes[p2].next=-999; // mark as unused
```

These examples are singly linked, but it works doubly-linked as well.

Chapter 43

Big-O notation

In a previous section about making a faster program I wrote that none of the obvious tricks worked very well. We can mess around rearranging `if`s and equations and, if we turn our program into an incomprehensible bug-ridden mess and are very lucky, maybe we get a x2 speed-up.

Profiling is just more of that same thing. All it does is help you use those little tricks, like telling you which functions get used the most. Or, after you rearrange some `if`'s it will tell you the function is *slower* so you need to keep trying. In a game, it will often tell you the program is plenty fast, but the graphics card is being swamped. You just need to use pictures with a smaller pixel-count.

Profilers are great, especially for code written by a lot of people which you suspect is sloppily assembled. If you're going to do that stuff, go ahead and use one. But it's still checking the couch for spare change compare to the real speed-up trick.

The one thing that gives a huge speed-up is avoiding extra loops. If you have a loop that runs 1,000 times, and you realize you can do the same thing without one, that's about a 1,000 times speed-up.

Avoiding unnecessary loops is the major trick for worrying about speed.

Nothing in this chapter will let you do things you couldn't do before, or make your program look nicer or anything else useful. It just does one thing, which is make it faster. But if you're lucky, it will make it *a lot* faster.

The first section explains that idea more. The second part has examples of using it to classify common array loops; then examples of removing loops by using a better plan. Then a little about how the loops are different for a linked-list.

The section after that introduces the formal way we write big-O notation. It's useful to know because it's the way computer manuals list speeds. Then there's more examples, data structures and the last section is a summary of how we use this thinking in general program writing.

43.1 Loop counting logic

The math and motivation are long and boring, so I'm going to skip ahead to the end results and what we do about it. I'll go over the explanation in a section at the end, and hopefully the examples will help. Just keep in mind there will be a few places where I'm just saying something is true without really showing why.

To help speed up programs, we classify every function by the worst nested loop it has. We don't worry about how many times the loops run, or how many lines are in them. We list them as only: not a loop, single loop, nested loop, triple loop and so on.

Just to make sure, here's a sample triple loop, using 3 arrays:

```
for(int i=0;i<A.Length;i++)
  for(int j=0;j<B.Length;j++) {
    count1++;
    for(int k=0;k<C.Length;k++)
      count2++;
    }
```

Suppose every array is 1,000 long. Loop one runs 1,000 times, loop two runs a million times (`count1` is a million when this ends), and it runs `count2++` a billion times. Nested loops get very big, very quickly.

That's why we simplify so much. In general, any single loop is much worse than any non-loop, any nested loop is much worse than any single loop, and so on.

After we label a function with the nested level, we try to reduce it. We don't waste time trying to make the loop run fewer times, or do less work each time. All we do is try to think of a plan that uses less nesting. That's where we get the big x100 speed-ups.

If we have a single loop, we try to think of a way to not use a loop. If we have a nested loop, we try to think of a way to do the same thing with only a single loop. And so on.

Again, I'm not really explaining the math or the thinking, but here are some comments why this plan sort of makes sense:

- This system says that non-loops are the simplest thing and all count the same, as one step. That's pretty much true. Compared to a big loop, the most complicated math and `if`s is insignificant. And we'll never get a decent x100 speed up simplifying `if`'s and math, like we can simplifying loops.

- If you speed up a single loop by running it fewer times, or having less lines inside, this system says it's still a single loop, so those speed-ups didn't matter. That's also pretty much true. Those are just x2 speed ups. Removing the loop would get you x100.
- A nested loop might run quickly on something small, and not be a problem at all. But sizes tend to grow and make it a problem. A nested loop on 12 cats might be only 144 steps. But as the pet store grows we'll have 100 cats and the loops shoot up to 10,000 steps.
- It works well for functions that take lists as inputs. Say the function is ranked as a nested loop. It's safe to call with very small lists. If we need to call it with a big list, in another loop, at least we know this is a super-slow triple loop.
- A function with two loops in a row doesn't seem to fit in our system, but it counts as a single loop. It's just twice as slow as one loop, which doesn't matter. We have to eliminate both to make it a non-loop (getting rid of just one is like a useless x2 speed-up.)
- Sometimes we have fixed, tiny loops, and they obviously don't count. For example, a 2D nested loop to set up a Tic-Tac-Tow board is always a mere 9 steps.

  It's usually pretty obvious. We have either very small loops that will never grow, or loops running 100+ times which will eventually get larger.

43.2 Integer array loops

The standard loops (or non-loops) on int arrays make good examples of how to apply the classifications.

Finding an item at a certain position in a list is a one step non-loop. We knew that, but I wanted to point out we don't always have to loop to do things with an array.

The first item is one step, `A[0]`, but finding the middle and last are three: `A[A.Length/2]` and `A[A.Length-1]`. We don't even care. It falls under the "basically 1 step" non-loop rule.

Counting how many times something occurs in an array is a single loop:

