Storing data is a basic feature of every programming language, but so is manipulating that data.

So now we’re going to discuss operators, the symbols you use to create expressions in C#.

The most common operator, which you are already familiar with, is the equals or assignment operator. This operator assigns the value of the statement on it’s right, to the variable on its left.

Equally as common are the arithmetic operators – addition, subtraction, multiplication, and division. If you’ve made it past the third grade, or have ever used a calculator, you know these.

What you might not be familiar with is the modulus operator, represented by the % sign. This operator returns the remainder of a division statement, rather than the division result.

Now, what do you do when you want to apply an operation to a variable, and store the result back into that same variable?

There are a couple ways to do this, and I’m sure you’re tired of looking at my face, so let’s look at some code.

Here we have a variable count, which I’ve initialized with a value of three. If we wanted to add 2 to the value of count, we could just say, count = count + 2

But this is repetitive, so we can also combine the + operator with the = operator, and say count+= 2. This is functionally the same. If we’re only changing the value by 1, we can use the increment and decrement operators.

You can actually place these operators on either side of the variable, and this entails a small quirk.

If you use an increment or decrement operator inside another statement, the execution order actually depends on whether you use the operator as a prefix or postfix.

Here we have a simple console print statement, and inside we’re going to increment count two different ways.

int count = 3;

count = count + 2; //count = 5

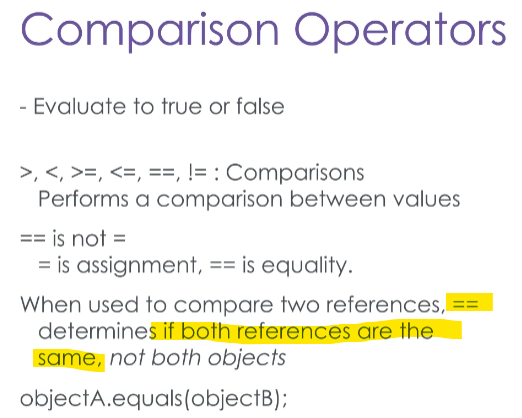
count += 2; //count = 7

Console.Out.WriteLine(“count++: “ + count++); //print count, then incrmt to 8

Console.Out.WriteLine(“++count: “ + ++count); //incrmt to 9, then print

When we execute these statements, we see that the prefix increment actually increases the value of count before printing, whereas the postfix increment will print the value of count first, then increment it later.

Comparison operators evaluate an expression to either a true or false value, which you should recognize as a Boolean data type. In this category we have >, <, >=, <=, ++, !=.

Note the difference between the equal-to operator and the assignment operator, commonly called double-equals and equals respectively.

At this point, I want to mention a quirk of using the double-equals operator to compare two reference types.

As you know, reference variables simply store the reference or memory address to an object in memory, not the object itself.

So if you use the double equals operator to compare two reference variables, you are not asking, “Are these objects the same,” you are asking, “Are these references the same?”

In other words, comparing two reference variables is asking if they both point toward the same object in memory.

If you want to determine the equivalence of two objects, you should use the Equals() method of one object, and pass in the second object as a parameter. However, you have to make sure the Equals method is properly written for the object type you are comparing – if you don’t override it, it will work just like the double-equals operator!

class Robot

{

public String type; // The type of a robot

public int maxSpeed; // How fast the robot can move in units

public override bool Equals(object obj)

{

// if the passed-in object-type reference does not point to

// a Robot object, then that memory object cannot possibly

// be equivalent to this robot.

if (!(obj is Robot))

return false;

Robot otherRobot = (Robot) obj;

// Robots are the same if their type is the same.

// The type is a string, so we use a string’s Equals method

// to compare them.

if (this.type.Equals(otherRobot.type))

return true;

else

return false;

}

}

Here I have the Robot class, and I have overridden the Equals method that is inherited from the System.Object class.

I want to determine if these objects are equivalent, and I have decided that equivalency means that both robots have the same type.

Because this method overrides the System.Object.Equals method, I must accept an Object reference type as a parameter just like the original equals method.

Right away, I want to check that the Object reference being passed in actually points to a Robot object in memory, so I check that with the “is” keyword. If it’s false, I return false right away.

If the passed-in reference does point to a Robot object, let’s store it in a Robot type reference, and cast the reference back to a Robot type. Now we can access its variables and methods correctly.

We do this to compare the type variables of this robot and the passed-in robot, and we do this using the String class’ Equals method.

Strings are objects too, but we’ll discuss that another time. What’s important is that the String object’s Equals method has also been overridden to compare the contents of the String.

Now that we have a way of determining object equivalence, we’ll move to a tester class, and create two Robots.

static void Main(string[] args)

{

Robot robotOne = new Robot();

robotOne.maxSpeed = 10;

robotOne.type = “Vacuum”;

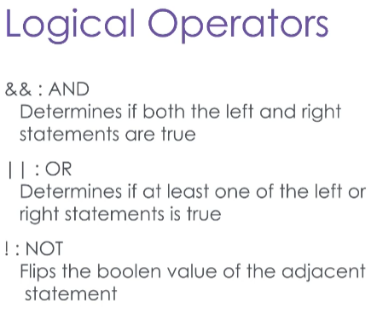
Console.WriteLine(“robotOne == robotTwo: “ + (robotOne == robotTwo));

Console.WriteLine(“robotOne.Equals(robotTwo): “ + (robotOne.Equals(robotTwo));

}

They will have the same maxSpeed, and the same type. For all intents and purposes, these robots are indistinguishable. Yet when we compare them with the double equals operator, we get a false result – the references point to different objects after all.

But the Equals method we implemented will return true, because both robots have the type “Vacuum”.



The logical operators, AND, OR and NOT, are used to chain comparison operators to make complex statements that still evaluate to **true** or **false**. AND is represented by two ampersands, while OR is represented by two pipes.

Not is an exclamation point, as seen in the not-equals operator. Let’s see an example of these in action.

int x = 2;

int y = 3;

Console.WriteLine(“X > Y : “ + (x > y)); //False

Console.WriteLine(“(X > Y) || (X == Y) : “ + ((x > y) || (x == y)); //False

Console.WriteLine(“(X < Y) && (Y == 3) : “ + ((x < y) && (y == 3)); //True

Here we have two variables, X and Y, initialized to 2 and 3 respectively. When we print the result of X > Y, we will see false.

Similarly, when we print the value of X > Y or X == Y, we will get false. However, the logical statement X < Y && Y == 3 will resolve to true.