A Brief Intro to Java Programming: The Basics of Java Code

By: Jacob Klimczak

Contributors

Author:

Jacob Klimczak

Editors:

Param Upadhyay

Zhengmao Ouyang

Chapter 1 – Intro To Java

Intro To Java

What Is Java?

Java is a programming language. This means it is used to give instructions to a computer. If you wanted to tell a robot how to move, or a computer how to interact with other computers, Java would be a good tool to accomplish this.

The instructions for the computer are written out in lines of what is called code. Each line is an instruction (or multiple instructions) that the computer executes. Lines of code are usually executed sequentially. This means that in most basic cases, one line of code is executed completely, and then the following line is executed, and so on.

Classes And Code

Code is written using a combination of English words and symbols. Java code is stored in *.java* files. These are like normal text files, they simply have the *.java* extension and can be compiled and run as Java code because of that. Each *.java* file contains a single *class*. We will talk more about classes later, but for now it is important to know that there is one per file. Every class has a name. The structure for naming a class is as follows:

```
class ClassName {
    //class contents go here
}
```

This would create a class called *ClassName*. Any class can be named in this manner; for instance, if you wanted to create a class called *Program*, then you would write:

```
class Program {
    //class contents go here
}
```

You may notice both of the examples above contain plain English text preceded two forwards slashes (*//*). This indicates what is called a comment. A comment is not an instruction for the computer,

in fact, the computer ignores it entirely. In Java any line preceded by // or any text in between a /* and a */ will be ignored by the computer. This text is only there to explain your code to someone reading it.

Entry Point

When you are writing code in Java, the computer has to know what instructions to run. It needs to know where to start. The computer starts at what is called the *main method*. It executes the first instruction in the *main method*, then the next, and so on. The *main method* looks like this:

```
class Program {

    public static void main(String[] args) {
        //this is where the program starts

        doStuff();
    }

}
```

In this case, the computer would start in the *main method*, skip the first line as it is a comment, and execute the *doStuff* method. You may notice that *doStuff()* is followed by a semicolon. This is because most operations in Java need to be followed by a semicolon. In general the only times you don't end you likes in semicolons are when designating sections in your code (i.e. *If statements*, *Loops*, etc…).

Printing

One of the most basic operations in Java is to *print* – display some text – some output on the console. This is accomplished with the *println* method. We will talk more about what a method is later, but for now, in order to *print* output to the console you will use the command:

```
System.out.println(s);
```

Where s is a *string* containing the text you would like to display. *Strings* will be covered in more detail later, right now you just need to know that if you wanted to *print* some text, you need to put that text in quotation marks and call the *println* method. For example, if you wanted to create a program with a class called *Program* that *prints* 'Hello World!' and then ends, you would do so like this:

```
class Program {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

This is what the output of the program would look like once it was run:

Hello World!

Note that the quotation marks are not *printed*. Only the text contained within them.

Questions

1) What does the following program output?

```
class Program {
    public static void main(String[] args) {
        System.out.println("Example Output In Quotation Marks");
    }
}
```

2) What is the problem with the following code?