```
int howManyOf(int[] A, int countMe) {
  int count=0;
  for(int i=0;i<A.Length;i++) if(A[i]==countMe) count++;
  return count;
}
```

A funny thing is it takes longer depending how many there are (`count++;` takes a step.) We can ignore whether the inside runs 1 or 2 steps and just count it as a loop.

Checking whether something is in an array is even more variable. It might be the first item, finding it halfway is average, and if it's not there we always run the full loop:

```
bool contains(int[] A, int findMe) {
  bool found=false;
  for(int i=0;i<A.Length; i++) if(A[i]==findMe) { found=true; break; }
  return found;
}
```

This is really about the best, average and worst time a loop takes. Usually those are about the same – within half or double of each other. In this case the minimum of 1 or 2 steps is obviously very rare. The average is obviously a loop.
A way to look at it is the `break;` is like a x2 speed up trick. It's better than nothing, but it's still a loop.

Some loops average only 1 or 2 steps. They almost never run the full time. We'd probably count them as non-loops (the math can be tricky,) but those are rare and you'll know them when you see them.

Finding the smallest item is also variable speed But it still easily counts as a standard big-O loop:

```
int indexSmall=0;
for(int i=1;i<A.Length;i++)
  if(A[i]<A[indexSmall]) indexSmall=i;
```

Starting the loop at 1 is the logical thing to do, and also a tiny speed-up. But a loop that skips a few items still counts as a loop.

Checking if a loop has duplicates is our first nested loop. The standard way is for the inner loop to only check items *after* us, and to quit early if we find a match:

```
bool hasDuplicates(int[] A) {
  for(int i=0;i<A.Length-1;i++)
    // compare to everything after it:
    for(int j=i+1;j<A.Length;j++) if(A[i]==A[j]) return true;
  return false;
}
```

This is another example where we ignore small tricks. A "real" nested loop on 1,000 items runs a million steps. This runs only 1/2 a million if there are no duplicates, and half that (on average) if there is one. But a quarter million is still only 4 times faster than a million, and way worse than 1,000 for a single loop. We count it as a standard nested loop.

Finding the index of the most common item is another nested loop:

```
int mostCommonItem(int[] A) {
  int mostNum=-999; // best count so far
  int mostIndex=-1;
  for(int i=0;i<A.Length;i++) {
    int cur=A[i];
    int count=howManyOf(A, cur);
    if(count>mostNum) { mostIndex=cur; mostNum=count; }
  }
  return mostIndex
}
```

This is a single loop around `howManyOf`, which is a loop. So it's a nested loop.

Inserting and removing items from arrays requires sliding everything after it (either sliding in to fill the hole, or sliding out to make room.) The `List` class has this built-in.

`L.Add(4)` takes 1 step. It adds to the end, so there's nothing to slide down. `L.Insert(0,4)` adds to the front, so is a full loop to make room. `L.Insert(i,4)`, where `i` is just some position is a half-loop on average, which counts as a loop.

All-in-all, adding or removing from the back is a non-loop, and anywhere else counts as a loop (in theory, only playing around with things at most 10 from the end would count as non-loops, but no one ever does that).

Sorted Arrays

Suppose an array is sorted. Sometimes it's not too hard to keep it sorted as we go. Or maybe it won't take too long to sort it when we need to. If it's sorted, there are a few things we can do faster:

Finding the smallest and largest now take only 1 step (first item, last item.) This seems like cheating, since sorting obviously takes longer than a loop. But if it was already sorted, 1 step is pretty cool.

Checking for duplicates in a sorted list is only a single loop: check whether each item is the same as the one next to it. This loop starts at 1 and compares to the item before it:

