

Conditional Statements

Conditional statements do not manipulate variables directly but rather affect which the basic statements will be executed and which will be skipped.

Let's write a class that makes use of conditional statements.

Represents a game where the purpose is to guess a number between 0 and 20.

```
public class HighLow{  
  
    /** represents the number to guess */  
    private int answer;  
  
}
```

Write a constructor for the class that initializes the value of answer. To make sure that the number is different each time we play the game, make the value random.

We can use class Random to generate a random number.

```
import java.util.Random;  
public class HighLow{  
  
    /** represents the number to guess */  
    private int answer;  
    public HighLow() {  
        Random r = new Random();  
        answer = r.nextInt(20);  
    }  
  
}
```

Now that we have initialized the instance variable, let's write a public method that lets the user guess the answer. If the guess is too low, we want to tell the user and to try again.

To do this we use an "if" statement with the general form:

```
if(condition) {  
    statement;  
}
```

```
}
```

where condition is a boolean expression - it evaluates to true or false.

```
public void guess(int x){  
    if(x < answer){ // condition is a boolean expression  
        System.out.println("Too low.  Guess again.");  
    }  
}
```

Compile.

Try it in interactions pane.

Notice that we use S.o.p to display the message on the screen.

That line of code will only be executed if the condition (x < answer) evaluates to true.

What happens if it is false? Nothing.

So we can print a message when the condition is not satisfied, we can expand the conditional statement into an "if-else-elseif" statement.

General form:

```
if(condition){  
    statement;  
} else if(condition){  
    statement;  
} else {  
    statement;  
}
```

Let's put one in the guess method:

```
public void guess(int x){  
    if(x < answer){ // condition is a boolean expression  
        System.out.println("Too low.  Guess again.");  
    }else if(x > answer) {  
        System.out.println("Too high.  Guess again.");  
    }else{  
        System.out.println("Correct!");  
    }  
}
```

Loops

Repetitive structure - allows you to keep doing something over and over again until some condition is met.

There are three types of loops in java:

1- while

```
// the condition is always a boolean statement
while (condition) {

    // the statement (s) you want to repeat- will only
    // occur if the boolean condition is true
    // The body MUST have some stepping expression that
    // changes the result of the Boolean condition
    body

}
```

E.g.

```
int i = 5;
while (i!=0){
    System.out.println(i);
    i--;
}
```

2- do...while

```
do {
    // the statement (s) you want to repeat- will only
    // occur if the boolean condition is true
    // The body MUST have some stepping expression that
    // changes the result of the Boolean condition
    body
} while (condition);
```

E.g.

```
int i = 5;
do {
    System.out.println(i);
    i--;
} while (i!=0);
```

What is the difference between the 2? What will happen if I change the statement in the beginning and initialize i to 0 instead of 5?

3- for

```
for (initialize counter; ending boolean condition ; stepping expression) {  
    body // the statement(s) you want to execute  
}
```

E.g.

```
for (int i = 5; i!=0; i--){  
    System.out.println(i);  
}
```

You **CAN** leave the three parts of a for loop empty and write this instead

```
for ( ; ; ) {  
}
```

but this is really bad style and not readable...all this is saying is

```
while (true){  
}
```

where the loop will keep going until you return or **break** inside.

Why is ++i equivalent to i++ in the for loop?

Because the value of i is not evaluated and used until the loop condition is evaluated -- **i is evaluated in a separate statement.**

In contrast, if we had the loop:

```
int i= 0;  
while(i++ < a.length) {  
    process a[i]  
}
```

this would differ from

```
int i= 0;  
while(++i < a.length) {  
    process a[i]  
}
```

Usually a for loop is used when there is a numerical counter and you know the starting and ending conditions. Sometimes a while loop is just more convenient. Look at File I/O

for an example.

Reading from files in Java

Reading data from files is a common and complex task. If the file is located on a network drive, or even on a hard drive, there is a lot that has to happen in order to read its data.

The API includes a class, `FileReader`, that does most of the hard work. This class will open a file and give you its contents one character at a time:

```
import java.io.*;
// open file c:\test.txt for reading.
// note the double '\'
FileReader fr = new FileReader("c:\\test.txt");
// read the first character and store it in c.
char c = (char)f.read();
```

Note that method `read` returns an `int`, not a `char`, so we cast. The cast is implicit (can be omitted), but is included here for reference.

Each time we call `read()`, it gives us the next character in the file.