```
class Program {
    public static void main(String[] args) {
        System.out.println("Some Output To Print")
    }
}
```

Answers

1) *Example Output In Quotation Marks*

2) Missing a semicolon at the end of the third line.

Chapter 2 – Variables

Variables

What Is A Variable?

A variable stores some kind of information. The information stored can be retrieved at any time. You can store text, numbers or even true or false in a variable. For example:

int x = 2;

This stores the number *2*, in the variable called *x*. The *int* at the beginning just tells the computer that the value you are assigning to the variable *x* is an integer (whole-number). The semicolon at the end just tells the computer that that line is finished. You need a semicolon at the end of most lines. You can also store text in a variable:

String y = "foo bar";

This stores the text *foo bar* in the variable *y*. The *String* bit at the beginning of the line tells the computer that you are assigning text (Text is called a string in programming) to the variable *y*. You need quotation marks around whatever string you are assigning to the variable *y*.

Data Types

Every variable or value has a data type. This data type tells the computer how to treat whatever variable it is dealing with. Without the data type, the computer has no idea if the value assigned to the variable is a number, text, or something else. For example:

int x = 2;

The computer needs the *int* at the beginning of the line to understand that the value assigned to *x* in an integer. All variables are given data types in this way. You just put the data type before the name of the variable. To start, we are going to deal with 4 different data types: strings, doubles, integers, and booleans.

An integer is any whole-number. 0, 1, 2, 7, 456, and 8 are all integers, while 7.8, -12.5, and 32.6 are not. In Java integers can be positive or negative, they just have to be whole numbers. An integer is defined like this:

int y = 7;

Doubles are also numbers, but they are more useful than integers. A double can be negative, and does not need to be a whole-number. 0.0, 1.0, 2.0, 7.0, 456.0, and 8.0 are all doubles, but so are 7.8, -12.5, and 32.6. A double is defined like this:

double z = 12.3;

Strings are text. When you define a string, you need to make sure to put quotation marks around it. This is how you define a string:

S*tring a = "foo bar";*

A boolean is basically a value that is either true or false. It can only be true or false, nothing else. Booleans are defined like this:

boolean b = true;

or

boolean c = false;

Operators

Operators represent basic mathematical operations. * is used for multiplication. / is used for division. + is used for addition. - is used for subtraction. For example:

double x = a + b;

The code above sets the double *x* to the sum of *a* and *b* whatever the values of those two may be. If you wanted to add b to a without defining a third variable, you could do it like this:

a = a + b;

The code above can be shortened into:

a += b;

Both of the examples above add *b* to *a*. The second one is just faster to type. The operators all work if you are dealing not only with variables, but with numbers too. For example, if you wanted to add 7 to a variable *d*, you could do it like this:

d += 7;

or

d = d + 7;

All of the operators work like this. If you wanted to set a variable *e* to *f* divided by *g*, you would do it like this:

double e = f / g;

But if you just wanted to divide *f* by *g* and set f to the result, you could do this:

f /= g;

The formats above work for all the operators. So if you wanted to multiply a variable *h* by 5, you could do it like this.

*h = h * 5;*

or

*h *= 5;*

If you wanted to subtract 3 from a variable *i*, you could do it like this:

i = i – 5;

or

i -= 5;

Questions

1) What does the following program output?

```
int a = 4;

int b = 5;

int c = 2;

System.out.println((b/a)*2);
```

2) Which of the following values would not be a valid value for an integer?

a. 7

b. 90554

c. 7.6

d. 8

3) Which of the following is not correct.

a. *double foo = "bar";*

b. *String bush = "The first big British band in America during the post Nirvana period";*

c. *String h = "no";*

d. *int i = 7;*

Answers

1) *2*

2) c.

3) a.

Chapter 3 – Conditionals

Conditional Statements

What Is A Conditional Statement?

A conditional statement is code that will only run if a certain condition is met. It is structured like this:

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

The code inside the *{}* won't run unless *condition == true* (the condition is met). An example of a valid if statement is as follows.

```
if (x == 7) {
        x += 1;
}
```

This code checks if the variable *x* is equal to 7. If it is, the computer adds *1* to *x*, changing *x* to 8. If *x* is not equivalent to *7*, the computer skips that portion of the code and doesn't change *x*. Comparisons and conditional statements can be made with any data type. For example, assuming you had a *String s* of an unknown value:

```
if (s.equals("Humphrey Bogart")) {
        System.out.println("here's looking at you kid");
}
```

This would only write *"here's looking at you kid"* if *s* was equivalent to *"Humphrey Bogart"*. You may notice that despite the fact we just stated the == operator was used to check if two values were equal, it is not used here. This has to do with the fact that Java is what is called *pass reference by value* for *objects* such as *Strings*. We will discuss this in greater detail later. For now, all that is important to remember is that when comparing whether two strings are equal, *.equals("")* must be used instead of the == operator. *s.equals("here's looking at you kid")*, returns true if the value of the string *s* is *here's looking at you kid*.

If Else Statements

An if else statement is used when you want to check one variable for many possible values, or when you want one of many possible outcomes to happen. For example:

```
if (q.equals("Yossarian")) {
        System.out.println("Catch 22");
} else if (q.equals("Gavin Rossdale")) {
        System.out.println("Bush");
}
```

This would write *"Catch 22"* if *s* was *"Yossarain"*, and *"Bush"* if *s* was *"Gavin Rossdale"*. The difference between this and a normal if statement, is that is the computer found that *s* was *"Yossarain",* it wouldn't bother checking if *s* was equivalent to *"Gavin Rossdale"*. Normally, the computer would check twice, once to see if *s* was equivalent to *"Yossarain"*, then again to see if *s* was equivalent to *"Gavin Rossdale"*. By using if else statements, the computer only needs to check once. This creates less work for the computer, and is generally a better practice.

Else

The else keyword is useful when you want your if statement to do something even if your condition isn't met. For example:

```
if (x == 2) {
        System.out.println("X is equivalent to 2");
} else {
        System.out.println("X is not equivalent to 2");
}
```

This code will write *"X is equivalent to 2"* when *x == 2*. It will write *"X is not equivalent to 2"* when *x != 2.* An else statement executes if none of the earlier if statements or if else statements run.

A complete set of conditional statements could look like this:

```
if (x == 1) {
        System.out.println("X is equivalent to 1");
} else if (x == 2) {
        System.out.println("X is equivalent to 2");
} else {
        System.out.println("X is equivalent to neither 1 nor 2");
}
```

Comparison Operators

The comparison operators are used to form the condition in your if statement. You can check if two variables are equivalent, not equivalent, or if one is greater than the other.

If you want to check if two values are equivalent, you need to use the operator ==. For example:

```
if (x == y) {
        doStuff();
}
```

If you want to check if two values aren't equivalent, you can use the operator *!*=. For example:

```
if (x != y) {
        doStuff();
}
```

The operator > checks if one variable is greater than another. < checks if one variable is less than another. >= checks if one value is greater or equal to another, and <= checks if one value is less than or equal to another. These are all used in the same way:

```
if (x > y) {
        System.out.println("X is greater than Y");
} else if (x < y) {
        System.out.println("X is less than Y");
} else if (x >= y) {
        System.out.println("X is greater than or equal to Y");
} else if (x <= y) {
        System.out.println("X is less than or equal to Y");
}
```

Questions

1) What is the following statement checking?

```
if (x == 2) {
        doStuff();
}
```

2) How could you rewrite this code to be more efficient?

```
if (x == 2) {
        doStuff();
}

if (x == 3) {
        doOtherStuff();
}
```

3) What is the problem with the following segment of code?

```
String s = "Hey";

String t = "No thank you";

if (s >= t) {
        doStuff();
}
```

Answers

1) This code is checking if the x variable is equal to 2.

2)

```
if (x == 2) {
        doStuff();
} else if (x == 3) { //no need to run this check if x is equal to 2, as x cannot be equal to 2 and 3
        doOtherStuff();
}
```

3) You cannot run a >= comparison on strings.

Chapter 4 – Arrays And Loops

Arrays & Loops

What Is An Array?

An array is a list of variables. Essentially, it is a list of information. Arrays are defined as follows:

int[] x = new int[] {2, 4, 6, 8};

This stores the numbers *2, 4, 6,* and *8* in the array called *x*. The *int* at the beginning tells the computer the data type of *x* (integer). The *[]* tells the computer that *x* is an array, not a variable. The values contained in an array can be retrieved as follows:

int a = x[0];

This stores the integer *2* in the variable *a*. The part of the line reading *x[0]* just tells the computer to use the first value in the array *x*. The index any array starts at 0 so if you want to reference the first value, you need to use *x[0]*, *x[1]* references the second value, *x[2]* the third, etc…

An array can only be one data type. All values contained by the array must be of that data type. If you wanted to create an array of strings, you would do so like this:

String[] b = new String[] {"Somebody", "once", "told me"};

The line above creates an array of strings where *b[0]* is *"Somebody"*, *b[1]* is *"once"*, and *b[2]* is *"told me"*.

Arrays have a fixed size. This means that if you create an array containing three values, you cannot add a fourth value. Sometimes, you may know how big you want your array to be, but not what values you want it to contain. For cases like this, code similar to the example below is used:

```
int[] y = new int[10];
y[0] = 1;
y[1] = 2;
y[2] = 3;
```

The line of code above creates an array of integers. The first line, tells the computer that the array can hold up to ten values. The next three lines define the first three values the first three values the array contains: *1*, *2* and *3*. Code like this can be used with any data type. For example:

```
String[] q = new String[5];
q[0] = "To be fair";
q[1] = "you need";
q[2] = "to have";
q[3] = "a very high IQ";
```

The code above created an array of strings called *q*. It defines the first 4 values, but leaves the fifth value blank.

Loops

Loops are frequently used in programming. They repeat certain sections of code until a condition is met. The two types of loops covered in this handout are while loops, and for loops.

A while loop, repeats any code contained within it, while a certain condition is true. A for loops repeats the code contained within it, a certain number of times.

The general structure of a while loop is as follows:

```
while (condition == true) {
        doStuff();
}
```

The *while* keyword tells the computer what type of loop you are using. The part in brackets next to that, is the condition that needs to be true for the loop to run. Once that condition is false, the computer will move past the loop. The part of the code in between the two curly-brackets, is just what you want the computer to loop through. The condition for the loop is checked once at the beginning, and before every repeat of the loop.

If you wanted a loop to write *"Hello World!"* on the screen while a *boolean b* was *true*, you would do it like this:

```
while (b == true) {
        System.out.println("Hello World!);
}
```

If you wanted to shorten that, you could write:

```
while (b) {
        System.out.println("Hello World!");
}
```

The *== true* comparison is never necessary. An *== false* comparison on the other hand, would still be necessary, although that can also be shortened.

For loops are structured as follows:

```
for (int i = 0; i < 10; i++) {
        doStuff();
}
```

The code above would execute whatever is contained within the brackets ten times. The *int i* part creates a new *int i* that is equivalent to *0*. The *I < 10* part tells the computer to repeat the contents of the loop until *i* is no longer less than *10*. The *i++* part is short for *i = i + 1*, this tells the computer to increase *i* by *1*, every time it repeats its contents. Once again, the part within the curly-brackets is just the code the loop repeats.

If you wanted to write *"Hello World!"* on the screen *7* times, you would do it like this:

```
for (int i = 0; i < 7; i++) {
        System.out.println("Hello World!");
}
```

Applications Of Arrays In Loops

Arrays are really useful when they are used in conjunction with loops. Say you had an array *s*. Imagine you wanted to print the contents of this array to the screen. A loop would be the fastest way to do this. For example:

```
String[] s = {"According to all", "known laws of", "aviation, there", "is no way a bee", "should be able to fly."}

for (int i = 0; i < s.length; i++) {
        System.out.println(s[i]);
}
```

Line by line, this is what the code above does. The first line creates the array *s* filled with various strings. The second line sets up the for loop. It creates the counter-variable *i*. Next, the second line tells the loop to stop when *i* is no longer less than the length of the array (for any array *x*, *x.length* returns the length of *x*). The final part of the second line tells *i* to increase by *1* with every iteration. The third line tells the computer to write out the contents of the array, by referencing the counter variable *i*. Since *i* increases by *1* every time the loop repeats, it will write a different line every time. The output from this code would be as follows:

```
According to all
known laws of
aviation, there
is no way a bee
should be able to fly.
```

There are more effective ways of looping through an array. For example:

```
String[] s = {"According to all", "known laws of", "aviation, there", "is no way a bee", "should be able to fly."}

for (String i : s) {
        System.out.println(i);
```

```
}
```

The output from this example is identical to the output from the example before it. It is simply easier to write. Loops through the array *s*. The variable *i* is used to store the value of *s* that the loop is currently on.

Questions

1) What does the following program output?

```
int a = 4;

int b = 5;

int c = 2;

int[] d = {b, a, c};

System.out.println(d[1]);
```

2) What is the problem with the following segment of code?

```
int[] a = new int[3];
a[0] = 3.5;
a[1] = 4.0;
a[2] = 6.7;
```

3) What is the problem with the following segment of code?

