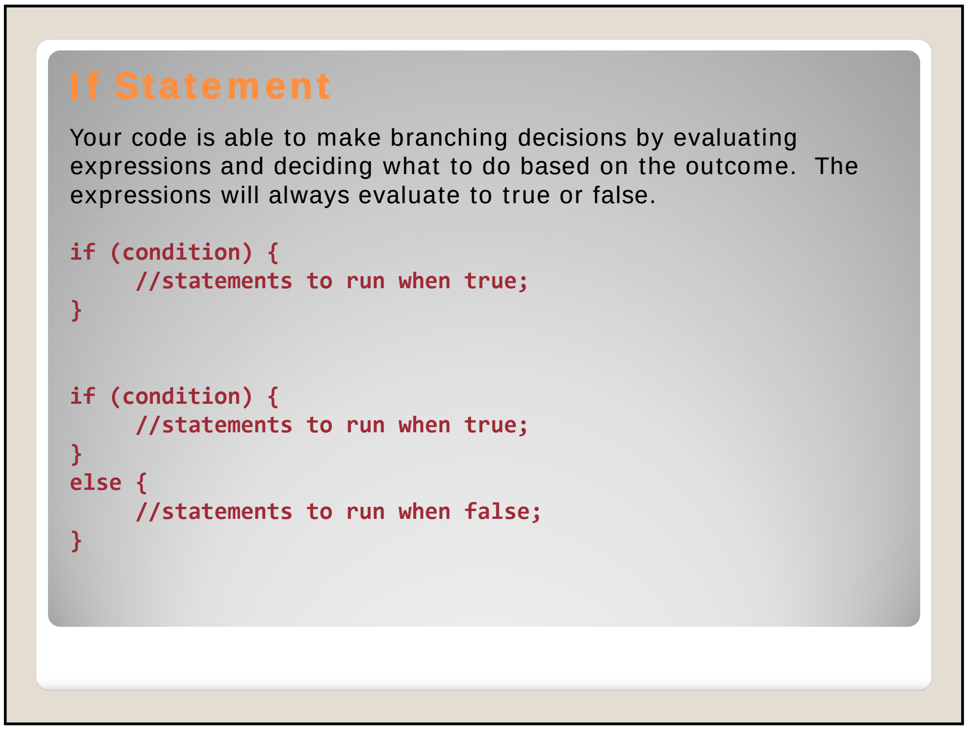
Google Android Development

Lesson#3

Code Flow

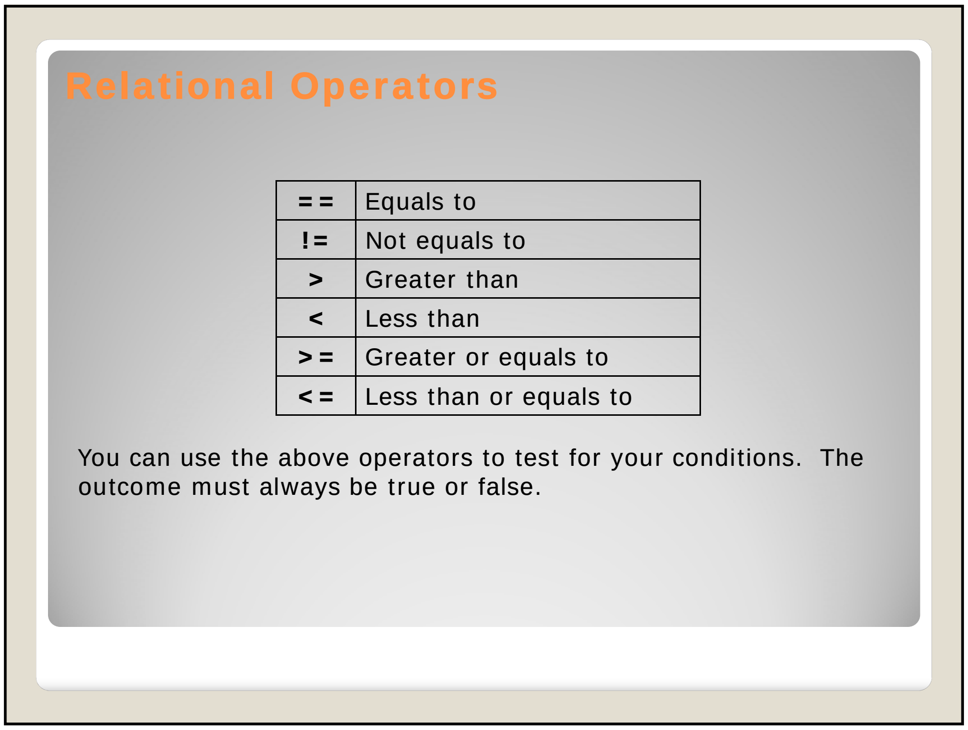
The code you write needs to be able to make decisions. Up until now, we've been working with very linear code – meaning that there was always one expected outcome. However, in many cases, you'll want to write code that branches to a different block of statements depending on a specific condition or a set of conditions being met.



If Statement

Your code is able to make branching decisions by evaluating expressions and deciding what to do based on the outcome. The expressions will always evaluate to true or false.