```
bool hasDuplicatesSorted(int[] A) {
  for(int i=1;i<A.Length;i++) if(A[i]==A[i-1]) return true;
  return false;
}
```

We removed a loop, sort of. Duplicates on a size 1,000 list went from 250,000 steps to about 750 (not important, but about half if there are any, and the whole list when there aren't).

If you remember, finding an item in a sorted list can be done with a binary search, which is an odd cut-in-half loop. You find which half of the array your item should be in (which you can do since it's sorted,) then find which half of that half it should be in, and so on.

The important thing is it's much faster than a regular loop, but it's still sort of a loop. For 1,000 items it takes 10 steps. It takes 20 steps for a million length list. We'll make a new category for this, and call it a "cut-in-half" loop.

The end result is searching for an item went from about 500 steps to 10 steps. We almost removed a loop – 50 times faster is pretty impressive.

Just for fun, here's a binary search. There's a binary search in the recursion section, but we can write it as a loop. This picks a middle, figures out the half we should look in, and repeats until we're down to 1 or 0 items in our half:

```
bool isInSorted(int[] A, int findMe) {
  int start=0, end=A.Length-1;
  while(start<end) { // at least 2 things in it
    int mid=(start+end)/2;
    if(findMe<=A[mid]) end=mid;
    else start=mid+1;
  }
  if(start>end) return false; // 0 items
  return A[start]==findMe; // 1 item. Is it our number?
}
```

Finding the most common item goes from a nested loop to a single loop (being sorted means identical items are together. We can count how many there are of an item in the same loop that looks at each different item. The code for this is later).

Sorting

Sometimes your array just happens to be sorted. Often it's almost sorted and a few easy changes can make it completely sorted. Even if it's not, it's often faster to sort it, then check for whatever you wanted.

The easy sorts are nested loops: bubble sort, selection sort and insertion sort. But sorting is pretty important and over the decades we've figured out some faster, but complicated ways of doing it. The built-in sort, `Array.Sort(A);`, uses one of those.

Fast sorts are a nested loop where one is just a cut-in-half loop. That means sorts are just a little bit worse than a single loop. For a length 1,000 array, the cut-in-half loop runs 10 times, for only 10,000 steps total (much better than hundreds of thousands for a nested sort.)

If you want to look them up, the fast sorts are named merge sort, heap sort and quick sort. They all work differently, but all use a regular loop nested with a cut-in-half loop.

This means for anything that takes a real nested loop – duplicates or finding the most common – we can usually do it a few hundred times faster by sorting first, than using a single loop.

Suppose we want to check whether two arrays are the same. If would be a nested loop. But we can go faster by sorting both, then using a single loop to compare items side-by-side.

43.2.1 Fun with bad loops

We know an easy way to make a reversed copy of a list is adding every item to the front of a new one, like this:

```
// copy L reversed into L2:
List<int> L2 = new List<int>();
for(int i=0; i<L.Count; i++)
  L2.Insert(0, L[i]);
```

But we know inserting to the front is a loop. This whole thing is a nested loop – a whooping million steps if `L` is length 1,000.

This version is a single loop:

```
List<int> L2 = new List<int>();
for(int i=L.Count-1; i>=0; i--)
  L2.Add(L[i]); // add to end is only one step
```

Removing all negative items looks like a simple loop. It goes back to front so that the remove-slide can't mess things up:

```
for(int i=B.Count-1; i>=0; i--)
  if(B[i]<0) B.RemoveAt(i);
```

But `RemoveAt` anywhere except the end, is a loop. We can have negative items all over, so the whole thing is a nested loop (assuming a decent fraction of them are negative.)

It so happens that if you know you want multiple removes, you can redo it to use only a single loop. There's a built-in `RemoveAll` which is a single loop. It takes a function input for what to remove:

```
L.RemoveAll( (n)=>n<0 );
```

The loop runs through the array from start to back. When it finds something to remove, it starts sliding items 1 to the left, to fill the hole. But it also keeps checking. When it finds another item to remove, it increases the slide to 2:

