Chapter 28

Nested loops

Putting a loop in a loop (a *nested loop*) doesn't have new rules, but works in surprising ways, and there are some common tricks.

This chapter is mostly various nested loop examples, But I added two new loop rules: `break;` and `continue;`. They are odd, but commonly used and make writing loops easier.

The main trick with one loop inside the other is that the inner loop restarts each time. This nested loop below runs 48 times, printing every combination from (0,0) to (7,5):

```
// demo nested loop
for(int i=0; i<8; i++) {
  for(int j=0; j<6; j++) {
    print( "i="+i+" j="+j );
  }
}
```

If prints (0,0) then (0,1) up to (0,5), then restarts with (1,0) to (1,5), ending with (7,5)

`j` seems like a bad choice for the inside loop variable, since it looks so much like an `i`. But it's traditional. If you have good names for the loop variables, use them. If not, use `i`, and then `j` for an inside loop.

Sometimes it helps to rewrite as a simpler loop. Here's the middle loop as a while, where we can clearly see `j` resets to 0 each time:

```
// inner while loop:
for(int i=0; i<8; i++) {
  int j=0;
  while(j<6) {
    print( "i="+i+" j="+j );
```

```
    j++;
  }
}
```

28.1 As a grid

Those numbers look like grid coordinates, and that's a common way nested loops are used: rows and columns. Here's a program using the nested loops from above to make an 8 by 6 grid (it uses `Instantiate` and assumes you have a 1x1x1 Cube being copied, probably a prefab).

```
public GameObject blockPF; // link to a 1x1x1 Cube (or anything, really)

void Start() {
  Vector3 blockPos; blockPos.z=0;

  for(int col=0; col<8; col++) {
    for(int row=0; row<6; row++) {
      GameObject bb = Instantiate(blockPF);
      blockPos.x = -6 + 1.2f*col;
      blockPos.y = -3 + 1.2f*row;
      bb.transform.position = blockPos;
    }
  }
}
```

We already know that the inside runs 48 times, so this Instantiates 48 blocks.

In our minds, the loop's job is to make `row` and `col` be every combination from (0,0) (7,5). The body of the loop is really "make a block, positioned where `row` and `col` say."

The exact positioning numbers I used aren't important. Block (0,0) is at position (-6,-3), which is near the bottom-left corner of our screen. I spaced them `1.2f` apart, so we can see the gaps.

A fun thing for testing is to color or somehow change some of them, to see it better. Putting this in the loop makes the (0,0) Cube yellow, turns row 0 red, and column 0 green:

```
  // color some blocks, to get a feel:
  color cc = new Color(1,1,1); // white (if nothing else)
  if(row==0 && col==0) cc = new Color(1,1,0); // 00=yellow
  else if(row==0) cc = new Color(1,0,0); // row0=red
  else if(col==0) cc = new Color(0,1,0); // col0=green
  bb.GetComponent<Renderer>().material.color = cc;
```

We should see yellow in the lower-left corner, red along the bottom and green along the left side. So row 0 is the bottom, column 0 is the left.

I color things since it's easy to get positions mixed up. For example, this change, to the `blockPos.y` line, makes the same arrangement, but puts row 0 at the top:

```
// 8 wide, (0,0) upper-left
blockPos.y = 2 - 1.2f*row; // row 0 at 2.0, going down
```

Or we can flip how row&col change x&y. This would make it 6 wide and 8 tall (instead of 8 wide and 6 tall.) It's confusing, since the "rows" now run up and down:

```
// 6 wide, (0,0) lower-left:
blockPos.x = -6 + 1.2f*row; // row is x
blockPos.y = -3 + 1.2f*col; // column is y
```

These examples seem a little silly, but they point out what the nested loop does and doesn't do. It makes all the combinations of (0,0) (0,1) .... If the outer loop counts to 10 and the inner counts to 14, you get 140 items in a 10x14 pattern. But how you orient and position them, if at all, is up to you.

28.2 Pattern in pattern

A grid is a good picture of a loop – it's easy to imagine a loop touching each row, then the second loop making everything in that row. But a nested loop can be any pattern in a pattern.

Suppose we need 11 12 13, and then 21 22 23, then 31 32 33 ... up to 93. That's a pattern in a pattern (count by 10's, count 1,2,3 for each.) We can make it with a nested loop:

```
// loops to count 11 12 13,  21 22 23 ... :
for(int i=10; i<100; i+=10) { // 10, 20 ... 90
  for(int j=1; j<=3; j++) { // 1, 2, 3
    int num=i+j;
    print( num );
  }
}
```

It seems like `tens` might be a better name for the first loop variable, then `one`s for the second. I kept it as `i` and `j` since the pattern doesn't have to be 10's and 1's.