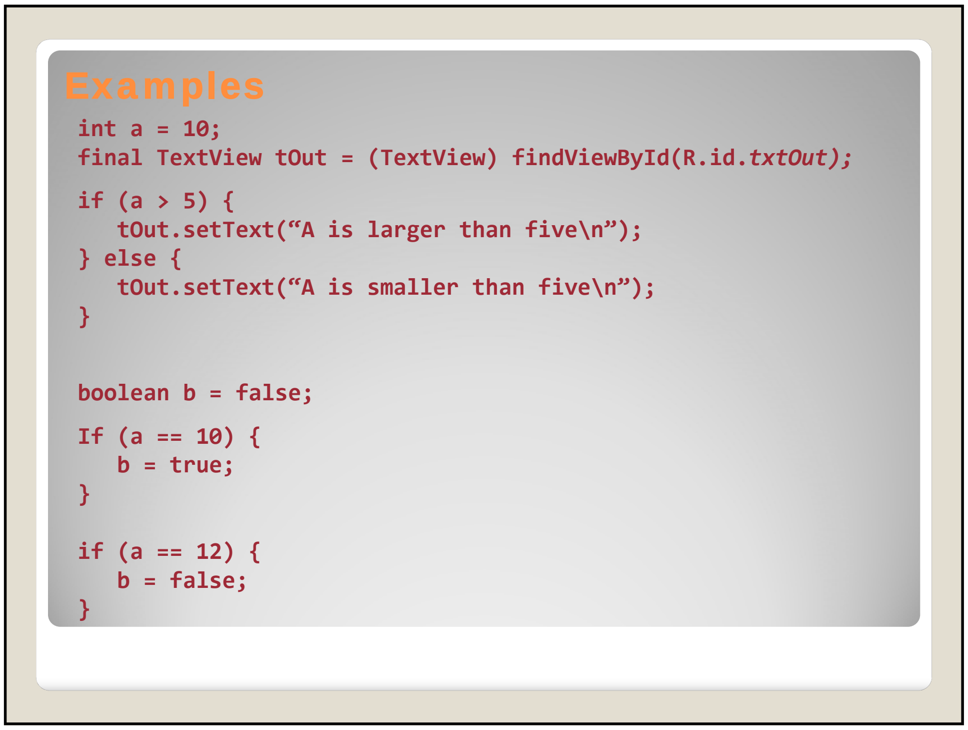
```
if (condition) {
     //statements to run when true;
}


if (condition) {
     //statements to run when true;
}
else {
     //statements to run when false;
}
```



Relational Operators

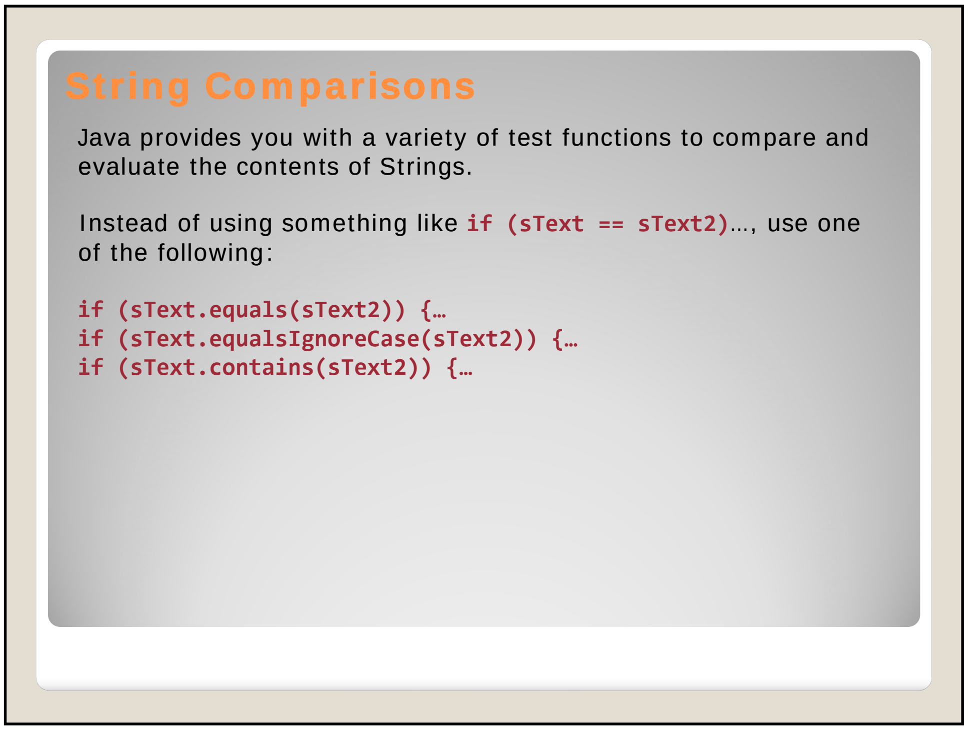
==	Equals to
!=	Not equals to
>	Greater than
<	Less than
>=	Greater or equals to
<=	Less than or equals to

You can use the above operators to test for your conditions. The outcome must always be true or false.



Examples

```
int a = 10;
final TextView tOut = (TextView) findViewById(R.id.txtOut);
if (a > 5) {
   tOut.setText(“A is larger than five\n”);
} else {
   tOut.setText(“A is smaller than five\n”);
}

boolean b = false;
If (a == 10) {
   b = true;
}

if (a == 12) {
   b = false;
}
```

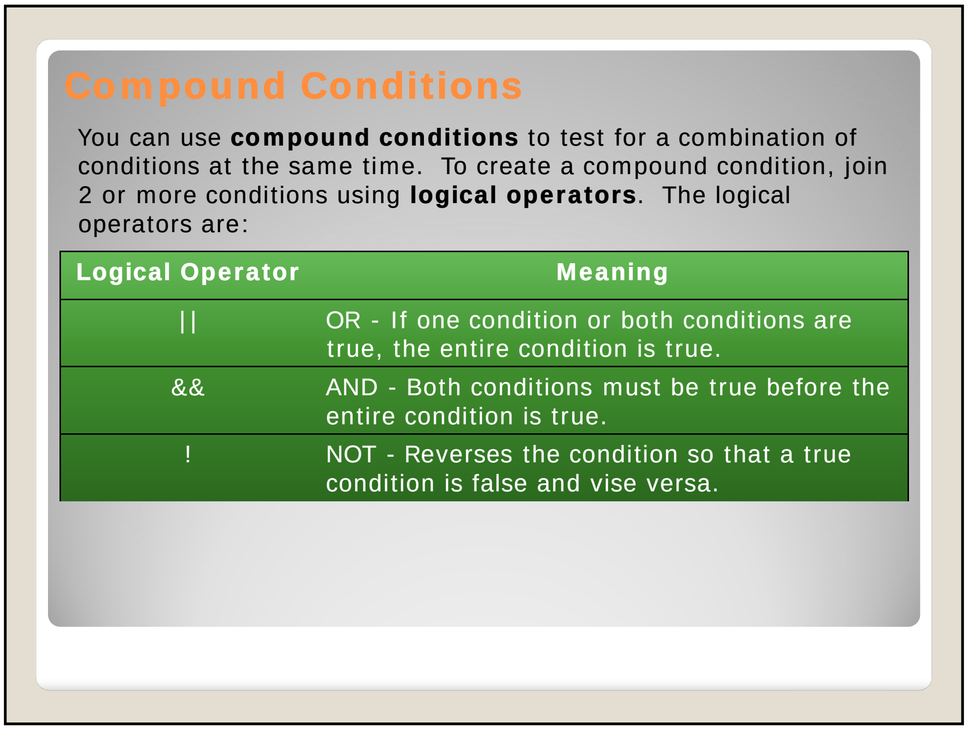


String Comparisons

Java provides you with a variety of test functions to compare and evaluate the contents of Strings.