```
int removedCount=0; // also how far to slide
// assume len is how much of the array we're currently using
for(int i=0; i<len; i++) {
  if(A[i]<0) { removedCount++; } // don't slide this down
  else A[i-removedCount]=A[i];
}
len-=removedCount;
```

In general, this is common: using a "do it once" method, over and over, can often be improved if we think about doing it to the whole list at once.

A standing-in-line type list is common – new items go in the back and are processed when they get to the front. It's often called a queue or first-in-first-out (FIFO).

Doing that with an array always has a loop. Depending on which way the line goes, we need to either add or remove from the front. With a Linked List we can use `L.AddLast(6);` and `n=L.First.Value` and `L.RemoveFirst()`, which all count as a single step.

43.3 Formal math

We really measure speed using simplified formulas, not loops. For example, a nested loop over a size n array takes $n * n$ steps. So we write it as $O(n^2)$. Once you get used to it, that's a shorter and more accurate way.

The rules are for writing Big-Oh notation:

- A capital O and parens goes around the whole thing. Besides standing for Big-Oh, it means "on the order of". $O(n^2)$ means: about n-squared steps.
- If there's no loop, write $O(1)$. It stands for "basically one step."
- Pick a variable to stand for the size of the input, usually n. This is often the size of the array.
- Write an equation using n with no constants and only the highest term.

The terms for what we've been using, in increasing order of slowness:

$O(1)$ - not a loop
$O(log(n))$ - cut in half loop
$O(n)$ - single loop
$O(n * log(n))$ - a fast sorting loop (cut-in-half around a normal loop)
$O(n^2)$ - nested loop

$O(n^3)$ - triple nested loop

These are the terms you'll see listed in manuals. Inserting to the front of a `List` is $O(n)$, meaning it's a loop over all items. Inserting to the front of a Linked List is $O(1)$, meaning it's a few lines, no matter how long the list is.

If you're interested, Big-Oh started as a way to simplify the actual formula for how many steps. A function might do some work taking 50 steps, then run a loop over an array with 10 steps inside, then run a nested loop where the middle skips part. The actual formula is $50 + 10n + 0.5n^2$.

Big-Oh says we may as well reduce that to $O(n^2)$. That function basically takes nested loop time.

Using these terms: suppose we sped-up a loop from $6n$ to only $3n$. We know that's minor, since it's still a loop. The technical way is to say that is they're both $O(n)$; you didn't reduce the Big-Oh.

This is another detail if you're interested in the math: $O(n^{1.9})$ is also a category. There are a few horribly complicated algorithms with strange running times like that (with even stranger math proving it.)

If you graph $n^{1.9}$ and n^2, the second will pull further and further ahead. Eventually $n^{1.9}$ will be a thousand times faster. So that's great. But it might not happen until your arrays are a billion items or more.

The actual definition of Big-Oh is about being much faster *once the arrays get big enough*. How big is that? It depends. For most simple stuff, like going from a nested loop to a single loop, a few hundred items is enough to see a big speed-up.

Put another way, these rules are only useful with large lists, that you expect to grow. If method A takes 1,000 steps and method B takes 2,000, A is faster.

If A is a triple nested loop (and B isn't,) it's still faster. Big-Oh is only about asking yourself: "am I running a test array with 20 items? Will the real one have 200? If so, do I have nested loops that will balloon up?"

The last funny math note: sometimes you'll see two variables. Like a function taking 2 arrays might be $O(m * n)$. If you know one array will have 4 items at most, that's like $O(n)$. If both are about the same size, that's like $O(n^2)$.

43.4 More tricks and exceptions

Obviously, not all loops are problems, and not all problems are loops. We're really looking for things that take more and more time as the input grows. A big array loop is just the most common way that happens.

Some notes:

Small, fixed-sized loops don't count. For example, suppose you're on a grid and want to check the 5x5 area near you. A nice way to do it is with this nested loop:

```
// get sides of 5x5 area around me (me -2 and +2) not off edge:
int x1 = Mathf.Max(myX-2,0), x2=Mathf.Min(myX+2,width-1);
int y1 = Mathf.Max(myY-2,0), y2=Mathf.Min(myY+2,height-1);
for(int x=x1;x<=x2;x++)
  for(int y=y1;y<=y2;y++) {
    if(x==myX && y=myY) continue; // skip space I'm on
    // check Grid[x][y] ...
  }
```

This runs at most 25 times. If the grid grows to 5,000 by 5,000 – this still runs 25 times. Since we're really trying to estimate the time it takes, we call this $O(1)$.

In Unity3D, `GetComponent` and `transform.Find` are also often tiny fixed-sized loops that should count as $O(1)$. GetComponent loops through all of your components until it finds the one you wanted. But I've never had a gameObject with more than six components. Realistically, it's 6 steps and counts as $O(1)$.

Likewise I often use `transform.Find` when I know I have two children and will never have any more (for example, a label with children: Text and Background.) It loops through them, but it's 2 steps, tops, so we'd call it $O(1)$.

`GameObject.Find` is more like a real $O(n)$ loop. It searches through all gameObjects. That's probably a big list, which will likely grow as you add a bigger map with more stuff in it. So it's a classic "it seems fast enough now, on this small list, but $O(n)$ is a warning it won't stay fast."

Some people like to use empty gameObjects as folders. Maybe all pick-ups go into an empty named `pickupHolder`. We can have lots of pick-ups, more and more as we grow the game. So now `transform.Find("healthPack5")` is like a real $O(n)$ function.

To see why that matters, suppose we do something with every health pickup, in a loop using Transform.Find:

```
for(int i=0;i<maxPacks;i++) {
  string pkName = "healthPack"+i;
  Transform tt = pickupHolder.Find(pkName); // <-this is a big loop
  // do something with tt
}
```

We just accidentally wrote a slow nested loop. If we have 100 pick-upable items on the map, this takes about 5,000 steps. The purpose of Big-Oh is to help us spot that – an $O(n)$ function call in a loop. Yikes!

We could rewrite using a "go through children in order" loop:

```
foreach(Transform t in pickupHolder) // Unity's each child shortcut
  // grab name, reject non-healthpacks:
  string w=t.name;
  if(w.Substr(0,10)!="healthPack") continue;
  int packNum = ...
    ...
}
```

This is messier to write, but it's only a few hundred steps, compared to thousands. Made possible through big-Oh thinking.

Count the work, not the loops. Sometimes a nested loop just goes through the array once, so is really $O(n)$. This is the loop I promised to find the most common item in a sorted list:

```
int maxRunLen=-1, maxRunVal=-999;
for(int i=0;i<A.Length;) { // will increase i inside the loop
  int runLen=1; int val=A[i]; // set-up to count this fresh number
  i++;
  // count the duplicates, check for a new winner:
  while(i<A.Length && A[i]==val) { i++; runLen++; }
  if(runLen>maxRunLen) { maxRunLen=runLen; maxRunVal=val; }
}
```

The form is a nested loop. But if you trace it, both loops work together to push `i` one time through the array. It's really doing $O(n)$ work.

We've already seen the cut-in-half loop. On a size million array it runs 20 times, so we don't count that as a normal loop.

And it's possible to write a single loop that runs like two nested loops (at the end it would hand-move `i` and `j` the same way a nested loop would).

Funny loops like those are rare, and you can usually spot them easily enough by how they move the loop variable in odd ways.

Non-array loops count. Sometimes you loop over numbers. Suppose you check every angle from 0 to 180, going by 0.1. That's about 2,000 steps, more if you have to reduce the step for accuracy. Two of those could be nested, taking roughly 4 million steps. It seems fair to call that $O(n^2)$, and to try to think of a less loop-using way.

Sometimes you have to pick `n`. If a loop counts up to a number, `n` is the number. But if a loop checks each digit of a number, `n` is just the number of digits, which isn't that large (but a double or triple-nested loop can still get big, fast.)

For grids, it might make sense to have n be the length of a side (especially if it's mostly square,) or be the number of squares (which is better for long, narrow boards).

A function that touches every square once would be $O(n^2)$ when n is side-length, and $O(n)$ if n is how many actual squares.

If seems funny that we can just pick something, but we're only using it for comparison. Whatever we pick for n, we'll use that to measure every plan.

Recursion can be just about any Big-Oh. This function uses recursion to walk through an array, one step at a time, so it's $O(n)$:

```
bool allPositive(int[] A, int startIndex=0) {
  if(startIndex>=A.Length) return true; // made it to the end w/o quiting
  if(A[startIndex]<=0) return false; // found a negative
  return allPositive(A, startIndex+1); // keep looking from next item
}
```

Most recursive functions that call themselves only once are $O(n)$. But real recursive functions call themselves twice or more.

You can often logic those out: Flood fill hits every square once. A recursive tree search hits every child once; and sometimes you know the tree has at most 6 things, so the recursive function is essentially $O(1)$.

But recursion also can blast out of control with $O(2^n)$. That's exponential time (which is as bad as it sounds – worse than a mega-nested loop.)

The end result, which we already knew: recursion is a very tough topic.

43.5 Other data structures

Computer science has several ways to store data which are only useful because they have interesting Big-Oh's.

Many of these are built into things you already use, but it might be fun just to see the various things we use:

- B-trees. These are always sorted lists where search, insert and delete are all $O(log(n))$. The B stands for binary: they're really trees where everyone has 2 children, making a big pyramid. As a nice feature, they can also be stored in an array.

- K-trees, red/black trees. These are like B-trees. A red/black tree can have 1 or 2 children – it doesn't have to be a perfect pyramid, but has to be close. K-trees can have up to 3, 5 or 7 children (the K stands for how many they have.)

  Like other trees, most things are $O(log(n))$. Different trees are a little better or worse depending if you insert and remove more, or search more.

- Heaps. There are trees made so you can easily add items with an "importance value" and always pull out the most important. Most things are $O(log(n))$. That's all they're good for, but they're very good at it.
- Hash tables. Search, insert and delete are $O(1)$! (on average.) But everything else is terrible and it takes a little more space.

  These are how Maps and Dictionaries are made (ex: `A["cow"]=6;`).

None of these do anything an array can't do. They just do some things faster (and other things worse.) There's an entire course studying these things, and learning the Big-Oh's for everything so you know which one to use when.

43.6 Overview

Altogether, including ideas from the previous section on efficiency:

- In many places you don't care about how fast the code runs. For example, things that rarely happen, that purposely have a time-delay, or things we're just trying out.
- Many things run fast enough. A 2D tap-tap-tap puzzle game will run fine with triple-nested loops. If removing a loop would make it hard to read, it's not worth a X100 speed-up.

Assuming we're in the section where we think speed might matter:

- As you plan, think about arrays vs. linked lists, based on the big-O's of whatever you want to do. Generally, if you need to jump around a lot, you have to use an array. If you need to insert/delete lots from anywhere except the end, you need a linked-list.

  When you see a new language, look for the nice built-in array and linked-list types (in `C# List<int>` is really an array).
- As you write, think a little about the big-Oh's of things. If you have 50 monsters, each running an array loop which calls an $O(n)$ function, that's like $O(n^3)$. It might be worth trying to get that down.
- Find the worst big-Oh. It probably sucks up 95% of the time. There's no point speeding up anything else.
- Think about sorting lists, then using the faster "only works if sorted" routines. If you have any nested loops, this is usually at least a x100 speed-up for that part.

  A little harder, you can add new items to the list in a way to keep it sorted. If you use the list a lot, and don't add or remove much, this can be fast.
- After you've gotten all the big-O speed-ups and want to look for small ones, use big-O estimates for where to start. For example, speed up the insides of nested loops before singles.

43.7 Numbers

This is the section with the numbers. Mostly explaining why measuring speeds is so fuzzy.

Adding two numbers might take 1 step, but adding floats takes longer than adding ints. Also, sometimes the computer can run two of your instructions at once. Different CPUs take different times for all of this.

`A[i]++;` might be 4 steps: look up `i`, jump to that spot in `A`, add 1, then save it. But they might not all take the same time, so it's about 4 steps.

Things like square root and trig functions "take a long time," but not really. They might take 10 or 20 steps each. A big, fat trig-using math expression might take 80 steps. If you can get rid of some trig you don't need, you might get it down to 62.

This is what I mean by non-loops not mattering. 80 to 62 isn't a big speed-up compared to x100 faster, and 80 is a small number compared to a few thousand.

If we want to accurately measure the smallest array loop, it might take 7 steps. Just moving the loop is `i++` (2 steps?) and `i<A.Length` (3 steps?) and then `A[i]=0;` is 2 or 3 steps? The hypothetical simple 100 length loop is 700 steps total, making the 80-step math seem even smaller.

If we add a second line inside the loop, that doesn't really double the time. It's more like 7 steps increasing to 9. Put another way, cutting 2 lines inside a loop to 1 is really only a 20% speed-up.

If we put our 80-step math equation in a 1,000 step loop, of course it matters. We're taking 80,000 steps total. But reducing the middle to 62 steps is still less than a x2 speed-up. Removing the loop is still the best thing.

A worse big-O isn't always larger, but will always *eventually* be much larger. Suppose we have a single loop with 50 steps in it, and a sneaky nested loop with only 1 step inside. The times look like this:

size	single	nested
20	1,000	400
50	2,500	2,500
100	5,000	10,000
200	10,000	40,000
500	25,000	250,000

Everything always charts out like this. More nested loops always eventually takes longer, then much longer, then much, much, much longer. The technical term for needing to get big enough is Asymptotic. As in "$O(n^2)$ is asymptotically much slower than $O(n)$".

If you only have small arrays, which will never get larger, you're probably not having speed issues anyway.