For this next one, I'd like a line of blocks repeating small, medium, large over and over, like a sawblade. I can do that with a nested loop – the inner makes the three sizes, the outer makes several groups:

```
// repeating small, medium, large blocks:
Vector3 pos = new Vector3(-6, 0, 0); // left edge of next block
for(int group=0; group<4; group++) {
  for(int sz=0; sz<3; sz++) { // 3 different sized blocks
    float bSz = 0.3f+sz*0.4; // equation for the actual size
    GameObject gg = Instantiate(blockPF);
    gg.transform.localScale = new Vector3(bSz, bSz, 1);

    pos.x += bSz/2; // move over 1/2 a block, to center
    gg.transform.position = pos;
    pos.x += bSz/2; // now move other half, to right edge
  }
}
```

Placing them side-by-side with changing sizes is tricker than it looks. Instead of trying to write a formula, I used a running total. `pos.x` starts on the left, and is increased it by the size of the current block (1/2 over, then place, then the other 1/2 over is to account for block placement using the center).

This next one is a variation of that, but with colors. We'll make a list of colors, with help from the Inspector, any amount. Then we'll make 2-4 balls of each color, in random positions. That's a nested loop: the outer one hits each color, the inner one makes 2-4 of that color:

```
List<Color> Col; // sized and filled in Inspector
public GameObject blockPF; // drag in some prefab

void Start() {
  Vector3 pos; pos.z=0;
  for(int ci=0;  ci<Col.Count; ci++) { // each color
    int count = Random.Range(2,4+1);
    for(int j=0; j<count; j++) { // how many of each
      GameObject gg = Instantiate(blockPF);

      gg.GetComponent<Renderer>().material.color = Col[ci];

      pos.x=Random.Range(-6.5f, 6.5f);
      pos.y=Random.Range(-3.5f, 3.0f);
      gg.transform.position = pos;
  }
}
```

I like to double-check the indexes: the inside uses `Col[ci]` to pick the color. The `ci` loop goes from 0 to `<Col.Count`. So that gets them all, and doesn't go off the edge.

With those last two I wanted to show how a nested loop feels like a grid, but doesn't have to be. The sizes one was a straight line, where-as this is just random.

The important thing about using a nested loop to make things is the description. "6 rows, with 8 each" is a nested description, so is "2-4 balls of each color."

28.3 Wedge exercises

An old trick to get practice with nested loops and indexes is drawing various triangles – really, grids with some of the boxes left out. Instead of making 7 in each row, we can change it up. For example, a triangular grid, where rows have 1, then 2, then 3 then 4:

```
0000
000
00
0
```

To keep things simple, I'm going to keep (0,0) on the lower left, with `i` going up, and `j` going right.

To simplify the loop code, and to show off functions and pointers more, I'll move creating and placing the blocks into a function. The pos x/y formulas are unchanged:

```
GameObject newBlockAt(int i, int j) {
  GameObject g = Instantiate(blockPF);
  Vector3 pos; pos.z=0; //
  pos.x = -6 + 1.2f*j; // +j goes right
  pos.y = -3 + 1.2f*i; // +i goes up
  g.transform.position = pos;
  return g;
}
```

This is mostly a wrapper around `Instantiate`. It returns a pointer to the block it made, just in case we want to make more changes. A normal use would be `GameObject blk=newBlockAt(0,0);`. That makes a block in the lower-left 00 position. We're allowed to ignore the return type. If we don't need to make any more changes, `newBlockAt(0,0);` by itself is fine.

An obvious triangle is the picture from above: 1 block in the first/bottom row, left side, then 2 in the row above that, and so on. Each row is as long as the row # plus 1:

```
// upper-left triangle loop:
for(int i=0; i<4; i++) { // 4 rows
```

```
  int numInRow = i+1; // will be 1,2,3,4
  for(int j=0; j<numInRow; j++) {
    newBlockAt(i,j);
  }
}
```

Before, the inner-loop always ran 8 times. Now, it runs based on an equation.

A shorter, more common version leaves out the extra `numInRow` variable. The equation for the row length goes directly into the inner loop test:

```
for(int i=0; i<4; i++)
  for(int j=0; j<i+1; j++) newBlockAt(i,j);
```

This is a mess to read, at first. The secret is that `i` isn't changing as the inner loops runs. When `i` is 2, `j<i+1` is locked at `j<3`. Clearly, this is very prone to infinite loops if the `i`'s and `j`'s are scrambled.