Instead of using something like `if (sText == sText2)`…, use one of the following:

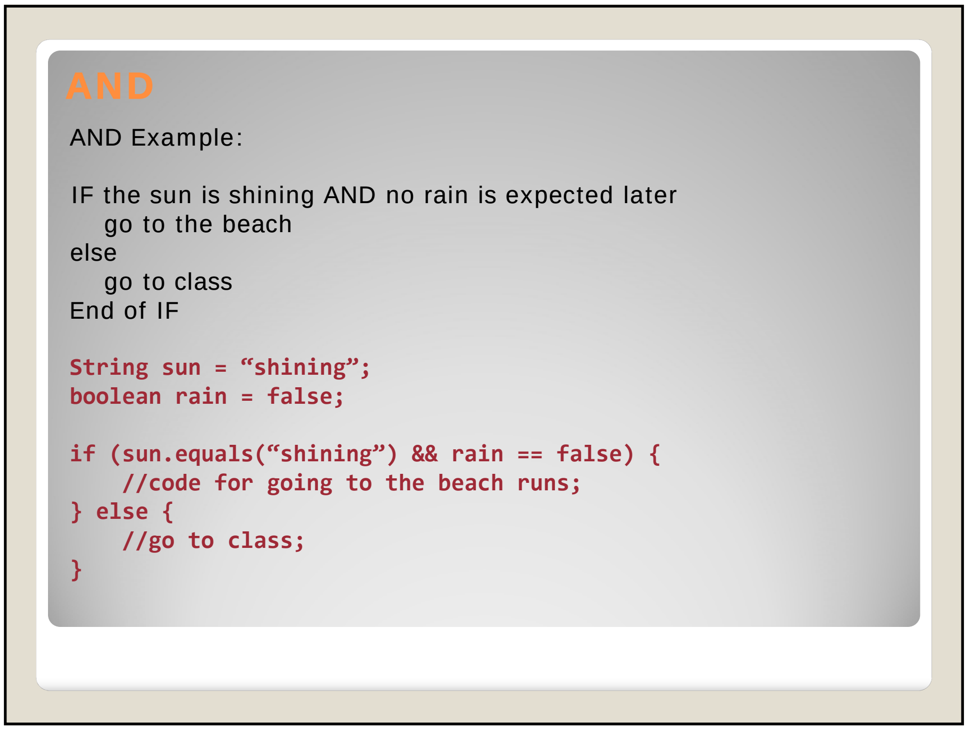
```
if (sText.equals(sText2)) {…
if (sText.equalsIgnoreCase(sText2)) {…
if (sText.contains(sText2)) {…
```



Compound Conditions

You can use **compound conditions** to test for a combination of conditions at the same time. To create a compound condition, join 2 or more conditions using **logical operators**. The logical operators are:

Logical Operator	Meaning
||	OR - If one condition or both conditions are true, the entire condition is true.
&&	AND - Both conditions must be true before the entire condition is true.
!	NOT - Reverses the condition so that a true condition is false and vise versa.

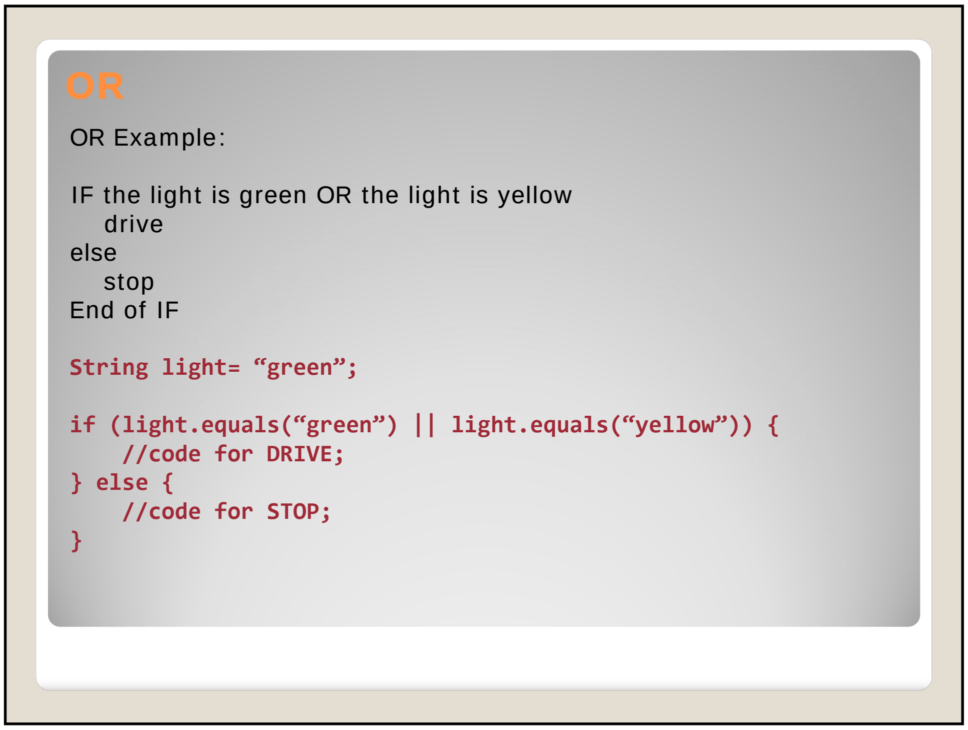


AND

AND Example:

```
IF the sun is shining AND no rain is expected later
   go to the beach
else
   go to class
End of IF
```

```
String sun = “shining”;
boolean rain = false;

if (sun.equals(“shining”) && rain == false) {
    //code for going to the beach runs;
} else {
    //go to class;
}
```