When the `FileReader` reaches the end of the file, the `read` method will return `-1`, which is not a valid char.

```
FileReader fr = new FileReader("c:\\test.txt");
c = fr2.read();
while(c!=-1){
    System.out.println((char)c);
    c=fr2.read();
}
```

When you are done working with a file, remember to close it.

```
fr.close();
```

BufferedReader

Reading a file one character at a time is really tedious and not what I want. Usually you want to read a whole line at a time.

`BufferedReader` is a class that will do the file reading from the `FileReader` and return Strings containing entire lines of text from a file. Here is how it is used.

```
// open the file reader for file c:\test.txt
```

```

FileReader f = new FileReader("c:\\test.txt");
// give the file reader to the buffered reader constructor
BufferedReader br = new BufferedReader(f);
// read one line at a time
String line1 = br.readLine();
String line2 = br.readLine();
// etc.
br.close();

```

When `BufferedReader` finishes reading a file, it returns "null" instead of a `String`.

```

String line = br.readLine();
while (line!=null){
    System.out.println(line);
    line = br.readLine();
}

```

Writing to files in Java

Similarly, the `FileOutputStream` class allows you to write to a file one character at a time:

```

FileOutputStream fo = new FileOutputStream("c:\\testoutput.txt");
fo.write('a')
fo.write("haha") → error

```

You need to create a `PrintStream` object in order to write to a file one line at a time:

```

PrintStream ps = new PrintStream (fo)
ps.println("haha")

```

Exceptions

Java detects errors at run-time by throwing exceptions. If you call a method that throws an exception, you will have to either catch it and do something about it, or else throw it away to be handled by something else.

```

import java.io.*;

public class DoIO {
    public void readAFile() {
        // data.txt is a file that doesn't exist
        FileReader fr = new FileReader (new File ("C:\\data.txt"));
        System.out.println( (char) fr.read() );
    }
}

```

```
}
```

Fix the error by:

```
1 - public class DoIO throws IOException { .. }
2 -
public class DoIO {
    public void readAFile() {
        try {
            FileReader fr = new FileReader (new File ("C:\\data.txt"));
            System.out.println( (char) fr.read() );
        } catch (Exception e) {
            System.out.println( "uhoh error happened");
        }
    }
}
```

Reading input from keyboard

you want to read from `System.in` which is an `InputStream` in Java, use

```
BufferedReader br = new BufferedReader(new InputStreamReader
(System.in));
br.readLine();
```

each `br.readLine()` call will prompt the user to enter some information
`readLine()` returns a `String` with the text entered by the user and it can be stored in a variable. So in order to prompt the user to keep entering data until some stopping condition,

```
BufferedReader br = new BufferedReader (new InputStreamReader
(System.in));
System.out.println("enter a positive number, Type -1 to stop");
String in = br.readLine();
while(Integer.parseInt(in)!=-1) {
    System.out.println(in);
    in = br.readLine();
}
```

Arrays

An array is an object that contains many spaces to store a list of values of the same type.

array *a*:

index:	0	1	2	3	4
	45	86	23	32	100

Indexing starts at 0. Refer to the individual elements by:

array name [index]

Eg:

`a [1]` → is equal to 86

Array Rules

1. All elements of an array are of the same type.
2. The number of elements in an array cannot change once you have instantiated the array.
3. The type of the array is `T[]`, read as "array of T" or "T array". T can be any data type, primitive or non-primitive (class type).
4. An array is an object - in the memory model, it appears in the object space.

Declaration:

```
baseType [] arrayName;
```

`baseType` is the type that you want all the elements to have. It can be a primitive type or any other class type.

eg:

```
int [] a; // a is of type "int Array" not of type "int"  
String [] names;  
BookRecord [] bookArray;
```

Instantiating (constructing) arrays:

You **MUST** know the number of elements in the array at the time of instantiation.

- 1) When you know the size but not the contents:

```
a = new int[5];
```

This constructs an array **a** with 5 elements of type int.

You *cannot*, do this:

```
a = new int[];
```

Initializing an array (putting values into it):

There are a few different ways to initialize the elements of an array.