A variant of that triangle is putting the blocks on the right side, making a lower-right triangle. Before, each row started at 0 and went a different amount. For this one, each row starts in a different spot, and always goes to the end:

```
    O
   OO
  OOO
 OOOO
OOOOO
// lower-right 5x5 triangle loops:
for(int i=0; i<5; i++) {
  int rowStart = i; // each row starts 1 further in
  for(int j=rowStart; j<5; j++) {
    newBlockAt(i,j);
  }
}
```

The first row runs all the way from 0-4, the one above it 1-4, 2-4, 3-4 then the top is just 4.

If we changed the `rowStart` line to `rowStart=4-i`, we'd have an upper-right triangle. The bottom row would be 4-4, then 3-4, 2-4, 1-4 and 0-4 for the top row.

When I need to make something like this, I have to draw a numbered grid, mark each row and check the start/end numbers, guess a formula, run it, tweak it . . . .

We can use both tricks to make a pyramid. The first row will go all the way across, but then each row with start at one more, and end at one less. This makes rows of length 9, 7, 5, 3, 1:

```
    0
   000
  00000
 0000000
000000000
// pyramid:
for(int i=0; i<5; i++) { // there are 5 rows
  int rowStart = i, rowEnd=8-i;
  for(int j=rowStart; j<=rowEnd; j++) {
    newBlockAt(i,j);
  }
}
```

The inner loop could have been `for(int j=i; j<=8-i; j++)`, but I think the extra `rowStart` and `rowEnd` make it easier to read.

For more fun, here's an upside-down 1, 3, 5, 7, 9, 11 pyramid:

```
// pyramid:
for(int i=0; i<6; i++) { // there are 6 rows
  int rowStart = 5-i, rowEnd=5+i;
  for(int j=rowStart; j<=rowEnd; j++) {
    newBlockAt(i,j);
  }
}
```

A quick sanity-check: the bottom row, when `i` is 0, goes from 5-0 to 5+0, which is just one thing, in slot 5. When `i` gets to the last row, on top, it's 5. `rowStart` is `5-i = 0` and `rowEnd` is `5+i = 10`. So the top row is 0 to 10, which is 11 across, which is correct.

A checkerbox is every other square. It's not a wedge, but it seems like it belongs in this section. A really slick way is to go through every square and skip the ones where row+column is an even number:

```
// checkerbox:
for(int i=0; i<6; i++) {
  for(int j=0; j<8; j++) {
    bool useMe = (i+j)%2==0;
    if(useMe) newBlockAt(i,j);
  }
}
```

Another try at a checkerboard is to have the loop making the rows go by 2's, starting on 1 for even rows and 0 for odd rows. This isn't as nice, but it works:

```
// checkerbox:
for(int i=0; i<6; i++) {
  int rowStart=0;
  if(i%2==0) rowStart=1; // even rows skip 1st square
  for(int j=rowStart; j<8; j+=2) {
    newBlockAt(i,j);
  }
}
```

28.4 break, continue

Loops have two special commands which seem like cheating, but are so nice we use them anyway.

The first one is `break;`. It quits a loop immediately. If it's a `for` loop, it won't even run the third `i++` part.

In some function examples, I've been using a mid-loop `return;` to quit a loop right away. `break` is the same idea, but better.

This clumsy `break`-using loop searches an int-list for the first negative number. As soon as we find one, we can quit, with `i` on that number:

```
// find index of first negative number:
int i;
for(i=0; i<A.Count; i++) {
  if(A[i]<0) break; // quits loop right now
}
if(i>=A.Count) print( "none are negative" );
else print( "index " + i + " is " + A[i] );
```

Notice how `i` needed to be declared before the loop. Otherwise it would be local to the loop and would be thrown away.

A neater way is using extra variables to record what you need. This checks for the first `'e'` in a string and saves the index:

```
int eIndex=-1; // fall-through "not found" value
for(i=0; i<w.Length; i++) {
  if(A[i]=='e') { eIndex=i; break; } // set answer and quit
}
```

Without the `break`, this finds the last `'e'`.

Then here's a fakey one. I want to count the `'z'`s, but strings with a space don't count (if we see a space, the answer is 0):

```
int zCount=0;
for(i=0; i<w.Length; i++) {
  if(A[i]==' ') { zCount=0; break; } // any space ruins it
  if(A[i]=='z') zCount++;
}
```

`break` won't let us do anything we couldn't do before, but it can make the code look nicer. When you see a `break` in a loop, it means we've seen all we need to see. It's often nicer than a `done=true;` style loop.