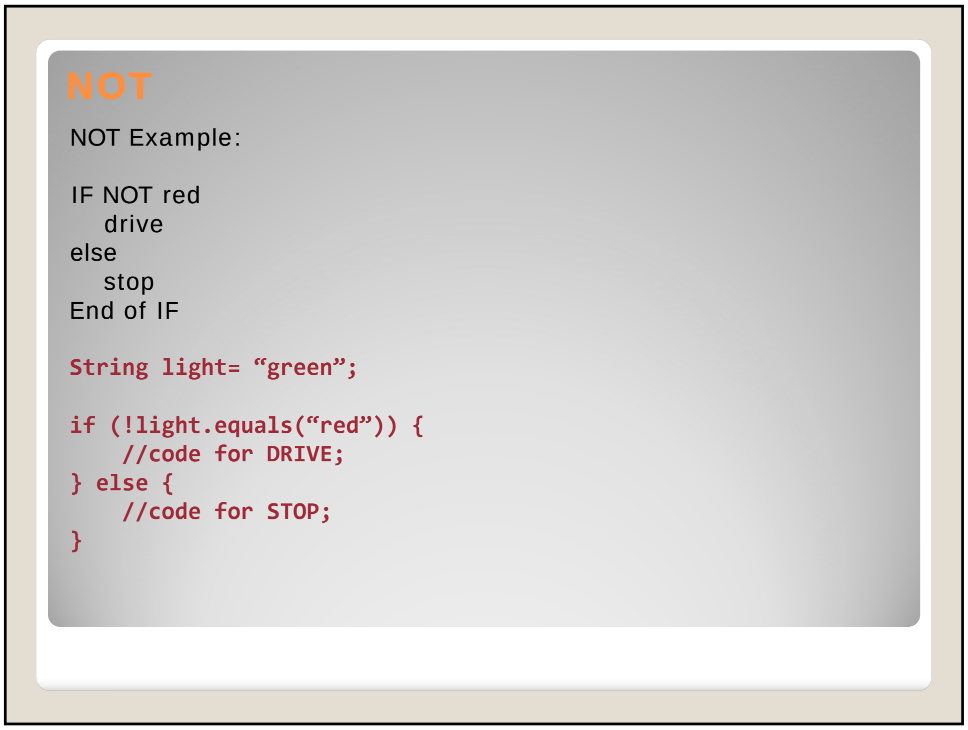


OR

OR Example:

```
IF the light is green OR the light is yellow
    drive
else
    stop
End of IF
```

```
String light= “green”;

if (light.equals(“green”) || light.equals(“yellow”)) {
    //code for DRIVE;
} else {
    //code for STOP;
}
```



NOT

NOT Example:

```
IF NOT red
    drive
else
    stop
End of IF
```

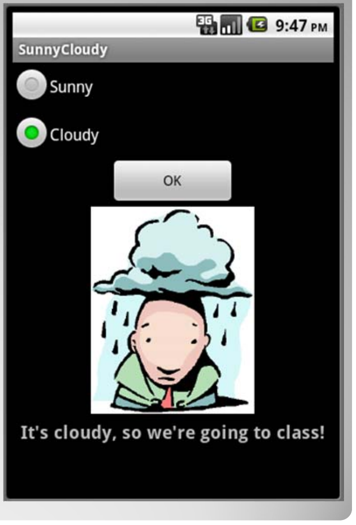
```
String light= “green”;

if (!light.equals(“red”)) {
    //code for DRIVE;
} else {
    //code for STOP;
}
```

Project

Today's first exercise will provide us with an example of IF statements in action, as well as plant the seed for our next main discussion point – Android GUI design.

We'll create a very simple GUI which will allow us to select Sunny or Cloudy radio button, and then click OK.

The image below the button will change depending on whether sunny or cloudy is selected.

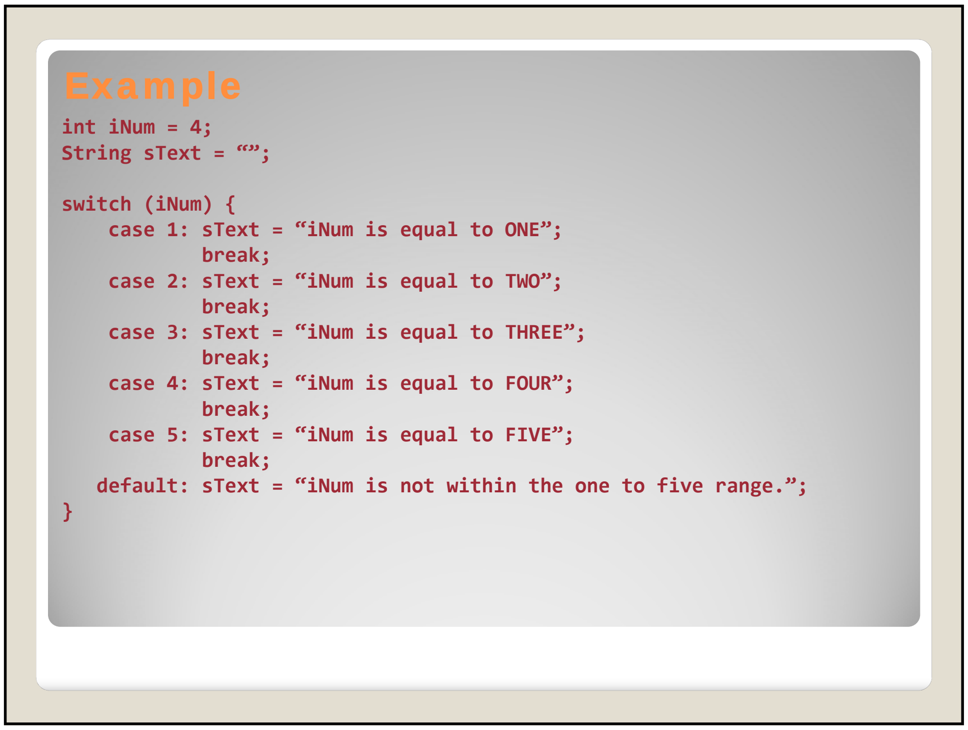


It's cloudy, so we're going to class!