```
String[] s = {"Hey", "ya", "you"}

for (int i = 0; i < 4; i++) {
        System.out.println(s[i]);
}
```

Answers

1) *4*

2) The type of the array is int, but the values assigned are doubles.

3) The loop runs 4 times, accessing 4 array elements, but the array only has a length of 3.

Chapter 5 – Methods

Methods

What Is A Method?

A method is a placeholder for a bunch of code. It can accept variables as inputs and output a variable. For example:

```
int addTheseNumbers(int a, int b) {
        return (a+b);
}

System.out.println(addTheseNumbers(1, 8));
```

This code would print "9" to the console. The three lines above the *println* call define a method. The first line tells the computer the data type of the method, the name of the method, and the inputs that the method takes.

First, the *int* part of the line tells the computer that the method returns an integer. This means that this method can be used in any place where you would normally put an integer. The next part of the line names the method, in this case, the method is called *addTheseNumbers*. The part of the line between the brackets, where it says *(int a, int b)* tells the method what inputs it takes. This method takes two integer arguments. All of this sets up the method. The part of the method between, the parentheses (*{}* these symbols) tells the computer what to do whenever the method is called. In this case, all the method will do is add the two integer inputs. The *return* keyword tells the computer that the integer that this method outputs is *a+b*.

Another example of a method would be as follows:

```
void print(String s) {
        System.out.println(s);
}

print("E5, E5, G5, E5, D5, C5, B5");
```

This creates a method that writes whatever input it gets to the console. The first word in the first line *void*, tells the computer that this method doesn't return anything. This means that this method can't be used in the place of any kind of value. As you can see, it can still be used to replace its contents. Writing S*ystem.out.println("E5, E5, G5, E5, D5, C5, B5");* and *print("E5, E5, G5, E5, D5, C5, B5");* both do the same thing, write *E5, E5, G5, E5, D5, C5, B5* on the screen.

Method Data Types

Most methods have data types. The data type indicates what kind of value the method can be used to substitute for (this is also called the return type of the method because when methods are used, they "give back" a value).

The return type of a method is defined by whatever data type is put in front of the method. The first method in this package, *addTheseNumbers*, is an integer method because it has the word *int* before the name of the method. Methods can be used as a substitute for a value of the same return type as them. For example:

```
int a = addTheseNumbers(7, 5);
int b = addTheseNumbers(addTheseNumbers(3, 2), a);
int c = addTheseNumbers(a, b);
```

In the example above, the *addTheseNumbers* method is being used as a substitute for many integer values. First, it is used when defining *a*. The integer *a* is assigned the value of 7+5, or 12. The method is used twice when defining *b.* As you can see, it is even used inside of itself. This is not a problem. The method is usable anywhere where you would normally put an integer. The integer *b* is assigned the value of (3+2)+*a*. Since *a* is equal to 12, be is assigned 5+12, or 17. The final variable, *c* is assigned the value of *a*+*b*, or 12+17, which is equal to 29. In the end, *a* = 12, *b* = 17, *and* c = *29.*

The second method, *print*, is a method that doesn't return anything. This means that it can't be used to substitute for any kind of value. This is conveyed to the computer through the use of *void* before the name of the method. Having *void* before the name of your method means that whatever method you are defining doesn't return any value.

If you wanted to change the *print* method so that it could be used as a substitute for string, you would do this by changing the method's return type and adding a return statement. For example:

```
String newPrint(String s) {
        System.out.println(s);
        return s;
}
```

or

```
String newerPrint(String s) {
        System.out.println(s);
        return "In this example this method will always return this string";
}
```

Now the newP*rint* method can be used as a replacement for a string. The *newerPrint* method which we just created can be used in the same way.

Return

The return statement in any method is used to show the computer what the method is equivalent to when it is being substituted for a variable. For example:

```
String s = newPrint("The variable s will be assigned whatever input this method gets");
String t = newerPrint("The input here is irrelevant");
```

The code above writes two lines on the screen, and assigns values to two variables. The two lines it writes are: *The variable s will be assigned whatever input this method gets*, and *The input here is irrelevant*.

The values assigned are a bit different. The *newPrint* method return it's unchanged input, so the variable *s* is assigned the value *"The variable s will be assigned whatever input this method gets"*. The outcome here is the same as if you had written *String s = "The variable s will be assigned whatever input this method gets";*. The variable *t* gets assigned a value in a completely different way. If you look at the return statement in the *newerPrint* method, you might notice that the return statement has nothing to do with the input. Even thought the *newerPrint* method writes *The input here is irrelevant* on the screen, that is not what is assigned to the variable *t*. Instead, *t* gets assigned the value *"In this example this method will always return this string"*, much the same as if the code was *String t = "In this example this method will always return this string";*.

The value returned by a method must be the same as its return type. If you were to define a method *doStuff* as follows, it wouldn't work:

```
int doStuff(String a) {
        return (a + "XD");
}
```

This is because the method returns a string but the return type is integer. A method can only be used as a substitute for a variable of its own return type. The code below wouldn't work because of this:

```
int no = newPrint("E5, E5, G5, E5, D5, C5, B5");
```

newPrint returns a string, but *no* is an integer so the program would throw an error (not work).

Questions

1) Are there any problems in the code below?