A common trick is a loop that runs forever, but you plan to use `break;` to stop it. The loop test is usually `while(true)`. Everyone knows that means you're using `break` or `return` to get out.

Here's a forever loop that rolls 2 6-sided dice until they get different numbers:

```
int d1=-1, d2=-1;
while(true) {
  d1=Random.Range(1,6+1);
  d2=Random.Range(1,6+1);
  if(d1!=d2) break; // yay! we got what we wanted
}
print("rolled " + d1 + " and " + d2);
```

A way to read this is "keep rolling both dice, quit when they're different numbers."

A funny thing is how the `break;` test is the opposite of what the loop test would be. As a normal loop, this would be `while(d1==d2)`. But the break uses `if(d1!=d2)`. The loop condition is when we keep going, the `break;` condition is when we stop.

Here's a `for(true)` plus `break`, abused to make a loop hitting every box in a list (never do this, but it's still pretty cool):

```
for(int i=0; true; i++) { // counts 0,1,2 ... forever
  if(i>=A.Count) break; // quit past end of list
  print(A[i]);
}
```

You usually don't have to think about `break;` while you write code. A loop will just naturally have a place where it would be nice to quit right now, and you remember the `break;` command.

Just so you know, `break`'s only quit one loop. A `break;` inside a nested loop will quit only the inside. The outer loop will keep going as normal. That's almost always what we want.

The other loop shortcut command is `continue;`. It's also used only by itself, inside a loop. It "skips to the next one" – it jumps to just before the final close-curly.

In this example it helps us print even numbers above 10, from some list:

```
// print even numbers over 10:
for(int i=0; i<A.Count; i++) {
  int n = A[i];
  if(n<=10) continue; // too small, skip to next
  if(n%2!=0) continue; // not even, skip to next
  print(n);
}
```

This loop hits every box in `A`. The `continue` command doesn't quit the whole loop early, just the step we're on now.

But otherwise `continue` is like `return` or `break`, since it magically changes flow-of-control. That last `prints(n)` loop line always runs, but only if the `continue` lets it get there.

`continue` is mostly used when you'd need to use complicated `if`'s. For example this uses `continue` to count words in a list that don't start with # and have the same first and last letters:

```
int firstLastSame=0;
for(int i=0; i<W.Count; i++) {
  string w = W[i];
  if(w.Length<2) continue; // skip words without 2+ letters

  char first=w[0];
  if(first=='#') continue; // skip words starting with #

  chart last=w[ w.Length-1 ];
  if(first==last) firstLastSame++;
}
```

Without continue, we'd have to use two ugly `if`s:

```
// first same as last, without continue:
int firstLastSame=0;
for(int i=0; i<W.Count; i++) {
  string w = W[i];
  if(w.Length>=2) {
    char first=w[0];
    if(first!='#') {
      chart last=w[ w.Length-1 ];
      if(first==last) firstLastSame++;
    }
```

```
  }
}
```

A common `continue` mistake is using it in a `while` loop and skipping the final `i++` (that can't happen with a `for` loop). This prints every word in `W`, skipping empty-strings, but has a fatal bug:

```
// infinite loop if we have "":
int i=0;
while(i<W.Count) {
  string w = W[i];
  if(w=="") continue; // skip last two lines. Repeats on same i !
  print(w);
  i++;
}
```

On an empty string, the `continue` jumps back before `i` increases. It will see the same empty string again, go back, forever.

Fun `continue` observation: adding it as the last line in a loop does nothing. It skips to the end, past 0 lines.

You can use `break;` and `continue;` in the same loop. They each do their normal thing.

Suppose we want to search through a list for short words (5 letters or less) and add them into one long string, but not past 25 letters total. If a word would go over, we skip it. This loop does that:

```
// silly loop to add all short words in W together, not past 25 letters:
string sum="";
for(int i=0; i<W.Count; i++) {
  string w = W[i];
  if(w.Length>5) continue; // skip long words
  if(w=="") continue; // there's no point adding ""

  int len1=sum.Length, len2=w.Length;
  int totalLen = len1+len2;
  if(ltotalLen>25) continue; // this word is too long to add, but keep trying

  sum += w;
  if(totalLen==25) break; // may as well quit, since at the max
}
```

The `break;` happened to be at the end, but there's no rule about that. I think this reads pretty well, if you remember `continue` means go to the next one and `break` means quit.

28.5 List nested loops

There are some natural `List` nested loops, for example checking whether two lists have any numbers in common.