Switch Statement

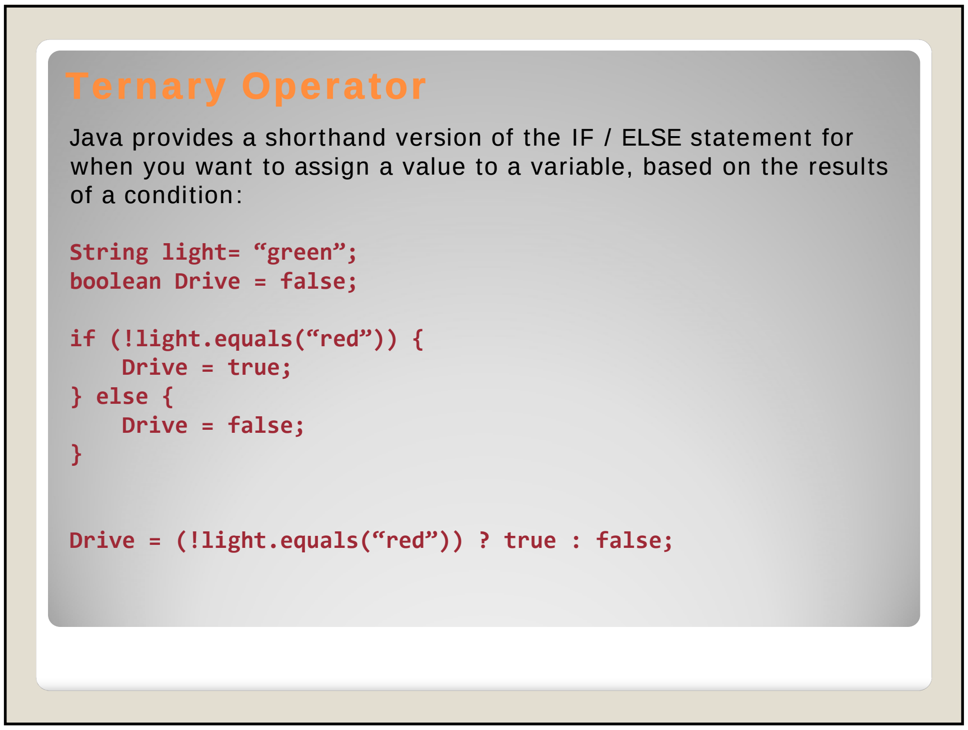
In the previous example, we used the state of a traffic light as the source for our condition. The traffic light is limited to three states, so testing for each one would require at most, three if statements. However, imagine if you have an object that has many more different states, and for each of those states, you want to do something different with your code. While you could write the necessary number of if statements, it may be much more efficient to create a switch statement.

```
switch (condition) {
    case 1: //code goes here;
            break; //use break or other conditions can be triggered
    case 2: //code goes here;
            break;
    case 3: //code goes here;
            break;
   default: //code goes here;
}
```



Example

```
int iNum = 4;
String sText = “”;

switch (iNum) {
    case 1: sText = “iNum is equal to ONE”;
            break;
    case 2: sText = “iNum is equal to TWO”;
            break;
    case 3: sText = “iNum is equal to THREE”;
            break;
    case 4: sText = “iNum is equal to FOUR”;
            break;
    case 5: sText = “iNum is equal to FIVE”;
            break;
   default: sText = “iNum is not within the one to five range.”;
}
```



Ternary Operator

Java provides a shorthand version of the IF / ELSE statement for when you want to assign a value to a variable, based on the results of a condition:

```
String light= “green”;
boolean Drive = false;

if (!light.equals(“red”)) {
    Drive = true;
} else {
    Drive = false;
}


Drive = (!light.equals(“red”)) ? true : false;
```

Loops

A loop within your application is typically a statement or a block of statements that has to repeat a number of times. Sometimes that number is known, and other times, it just has to run until a certain condition or set of conditions are met.

In Java, we have three types of loops:

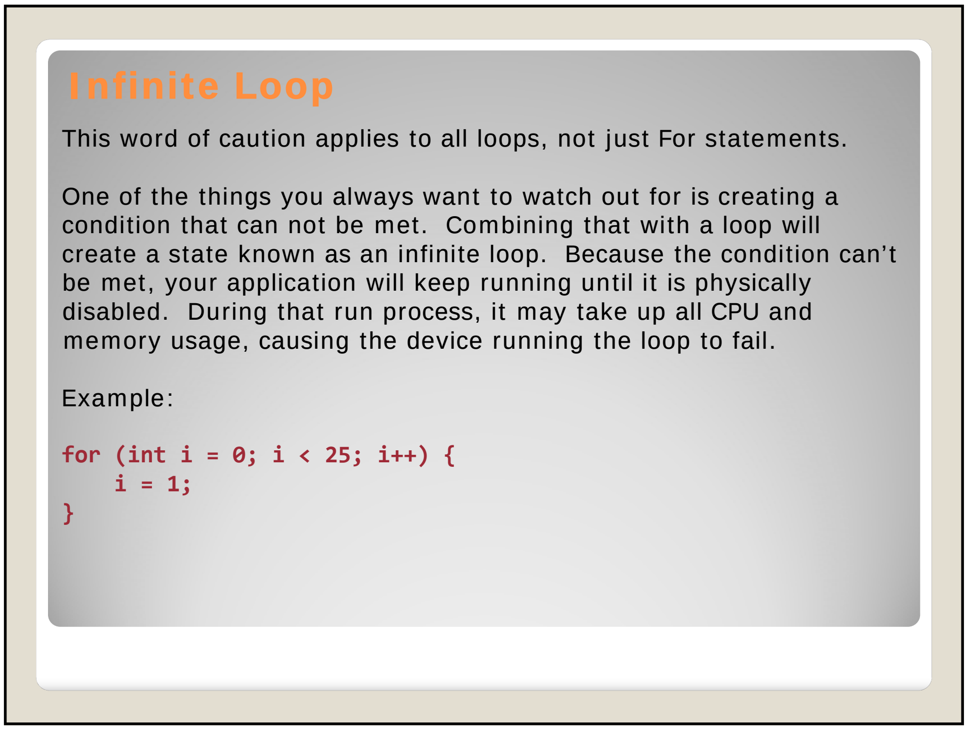
- **For Loop** – Number of iterations is known up front.
- **While Loop** – Number of iterations is based on a condition. May not execute if the condition is met before the first iteration.
- **Do-While Loop** – Number of iterations is based on a condition. Will execute at least once before testing for the condition.

