Chapter 10

More ifs and randomness

This section is about using random numbers. But, really, it's more examples using `if`s and programming tricks. Random numbers are to give us more things to do. And they're just fun to use.

10.1 Random rolls

The command to roll a random number is `Random.Range`, and then the numbers you want to roll between:

`n = Random.Range(1.0f, 10.0f);` rolls a random number between 1 and 10. It's a `float`, so it could be 1.35 or 6.93201.

The program below tests this in Update. It sprays out random numbers between 2 and 6:

```
void Update() {
  float n = Random.Range(2.0f, 6.0f);
  print( n );
}

// Possible output:
5.573
4.152441
2.80017
4.376728
```

For now, the command just works that way. The only thing we need to know is that we can change the two numbers inside.

Here are more of the rules:

- The command is really `Range` in the `Random` namespace. That's why it's `Random.Range`.

- The parens are like the ones for `print`. They have to be there. The insides are: the low number, a comma, and the high number. The comma is special, and just goes there.
- Random numbers naturally get funny streaks – it's normal to get lots of high numbers in a row, or lots of low ones. Don't worry if it does – just let it run for a while and things will even out.

This example shows how variables and assignment statements work the same as always. We roll a random number once, using Start, then print the variable over and over in Update:

```
float n; // global

void Start() {
  n=Random.Range(100.0f, 200.0f);
}

void Update() {
  print(n);
}

// Possible output:
106.5
106.5
106.5
```

All of the magic is the way `Random.Range(100.0f, 200.0f)` could be anything. But once it picks a number, it's just a number. `n` doesn't change by itself, since variables never do that. It's not "infected" with randomness.

And, of course, if you ran this over and over it would print different numbers, but always the same number repeated.

This next example is the same idea. We roll a number at random, print it, then add 1 and print it. The second number is always 1 more than the first. The trick is, don't think "one more than a random number." Think "one more than `n2`":

```
float n2;

void Update() {
  n2 = Random.Range(50.0f, 60.0f);
  print("A " + n2);
  n2 = n2 + 1;
  print(" B " + n2);
}
```

```
// Possible output:
A 52.6
 B 53.6
A 59.23
 B 60.23
```

This last example shows we can roll any range we want at any time, and they don't interfere with each other or "carry over":

```
void Update() {
  float a, b;
  a = Random.Range(3.5f, 4.5f);
  b = Random.Range(-1000.0f, 1000.0f);
  print(a+" / "+b);
}

// Possible output:
3.78 / -765.4
4.36 / 350.4563
4.145 / 204.5
4.0 / 4.0 <-- odds are super-slim, but it could happen
```

`a` is always between 3.5 and 4.5, and `b` is always something in-between plus and minus a thousand.

10.2 Random Positions

We can make the moving, wrap-around code more interesting with random rolls. If have make a variable for the `y` position, we can roll it randomly after each lap:

```
public float x=-7;
public float y=0;

void Update() {
  x+=0.05;
  if(x>7) { // wrap-around:
    x=-7;
    y=Random.Range(-4.0f, 4.0f); // <- new line
  }
  transform.position = new Vector3(x,y,0);
}
```

It rolls random `y` once at the start of each lap. This means it stays perfectly level as it moves right. `y` won't change again until the next lap.

Because of how random works, it might be near the bottom for a few laps, or appear to restart in the same place a few times. But if you let it run (increase the speed if you want) you should see it jump around.

A fun thing is we know our Cube is just snapping to a new `y`. But if the edges are set right, it looks like a new Cube coming in on a different row.

Another thing we can do is pick a completely random spot on the screen, by randomly picking `x` and `y`. This uses a delay counter to pop anywhere at random about once a second:

```
public int delayCount=0; // no delay for first time

void Update() {
  delayCount--;
  if(delayCount<=0) {
    delayCount=80;
    float x=Random.Range(-7.0f, 7.0f);
    float y=Random.Range(-4.0f, 4.0f);
    transform.position = new Vector3(x,y,0);
  }
}
```

Suppose we want the random position to alternate left side / right side. If we save the `x` position, we can check which side we're on now with `if(x<0)`, then roll the opposite:

```
public int delayCount=0;
public float x=0; // global so we can look at the old x

void Update() {
  delayCount--;
  if(delayCount<=0) {
    delayCount=80;
    float y=Random.Range(-4.0f, 4.0f);

    // left or right. Do opposite of old x value:
    if(x<0) x=Random.Range(1.5f, 7.0f);
    else x=Random.Range(-7.0f, -1.5f);

    transform.position = new Vector3(x,y,0);
  }
}
```

I just picked 1.5 as a number far enough from 0 that we can easily tell which side we rolled. Also note how -7 is first in `Random.Range(-7.0f, -1.5f)`, since it's the smaller number. That's easy to mess up with two negative numbers.