1) To assign values to the int elements of the array 'a':

```
a[0]= 1;  
a[1]= 2;  
a[2]= 4;  
a[3]= 8;  
a[4]= 16;
```

2)

```
int[] c;  
c= new int[3];  
c[0]= 1;  
c[1]= 2;  
c[2]= 3;
```

3)

```
for(int k= 0; k < 3; k++) {  
    c[k]= k+1;  
}
```

4)

```
int[] a= {1,2,3};
```

5)

```
int[] b;  
b = new int[] {1,2,3};
```

To find out the number of elements in an existing array use the expression

```
a.length
```

NOT

```
a.length()
```

Note that length is NOT a method. It is a final variable that is set once (on initialization of the array object) and cannot be changed afterwards. If you run out of space, for example, you cannot double the size of an existing array:

```
a.length= 2*a.length; // ERROR!!
```

Example: Array reversal

- Print an array - why doesn't `System.out.println(a)` work?
- Make an array by accepting input from a user
- send that array to another method to reverse it

```
import java.io.*;

public class ArrayReversal {

    public static void printArray(int [] a ){
        for (int i=0; i<a.length ; i++){
            System.out.println(a[i]);
        }
    }

    public static int[] makeArray() throws IOException{
        BufferedReader br = new BufferedReader (new
InputStreamReader(System.in));
        int [] a = new int [10];
        System.out.println("Enter 10 numbers");
        for (int i = 0; i<10; i++) {
            a[i] = Integer.parseInt(br.readLine());
        }
        return a;
    }

    /* Sets a to the reverse of a
    * note: don't make another array
    */
    public static void reverse(int[] a) {

        int k= 0;
        int n = a.length - 1; // try without -1...why does it give an error

        while (k<n){ // why is k!=n incorrect?
            int temp= a[k];
            a[k]= a[n];
            a[n]= temp;
            k++;
            n--;
        }
    }
}
```

Things to remember:

Draw memory models:

```
int [] a = new int [3];
```

```
String[] names= new String[3];
```

After execution, each element of names automatically contains null but each element of a gets the default value 0. In order to give the boxes in names a value, you need to explicitly instantiate each element.

```
names[1]= new String("Dave");
```

(show changes in mem model)

Passing an array as a parameter

If you create an array, pass it as a parameter to a method, and that method changes the array, the original array will also get changed. (Eg: reversal method above)

But lets see when that doesn't work:

Resizing Arrays

When we run out of space and need a larger array, we can use a method to double the size of the array. If we try to do so the way we did reversal,

```
// parameter b is born here. b gets initialized with a copy of the argument  
// passed in from the caller.
```

```
public static void doubleSize(int[] b) {  
    // calculate the capacity of the new array  
    int capacity= b.length * 2;  
    // construct the double-sized array  
    int[] b2= new int[capacity]; // B b2 born here  
    // copy the elements from b to the new array  
    for(int i=0; i<b.length; i++) { // i born here  
        b2[i]= b[i];  
    } // i dies here after final iteration  
    b= b2; // this doesn't work  
} // b2 and b die here
```

The parameter b dies at the end of method doubleSize.

As does the local variable b2.

When we call method doubleSize, we pass a **copy** of the value stored in reference variable b.

Assigning (changing the value of) an alias doesn't change the value of b.

We must explicitly assign a value to the variable b in method main or the object created in method doubleSize is lost.

```

/**
 * Returns an array containing the same elements as b
 * in the same positions but with double the length of b.
 */
public static int[] doubleSize(int[] b) {
    // calculate the capacity of the new array
    int capacity= b.length * 2;
    // construct the double-sized array
    int[] b2= new int[capacity];
    // copy the elements from b to the new array
    for(int i=0; i<b.length; i++) {
        b2[i]= b[i];
    }
    return b2;
}

```

How can you allow an array to store any kind of object?

Make an array of Objects.

```

Object [] anything = new Object[3];
anything[0] = new String ("hi");
anything[1] = new Integer(5);
anything [2] = new Double (4.5);

```

StringTokenizer, split

Allows you to divide a string of words into many pieces at a delimiter

Eg: "My name is Alifiya"

to separate that string at the spaces, and put each word in an array you can:

```

StringTokenizer st =new StringTokenizer("My name is Alifiya");
int length = st.countTokens();
String [] s = new String [length];
int i=0;
while (st.hasMoreTokens() ){
    s[i++] = st.nextToken();
    // System.out.println(st.nextToken()); This will make it skip to the
next token so if
    // you want to print at the value being used, store it in a separate
variable.
}

```

space is the default delimiter. If you want to change it, use the constructor:

```
StringTokenizer st = new StringTokenizer("My name is Alifiya", "a");
```

The other option is to use the method "split" from the String library.

```
String a = "My name is Alifiya";  
String[] s2 = a.split(' '); // split takes as an argument the  
character you want to split at.  
String[] s3 = a.split('a');
```

2-D Arrays

If you want to store tables, game boards etc..