For Loop

When you want to repeat a statement or block of statements a known number of times, the FOR loop is may be your best choice. The For loop consists of a variable counter known as the loop index. As part of the declaration, you define a starting value, ending value, and the increment value for the loop index.

Syntax/Example:

```
for (int i = 0; i < 25; i++) {
    //statements in loop
}
```



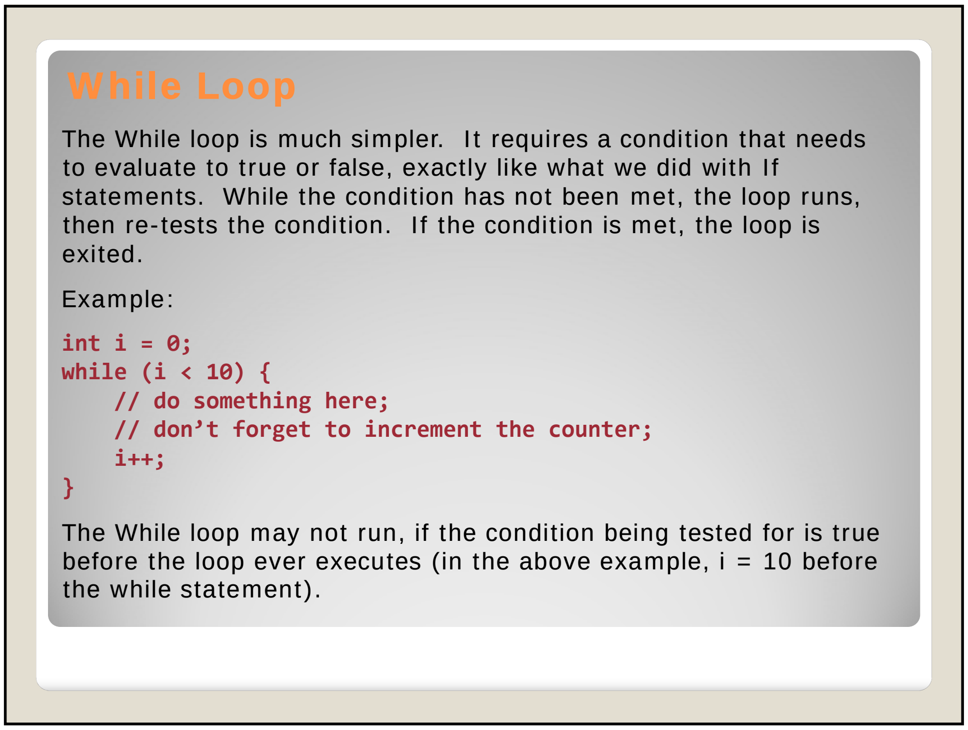
Infinite Loop

This word of caution applies to all loops, not just For statements.

One of the things you always want to watch out for is creating a condition that can not be met. Combining that with a loop will create a state known as an infinite loop. Because the condition can't be met, your application will keep running until it is physically disabled. During that run process, it may take up all CPU and memory usage, causing the device running the loop to fail.

Example:

```
for (int i = 0; i < 25; i++) {
    i = 1;
}
```



While Loop

The While loop is much simpler. It requires a condition that needs to evaluate to true or false, exactly like what we did with If statements. While the condition has not been met, the loop runs, then re-tests the condition. If the condition is met, the loop is exited.

Example:

```
int i = 0;
while (i < 10) {
    // do something here;
    // don't forget to increment the counter;
    i++;
}
```

The While loop may not run, if the condition being tested for is true before the loop ever executes (in the above example, i = 10 before the while statement).

Do While Loop

The Do While loop is very similar to the While loop, except that the condition test takes place at the end of the loop, instead of the beginning.

Example:

```
int i = 0;
do {
    // do something here;
    // don't forget to increment the counter;
    i++;
} while (i < 10);
```

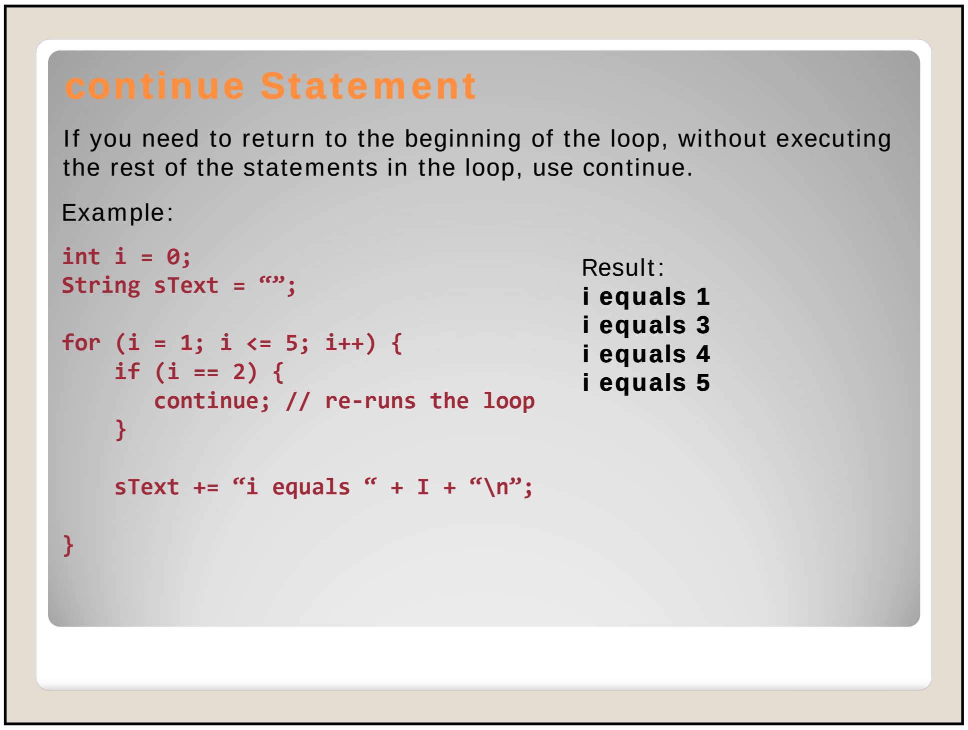
The code inside the loop will run at least once, because the condition is being tested after the code executes.

break Statement

If you need to exit any of the loops (for, while, do while), for any reason before they reach the end naturally, you can use the break statement.