10.2.1 Random size

This randomly rolls the width and height of a Cube, about once a second. It's really no different than rolling a random x/y position, but it looks different:

```
int delayCount=0;

void Update() {
  delayCount--;
  if(delayCount<=0) {
    delayCount=80;
    float wd=Random.Range(0.4f, 3.0f);
    float ht=Random.Range(0.4f, 3.0f);
    transform.localScale = new Vector3(wd, ht, 1);
  }
}
```

This might make a square, if it happened to roll width and height close together, but it will usually make rectangles.

If we wanted a random *square*, we could roll one size and use it for width and height:

```
    // replace last three lines of "random-size cube" with this:
    float sideLen=Random.Range(0.4f, 3.0f);
    transform.localScale = new Vector3(sideLen, sideLen, 1);
```

I like this example, because it shows that even random numbers work the way we program them. We can roll once for each side, or once for both, or anything else we can think of.

10.2.2 Color

This rolls a completely random color. 0-1 for red, green and blue:

```
void Start() {
    float r=Random.Range(0.0f, 1.0f);
    float g=Random.Range(0.0f, 1.0f);
    float b=Random.Range(0.0f, 1.0f);

    transform.GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

A completely random color is usually an ugly grayish brown. A trick is to pick a good color, and roll close to those values.

This rolls a random "orangey" color by picking a smaller range for each slot:

```
void Update() { // every frame, so will flicker
    float r = Random.Range(0.8f, 1.0f); // big red
```

```
    float g = Random.Range(0.3f, 0.7f); // medium green
    float b = Random.Range(0.0f, 0.4f); // low blue
    transform.GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

Another method is to pick a random "brightness", and multiply the original orange color by it. We always get the same shade of orange, but brighter or darker:

```
void Update() { // every frame, so will flicker
    float bright = Random.Range(0.3f, 1.0f); // "brightness" 1=100%

    // using base orange of (1, 0.5, 0.1)
    float r = 1.0f*bright;
    float g = 0.5f*bright;
    float b = 0.1f*bright;
    transform.GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

Colors are harder to tell apart than sizes or positions. In the first program, I can see some red and yellows pop up. The second one has more light/dark difference, but always looks pretty orange, to me.

I use tricks like this to give every brick in a wall a slightly different color. Sometimes one way just doesn't look right, and the other way does.

10.3 More things to randomize

Way back, we could take x,y movement code and change `ySpd` to make it go diagonal. But we had to do it by hand. Now we can roll it randomly. Then we also randomize `xSpd` and have it randomly travel only 1 to 8 across. It rerolls all three things each time it ends a lap. It seems complicated, but watching it run should clear it up:

```
public float x=-7, y=0; // the position, starts center-left
public float xSpd=0.1f, ySpd=0; // straight and slow, for now
public float endXthisLap=7.0f; // randomly changes each lap

void Update() {
  x+=xSpd;
  y+=ySpd;
  if(x>endXthisLap) { // wrap-around and re-roll everything:
    x=-7; y=0;

    // randomize speed, slope, and distance we go:
    xSpd=Random.Range(0.05f, 0.15f); // always forward
    ySpd=Random.Range(-0.08f, 0.08f); // up or down
```

```
    endXthisLap=Random.Range(-1.0f, 8.0f); // x for when lap ends
   }
  transform.position = new Vector3(x,y,0);
}
```

The trick is it only rerolls the speeds at the end of a lap. While it moves in one lap, `xSpd` and `ySpd` don't change, so it gets a straight diagonal for each one.

The examples using a delay counter always used 80. We can roll that randomly. This pops the Cube to random spots, with a random delay between pops:

```
float delayCount=0; // changing to float, since random rolls are floats

void Update() {
  delayCount--;
  if(delayCount<=0.0f) {
    delayCount=Random.Range(30.0f, 150.0f); // the new part

    // same random position code from before:
    x=Random.Range(-7.0f, 7.0f);
    y=Random.Range(-4.0f, 4.0f);
    transform.position = new Vector3(x,y,0);
  }
}
```

10.4 Coin flips, percents, random tables

There's no built-in heads/tails coin flip, but we can make one. The usual way is to roll a random 0 to 1 and check for above 0.5.:

```
int n=Random.Range(0.0f, 1.0f);
if(n>0.5f) print("heads");
else print("tails");
```

Here's the code to pop us around randomly, but it flips a coin for each delay. This always waits either 60 or 140 between pops:

```
public int delayCount=0; // trick it into moving right away

void Update() {
  delayCount--;
  if(delayCount<=0) {

    // next delay: heads=short, tails=long:
    float delayFlip=Random.Range(0.0f, 1.0f);
```

```
    if(delayFlip<0.5f) delayCount=60;
    else delayCount=140;

    float x=Random.Range(-7.0f, 7.0f);
    float y=Random.Range(-4.0f, 4.0f);
    transform.position = new Vector3(x,y,0);
  }
}
```

I like this because there's no built-in way to randomly roll 60 or 140. But, we can make a plan: "flip a coin, heads=60, tails=140." If you watch it run, you really can see that it's using 2 delay lengths.

This next example flips a coin twice, just to show we can. Each time the Cube wraps around, we restart the lap either high or low, and turn either red or green:

```
public float x=-7, y=0;

void Update() {
  x+=0.2f;
  if(x>7) {
    x=-7;

    // heads=high, tails=low:
    float coinFlip = Random.Range(0.0f, 1.0f);
    if(coinFlip>0.5f) y=3;
    else y=-3;

    // heads=red, tails=green:
    coinFlip = Random.Range(0.0f, 1.0f);
    if(coinFlip>0.5f)
      GetComponent<Renderer>().material.color = new Color(1,0,0);
    else
      GetComponent<Renderer>().material.color = new Color(0,1,0);
   }
   transform.position = new Vector3(x,y,0);
}
```

Again, the way randomness works, this might get high+red a few times in a row, or get green the first four times. But if you wait enough laps it will even out and you'll see high/green, low/green, high/red and low/red.

I snuck in a tricky part. Notice how the first roll uses the declare-and-assign shortcut: `float coinFlip=Random` .... I reuse it for the next flip with just `coinFlip=Random` .... I had to remember not to double-declare `coinFlip`.

Moving on, a coin flip is really just a 50% chance. If we want a 10% chance we can change it to `if(coinFlip<0.1f)`. That's a tiny programming change, but it let's us do different stuff.

This changes the 60/140 delay so give an 85% chance for a short delay. The only code change is 0.5 becomes 0.85. But when you watch it, it seems like the Cube is popping at an exact interval, then "whoa" – it surprises you by staying in place extra-long:

```
// next delay: 85% short, 15% very long:
delayFlip=Random.Range(0.0f, 1.0f);
if(delayFlip<0.85f) delayCount=60; // 85% chance
else delayCount=240;
```

If we have more than two choices, we can make a table using a cascading if. Here's a table I made up for how long the delay should be:

```
// delay: medium 60%, short 30%, long 10%
// Table:
//  0 to 0.6 medium (90 ticks)
//  0.6 to 0.9 short (30 ticks)
//  0.9 to 1.0 long (240 ticks)
```

Here's the cascading, range-slicing `if` to make the table. We roll a 0 to 1 percent, then see which one it is:

```
// next delay: medium, short or long:
delayRoll=Random.Range(0.0f, 1.0f);
if(delayRoll<0.6f) delayCount=90; // medium
else if(delayRoll<0.9f) delayCount=30; // fast
else delayCount=240;
```

Notice how there's only one roll.

Here's the same idea, but with random treasure (the usual way games do it – mostly cheap stuff, and only a 2% chance for the top prize):

```
float roll=Random.Range(0.0f, 1.0f);
string treaure;
if(roll<0.5f) treasure="copper pennies";
else if(roll<0.85f) treasure="silver silverware";
else if(roll<0.98f) treasure="gold rings";
else treasure="magic sword";
```

We can mix and match these ideas. In this next example, I want something more complicated for where to put `y` when we wrap around: 70% of the time, y should be 0, otherwise flip a coin for high or low.

```
public float x=-7;
public float y=0;

void Update() {
  x+=0.2f;
  if(x>7) {
    // wrap-around:
    x=-7;

    float inCenterRoll=Random.Range(0.0f, 1.0f);
    if(inCenterRoll<0.7f) y=0.0f; // y is usually centered
    else {
      float hiLoFlip=Random.Range(0.0f, 1.0f);
      if(hiLowFlip<0.5f) y=Random.Range(0.5f, 4.0f); // high
      else y=Random.Range(-3.0f, -0.5f); // low
    }
  }

  transform.position = new Vector3(x,y,0);
}
```

This has got 3 levels of random. Even after it rolls "not the center" then gets the coin flip for "below," it still rolls for exactly how much below. This type of random looks different than a single anywhere roll. Over time it's obvious that it favors the center, but avoids the zone near the center.