The obvious way is how we'd do it by hand. To check two pages of numbers, put your finger on the first number on page 1, and scan page 2 for a match. Repeat for the next number on page 1. That's a nested loop:

```
bool nothingInCommon(List<int> A, List<int> B) {
  for(int i=0; i<A.Count; i++) {
    // compare A[i] to everything in B:
    for(int j=0; j<B.Count; j++) {
      if(A[i] == B[j]) return false;
    }
  }
  return true;
}
```

The outer loop selects `A[0]`, then the inner loop compares it to everything in `B`. Then the outer loop moves to `A[1]`, and the inner loop compares that to everything in `B`, and so on.

`i` goes with `A`, and `j` goes with `B`. I get them mixed-up a lot, using `A[j]` and `B[i]` by mistake, which off-end crashes and gives wrong answers.

Here's the same idea, but it counts how many things in `A` are also in `B`. It uses a `break;` so nothing gets double-counted ([3,6] and [3,3] have only one number in common):

```
// how many in A are also in B:
int inOtherCount(List<int> A, List<int> B) {
  int count=0;
  for(int i=0; i<A.Count; i++) {
    for(int j=0; j<B.Count; j++) {
      if(A[i] == B[j]) { count++; break; }
    }
  }
  return count;
}
```

The `break;` quits the one loop it's in, which is perfect. We check whether `A[0]` matches anything in `B`. If we find a match, we count it, quit the `B` loop, and move onto `A[1]`.

Checking just one list for duplicates is the same idea: take each item, and compare it to every other. But there are two differences. One is we can't compare an item to itself (if we did, every list would appear to have duplicates).

The other is we may as well skip double-compares: we compare box 0 to box 1, so there's no need to compare box 1 to box 0.

Combining those ideas, we get this rule: the inner loop should compare `A[i]` to everything that comes after it in the list.

Another rule, not as important: the outer loop can stop one before the end (the last item has already been compared to everything):

```
bool hasDuplicate(List<int> A) {
  for(int i=0; i<A.Count-1; i++) { // no need to check last one
    for(int j=i+1; j<A.Count; j++) { // compare with everything past i
      if(A[i]==A[j]) return true;
    }
  }
  return false;
}
```

That inner loop is a little like the triangle nested loops, the way it starts based on `i`. For a double-check: suppose `i` is 3. The inside loop starts by comparing `A[3]` to `A[4]`, which is what should happen.

28.5.1 Sorting

There are some very sneaky nested loops that sort a list. This is getting into the area of algorithms – where three days of work comes down to a few innocent-looking loops. But these aren't too bad.

We know how to find the smallest item in a list. To sort, we'll do that, add it to a new one, and cross it out from ours. Then we'll do it again: find the new smallest (since we crossed-out the old smallest,) add that to the end of the new list, and cross that one out.

It works like this:

```
A: 4 3 1 5 2 <-unsorted list
B:

A: 4 3 99999 5 2
B: 1

A: 4 3 99999 5 99999
B: 1 2
 ...
```

It's a loop around "find the smallest" and is called a selection sort. Here's the code for that crude idea above:

```
// A is unsorted, slowly sort it into B:
List<int> B = new List<int>();
```

```
for(int i=0; i<A.Count; i++) {
  // find smallest in A (code copied from old chapter):
  int smallIndex=0; // A[0] is smallest, so far
  for(int j=1; j<A.Count; j++) {
    if(A[j]<A[smallIndex]) smallIndex=j;
  }
  // now smallIndex is on the smallest thing in A, use it and cross it out:
  B.Add(A[smallIndex]);
  A[smallIndex]=99999; // cheap way to cross it out
}
```

These are tricky. If you write something like this, put some `print`'s in the middle so you can see where it's messing up (the first try always messes up).

There's an even cleverer, more confusing way to do it with just one list. We find the smallest and *switch* it into the first position. Then find the smallest out of the rest and switch that into the second spot.

This is a real selection sort (you can probably look it up for more explanation):

```
for(int i=0; i<A.Count-1; i++) {
  // skip everything before i -- it's sorted,
  // find smallest thing from i to end:
  int smallIndex=i;
  for(int j=i+1; j<A.Count; j++) {
    if(A[j]<A[smallIndex]) smallIndex=j;
  }
  // now swap it into A[i]:
  int temp = A[i];
  A[i]=A[smallIndex];
  A[smallIndex]=temp;
}
```

If you liked this, insertion and bubble sort are the other two you could look up. They're both nested loops that make no sense until you read the plan.