Example:

```
int i = 0;
do {
    // do something here;
    // don't forget to increment the counter;
    i++;
    if (i == 5) {
       break;  // exits the loop
    }
} while (i < 10);
```



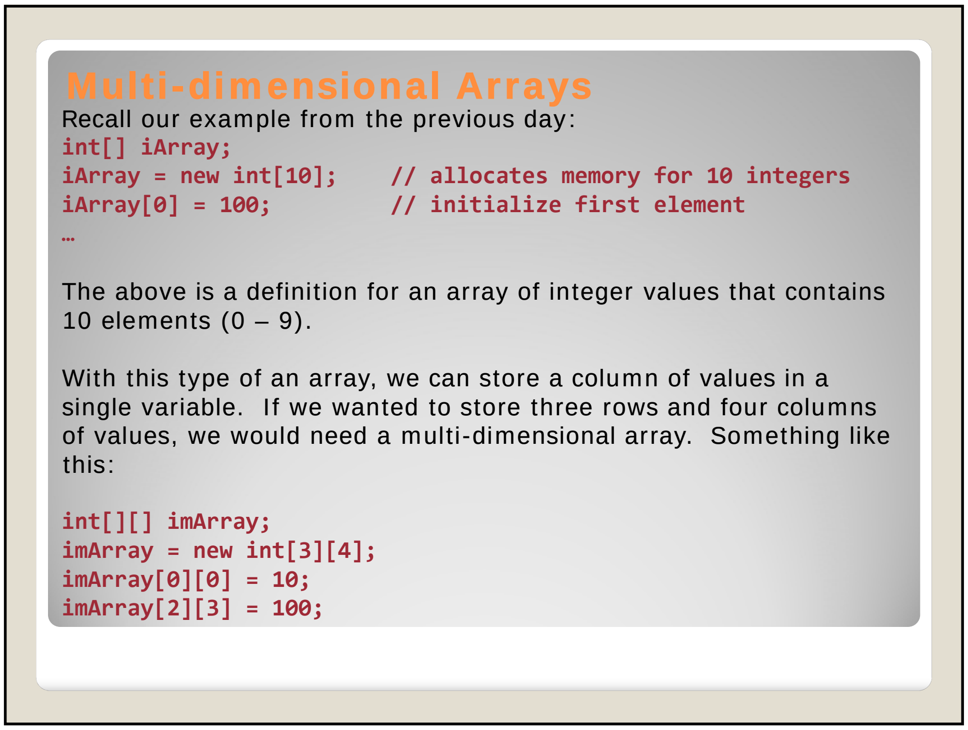
continue Statement

If you need to return to the beginning of the loop, without executing the rest of the statements in the loop, use continue.

Example:

```
int i = 0;
String sText = “”;

for (i = 1; i <= 5; i++) {
    if (i == 2) {
       continue; // re-runs the loop
    }

    sText += “i equals “ + I + “\n”;

}
```

Result:
i equals 1
i equals 3
i equals 4
i equals 5



Multi-dimensional Arrays

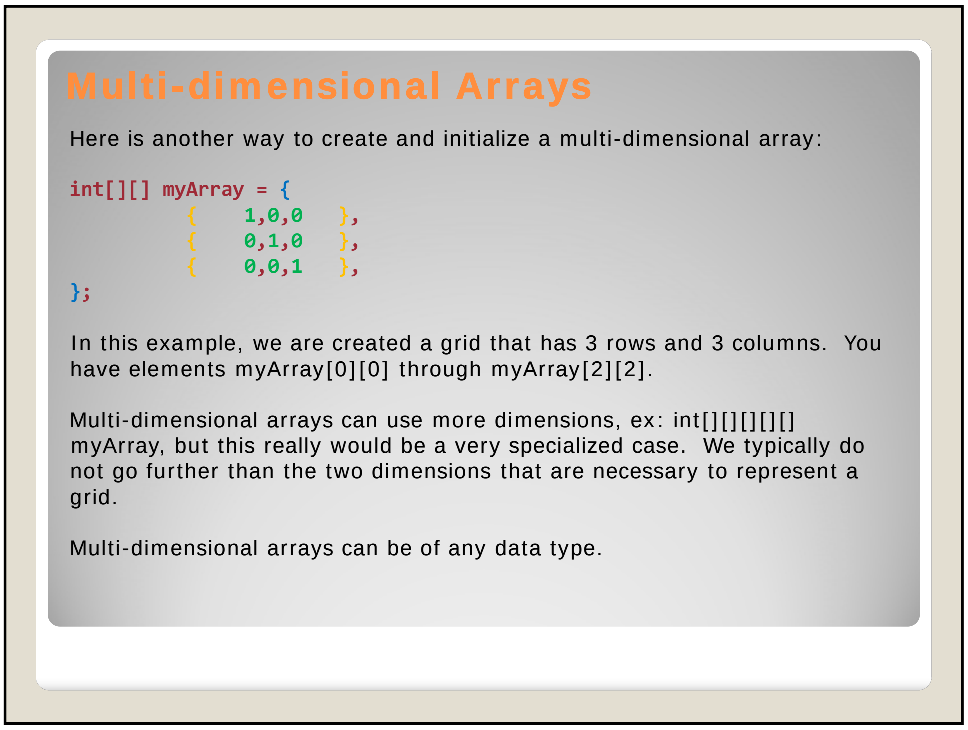
Recall our example from the previous day:

```
int[] iArray;
iArray = new int[10];    // allocates memory for 10 integers
iArray[0] = 100;         // initialize first element
…
```

The above is a definition for an array of integer values that contains 10 elements (0 – 9).