Size is determined by number of rows and number of columns

Like a square or rectangle

It is an array of arrays (draw on board)

declaration

```
int [][] a;
```

instantiation

```
a = new [2][3]; // 2 rows and 3 cols
```

	0	1	2
0	0	0	0
1	0	0	0

	0	1	2
0	0	0	0
1	0	0	0

initialization

1)

```
a[0][0] = 1;  
a[0][1] = 2;  
a[0][2] = 3;  
a[1][0] = 4;  
a[1][1] = 5;  
a[1][2] = 6;
```

2)

```
a = new int[] [] { {1,2,3} , {4,5,6} }
```

Getting the dimensions: (refer to array of arrays picture)

number of rows: `a.length`

number of columns: `a[0].length`

`a[0]` is the first row element which is an array - the length of a row is the number of columns.

3) Each row can have a different number of columns. So leave the number of columns empty until initialization.

```
char [][] c = new char [2][];  
c[0] = new char[] {'a', 'b', 'c'};  
c[1] = new char[] {'d', 'e'};
```

4)

```
String [][] s = new String[4][4];  
int numRows = s.length;  
int numCols;  
// for each row  
for (int i = 0; i < numRows ; i++){  
    // for each column  
    numCols = s[i].length;  
    for (int k = 0 ; k < numCols; k++) {  
        s[i][k] = new String (" ");  
    }  
}
```

Inheritance

pg 142

inherit : when a class inherits from another class, it gets all the functionality from it. The inheriting class is called the subclass, or child class. The class it is inheriting from is the super class or parent class.

The subclass gets all the components of the super class: the variables, the methods. During execution, Java looks for the lowest method in the class hierarchy of the instance.

Syntax:

```
public class subclassName extends superclassName {  
    ...  
}
```

super:

super is the keyword used to access public components of the super class.

Eg: `super.toString()` will call the parent class's `toString` method instead of the one that is in your own class.

To call the super class constructor, use just the word **super** and pass to it the relevant parameters. Note that there is no period after the word **super** the way it is used when calling a procedure or function. In a constructor body, **super** can only be the very first statement.

Eg:

```
public class LazyStudent extends Student {  
    private String lazyHabits;  
    public LazyStudent (String name, String lazyHabits) {  
        super (name); // take out this call and see the error  
        this.lazyHabits = lazyHabits;  
    }  
}
```

If you don't explicitly make a call to **super** in the subclass constructor, java automatically puts a call to `super()` in implicitly and if the super class doesn't have a constructor with no parameters, the compiler will give an error.

Note : If you write NO constructors in a class, then Java automatically provides a default constructor which takes no parameters. But if you write a constructor yourself, then Java no longer provides an implicit default constructor and you must construct the object using the given constructor.

Eg:

```
public class Student {  
    private String name;  
    public Student (String name) {  
        this.name = name;  
    }  
}
```

A Student object can only be constructed by saying:

```
Student s = new Student ("bob");
```

in order to be able to say the following, you must provide a constructor which takes no parameters.

```
Student s = new Student();
```

memory model p 144

polymorphism, Casting , instanceof (p149-150)

An object can have a real and an apparent type.

```
LazyStudent ls = new LazyStudent ("bob", "I am lazy");  
Student s = new LazyStudent ("steve", " I want to watch tv");
```

Looking on the left hand side of the equals, you can tell that the APPARENT type of ls is LazyStudent, and the apparent type of s is Student. But when you look at the rhs, the real type of s is actually LazyStudent. To see what the real type of an object is, use instanceof

Eg:

```
if (s instanceof LazyStudent) {  
    System.out.println (LazyStudent) s.toString();  
} else {  
    System.out.println (s.toString()); // this will call Student's  
    toString b/c Student is the apparent type.  
}
```

- (C) e
 - o widening
 - is usually automatically done
 - C is apparent type or super class of e.
 - o narrowing
 - if C is the subclass (and the real type) while the apparent type of e is the super class.
 - This is needed to reassure Java that if this is leading to some loss of information, then we are intending to do it.