```
void fillOutForm(String a) {
        return (a + "IS");
}

String s = kecksConcatenator("WHM");
```

2) Write a method that takes an integer input, and returns an array of strings where each string in the array has a length corresponding to its position in the array. The length of the array should be equal to your integer input. Print the output of that method to the screen.

Your code should look something like this:

```
String[] secretMethod(int a) {
	doSomeStuff();

	//...

	doSomeMoreStuff();

	//...

	thereAreProbablyAFewLoopsInHere();

	//...

	etc();
}

String s = secretMethod(5);
for (int i = 0; i < s.length; i++) {
	System.out.println(s[i]);
}
```

And output something like this:

```
$
$$
$$$
$$$$
$$$$$
```

Answers

1) The method has a void return type, but attempts to return a string.

Chapter 6 – Classes

Classes

What Is A Class?

As we have already discussed, the class is the fundamental unit in Java. But what is a class really? In essence a class is a way of structuring and storing information, as well as the ways to access and manipulate that information. A class is really just a data structure.

You could have a class representing a person. This class might store information such as the person's age, gender, name, and eye color. It might also contain ways to change that information, for instance a method that ages the person by one year. This class would look something like this:

```
class Person {
    String m_Name;
    int m_age;
    boolean m_isMale;
    EyeColor m_eyeColor;

    public Person() {

    }

    public Person(String name, int age, boolean isMale, EyeColor eyeColor) {
        m_name = name;
        m_age = age;
        m_isMale = isMale;
        m_eyeColor = eyeColor;
    }
```

```
        public void birthday() {
            m_age++;
        }
}
```

You may notice we reference the *EyeColor* type which does not seem to be defined. To rectify this, in another file you might have something like:

```
public enum EyeColor {
        GREEN, BROWN, BLUE, HAZEL, GREY;
}
```

This is what is called an *enum*. It represents a type with a fixed number of possible values. In this case any *EyeColor* variable have a value of *GREEN*, *BROWN*, *HAZEL*, *BLUE*, or *GREY*. You reference these values as follows:

```
EyeColor.GREY
```

So if you wanted to store an *EyeColor* you would do so as follows:

```
EyeColor color = EyeColor.BLUE;
```

Returning to the *Person* class, you may notice it has a few key features. Firstly, the first two methods in the class seems to be missing either a name or a return type. You may notice that the names of these methods seem to be the same as the name of the class. Incidentally, the types are also the same as that of the class. These are what is known as constructors. The constructors are the methods used to *instantiate* this class. This will be discussed in more detail in the next section. Moving on, the class contains another method: a *void* called *birthday*, and some variables. The variables names are all

prefaced with '*m_*' to indicate that they are *member variables*. That means that these variables are stored by the class, and not just accessible locally (within the same method). These variables are the information stored by the *Person* class, and *birthday* is the manner through which that information is manipulated.

Instantiating Objects

So, now we have a class which is a great way of showing how some information should be sorted, organized, and manipulated; however, this class contains no actual information. No values are ever assigned to any of its *member variables*. If we wanted to store information about one particular person. For example, to store information about a blue eyed, 31 year-old, man, named John, we would do the following:

```
Person john = new Person("John", 31, true, EyeColor.BLUE);
```

The code above *instantiates* the object. In essence it creates a variable with the *Person* type that stores the information we want. You can create as many instances of a class as you would like, and the information contained by one instance has no effect on the information contained by another. For example, if we ran the following code:

```
Person john = new Person("John", 31, true, EyeColor.BLUE);
Person jeff = new Person("John", 72, true, EyeColor.GREY);
Person barbara = new Person("John", 31, false, EyeColor.GREEN);
Person chanelle = new Person("Chanelle", 16, false, EyeColor.BROWN);
chanelle.birthday();
john.m_age = 34;
System.out.println(barbara.m_age);
System.out.println(jeff.m_age);
System.out.println(chanelle.m_age);
System.out.println(john.m_age);
```

The output would look like:

31

72

17

34

As you can see, changing John's age has no effect on Barbara's age, or on the ages of anyone else. Furthermore, when *chanelle.birthday()* is called, this increases Chanelle's age, but no-one else's. This is how classes work. John, Barbara, Jeff, and Chanelle are all *objects* (*instances* of a class). As you can see, classes are kind of like templates for creating *objects*.

Constructors

So what actually happens when we create an *object* from a class? Well, when we create an *object*, the computer calls the constructor for the class we are *instantiating.* So when we write:

Person jeff = new Person("John", 31, true, EyeColor.BLUE);

The computer looks at the *Person* class and searches for a *constructor* (which as we've already discussed is a method with the same name as the class, the return type shouldn't be specified because it will also be the same as the class) whose arguments match those provided. In this case, the computer checks the *Person* class and sees that there are two *constructors*. The first one has no parameters, but the parameters of the second match the argument data types: *string*, *int*, *bool*, *EyeColor*. For this reason the computer calls the second constructor, which works from that point on as a normal method and assigns the values provided as arguments to the *member variables* in our new object. If instead of:

Person jeff = new Person("John", 31, true, EyeColor.BLUE);

We were to call:

```
Person jeff = new Person();
```

The blank constructor, taking no parameters would be run. You may notice this one does not do anything, so all the *member variables* would be unassigned. This means that if you created a new *Person* in the manner shown above, assigned it to *jeff*, and then did the following:

```
int a = 1 + jeff.m_age;
```

The computer would get confused and *throw* an *Exception* (a *NullPointerException* to be precise). This is because no value was ever assigned to the *m_age* variable. Because of this when you try and reference it, the computer doesn't know what to do.

Static Fields

As we have previously stated, in Java one object's methods and *member variables* do not equate in any way with the methods and *member variables* of another object, even if the two objects are of the same type and are *instantiated* from the same class. There is one exception to this however: *Static* fields. Any method or *member variable* marked *static*, can be though of as not being stored in the instances and instead being stored directly in the class. This means that changing a *static* field changes it for every *instance*, not just for one. In general, anything *static* should not be accessed through an *instance*. Say for example you had a class as follows:

```
class ExampleClass {

    public static int staticInt = 1;

    public int nonStaticInt = 3;

    public ExampleClass() {

    }
```

```
    public void nonStaticMethod() {

    }

    public static void staticMethod() {

    }

}
```

This class has a mix of *static* and *non-static* members. If we were to do the following:

```
ExampleClass c = new ExampleClass();
ExampleClass d = new ExampleClass();
c.nonStaticInt = 4;
System.out.println(c.nonStaticInt);
System.out.println(d.nonStaticInt);
c.staticInt = 5;
System.out.println(c.staticInt);
System.out.println(d.staticInt);
```

The output would look like this:

```
4
3
5
```

5

As you can see, changing the *non-static* property in one *instance* did nothing to the value in the other, but when the *static* value was changed in one, it changed in both. It is important to note that accessing *static* members through an *object* is bad practice and should be avoided. The proper way to access a *static* member does not require an *object*. It is as follows:

ExampleClass.staticInt

As you can see you do not need an *object*. If you wanted to change this value and print it to the screen you could do the following:

```
ExampleClass.staticInt = 17;
System.out.println(ExampleClass.staticInt);
```

This would output:

17

Static methods work in the exact same way. A *static* method will work the same wherever. It is important to note that since *static* members are not associated with an *instance*, when writing them you cannot reference *non-static* members which are associated with *instances*. A good way to remember the difference between *static* and *non-static* members, is that a *non-static* member will be created separately for every instance, while a *static* member will only be created once for the whole class.

Questions

1) What is the problem with the following code?

```
class DummyClass {
        int a = 2;
        public DummyClass() {

        }
}

class Program {
        public static void main(String[] args) {
                System.out.println(DummyClass.a);
        }
}
```

2) How would you instantiate an object of the dummyClass class?

Answers

1) DummyClass.a is accessed like it is static when it is not.

2) DummyClass dummy = new DummyClass();