With this type of an array, we can store a column of values in a single variable. If we wanted to store three rows and four columns of values, we would need a multi-dimensional array. Something like this:

```
int[][] imArray;
imArray = new int[3][4];
imArray[0][0] = 10;
imArray[2][3] = 100;
```



Multi-dimensional Arrays

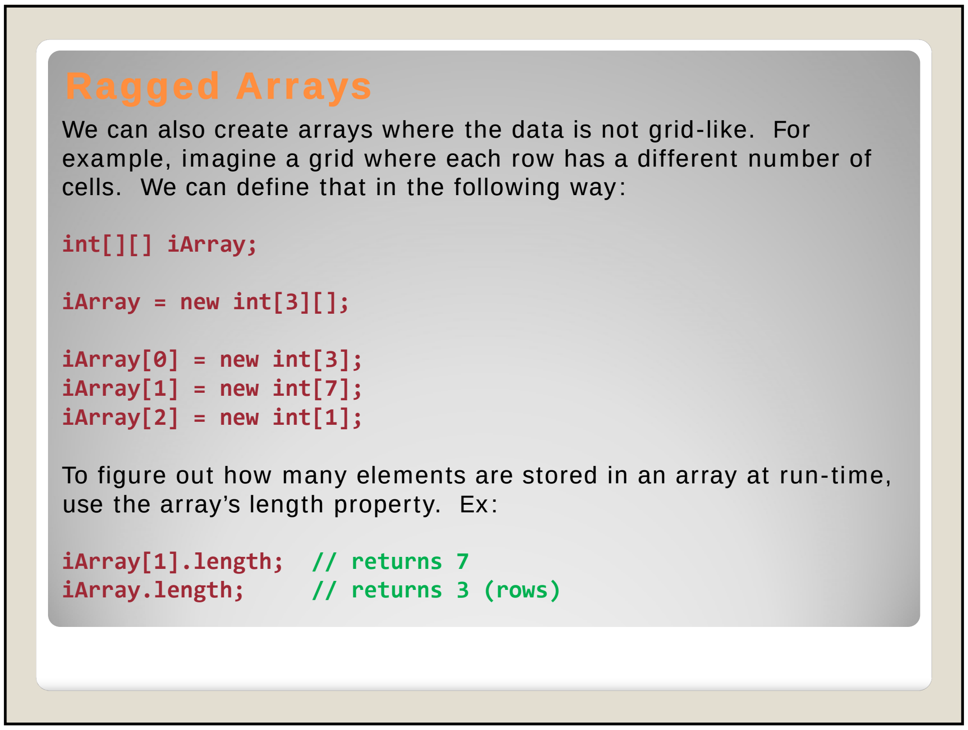
Here is another way to create and initialize a multi-dimensional array:

```
int[][] myArray = {
          {    1,0,0    },
          {    0,1,0    },
          {    0,0,1    },
};
```

In this example, we are created a grid that has 3 rows and 3 columns. You have elements myArray[0][0] through myArray[2][2].

Multi-dimensional arrays can use more dimensions, ex: int[][][][][][] myArray, but this really would be a very specialized case. We typically do not go further than the two dimensions that are necessary to represent a grid.

Multi-dimensional arrays can be of any data type.



Ragged Arrays

We can also create arrays where the data is not grid-like. For example, imagine a grid where each row has a different number of cells. We can define that in the following way:

```
int[][] iArray;

iArray = new int[3][];

iArray[0] = new int[3];
iArray[1] = new int[7];
iArray[2] = new int[1];
```

To figure out how many elements are stored in an array at run-time, use the array's length property. Ex:

```
iArray[1].length;  // returns 7
iArray.length;     // returns 3 (rows)
```

Loops and Arrays

Loops love arrays. Here's some basic code that let's iterate through a two dimensional array, setting the value for each cell.

```
int[][] imGrid;

imGrid = new int[3][3];

for (int i = 0; i < 3; i++) {  //rows

    for (int j = 0; j < 3; j++) {  //cols
        imGrid[i][j] = i+j;
    }

}
```

For Each

Arrays and loops go so well together that we even have a special loop that allows us to read data from an array. Note that this is for reading only, you can not assign values to an array using the for-each loop.

```
for (int[] row : imGrid) {

        for (int cell : row) {
             sText += " | " + cell + " | ";
        }

        sText += "\n";
}
```

Methods

Java allows us to define our own blocks of code, and store them as a method. What that means is that we can define our re-usable code, and name it. Then we can call that code when ever we need to perform whichever action our code performs.

There are really two types of methods that we can define – one that returns a value, and one that doesn't.

void Methods

To declare a method that doesn't have a return value, use the following syntax:

```
public void method_name (input variables) {
   // method code goes here
}
```

The **public** keyword is optional. If used, it will allow your method to be seen and worked with from other packages and other classes.

void tells Java that the method does not produce any sort of output.

The naming convention for your method is similar to variables. You can not start it with a number, and it should not be one of the reserved keywords.

Finally, the input variables are values that our method will use that come from outside of the methed. These are declared just like regular variables (ex: int iNum)

Methods With Return Values

To declare a method that has a return value, use the following syntax:

```
public data_type method_name (input variables) {
   // method code goes here
   return variable (of data_type)
}
```

data_type can be any of the variable data types we discussed before, such as int, String, boolean, double, float, etc.

The method will need a return statement which specifies the value of the particular data type to return back to the point in the code that called it. In essence, a method that returns a value acts just like a variable, but performs a predefined set of actions first. The return statement is typically the last statement in the routine because once return is reached, any code after it is not executed within the method.

Overloading Methods

Java allows you to define the same method multiple times with different signatures. A signature consists of the number and data types of variables being specified as input to the method.

Examples:

```
void doAddToGlobal() {  // regular method, no input variables
   iGlobal += 100;
}

void doAddToGlobal(int iAdd) {  //overloaded method
   iGlobal += iAdd;
}

int CustomAdd(int iNum, int iAdd) {  //method returns an int
   return iNum + iAdd;
}
```