Eg:

```
LazyStudent s = new LazyStudent();  
Student ls = (Student) s; // this is widening - it is
```

automatically done.

```
LazyStudent ls2 = (LazyStudent) ls; // this is narrowing; only  
you know that s is a lazystudent but the computer won't know  
until run time.
```

Go through Student, LazyStudent, LazierStudent, ProcrastinatingStudent example.

-

Testing Java Classes

Unit Testing with JUnit:

It is very important that you test your Java classes to ensure they work correctly. Testing can be done by creating an instance of your class in the interactions pane, and well, trying lots of stuff.

The problems with this approach are:

1. The tests are not repeatable
2. The tests are not thorough
3. The tests take a long time to run multiple times

Instead, we use a formal testing method in Java, called "JUnit".

Unit testing is the process of creating and executing a thorough set of tests for some small "unit" of code. Generally, the units we work with in JUnit are the public methods of the class we are testing.

Step 1: Create a TestCase Class

Each class you want to test (Player, in this example), must have written for it a custom TestCase class. TestCase is a class that includes basic testing functionality, and we extend it to include tests for our specific class.

Here's an example:

```
// we need to bring this in from the API (it is the class we are  
// extending)  
import junit.framework.TestCase;  
  
// we have some methods we want to test that use "Date", so we'll  
// need that class too  
import java.util.Date;
```

```

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when
 * running
 * the test with JUnit.
 */
public class PlayerTester extends TestCase {

```

Step 2: Create Unit Test Methods

Now that the beginning of the class has been written, we create a set of unit-tests for each method in the class we are testing. We do this by creating a method in the tester class for each public method in the class being tested. The names must start with "test". For example:

```

/**
 * Test the constructor
 */
public void testConstructorAndGetName () {

```

Step 3: Create "assert" Statements to do Testing

You will now populate the testing method with code to create an instance of your class, and test whatever method(s) this unit will test. Each test is done in an "assert" statement. To "assert" something is to claim that it is true.

For example, "I assert that the Earth is round" means "I claim that it is true that the Earth is round". In Java, we use the "assertEquals()" method to make claims about our testing. Here's a complete, but basic, testing method based on the header written previously:

```

/**
 * Test the constructor
 */
public void testConstructorAndGetname() {
    // create a new Player
    Player p = new Player ("serena");

    // claim that getName should now return "serena"
    assertEquals("serena", p.getName());

    // a constructor can also be tested with toString
    assertEquals("serena:0:0", p.toString());

    // more tests go here...like testing that wins, losses
    // should be zero etc..
}

```

Step 4: Run Your Tester

You now have a class (Player.java), and a class that you can use to test it (PlayerTester.java). We will now run the tests.

All you need to do to run the tests is have the tester class open in Dr. Java, compile it, and click the "test" button. Dr. Java will test your code by running all the methods that start with the word "test", and take you to any assertEquals() method calls that assert false things.

Testing static methods:

To test a static variable, test the **change** in the variable.

Eg:

```
/**
 * Test the getOverallWins method.
 */
public void testGetOverallWins () {
    // create a new Player
    Player p = new Player ("serena");
    int tempWins = Player.getOverallWins();
    // make the player win.
    p.won();
    // check if the OverallWins got incremented by one.
    assertEquals(p.getOverallWins, tempWins+1);
}
```

Testing rules

- Design test cases first, before coding
- Use testing tools to help prepare test cases (example: junit).

Choosing test cases

Consider the set of all possible inputs to the code being tested.

Pick subsets to ensure that you've satisfied two kinds of completeness requirements:

- 1) Complete coverage of the kinds of data
 - a. positive, 0, negative
 - b. other boundary values: blank character, empty string, empty list

- c. 0, 1, 2; and large (or all if appropriate)
 - d. extremes: selected values at and near the beginning and end of a list
 - e. duplicates: list containing pairs, triples or all the same
 - f. special sets: sorted, reverse-sorted, all the same
- 2) Complete coverage of your code
- a. Your test values should cause every line of your program to be executed.
 - b. both ways in an if
 - c. each loop 0 times, 1 time, and >1 times

The two kinds of completeness interact. While doing (2) you'll think of things to add to (1), and vice versa.

Justify your test cases.

You've just made sure your test cases cover everything. Conversely, do you need all your test cases? If you can't explain why you're using a test case - or if the explanation is the same as for another test case - get rid of it. Make sure you comment your test cases eg: this tests an empty string, this tests an array of odd size etc..

Example 2: testing array reversal case by case instead of method by method.