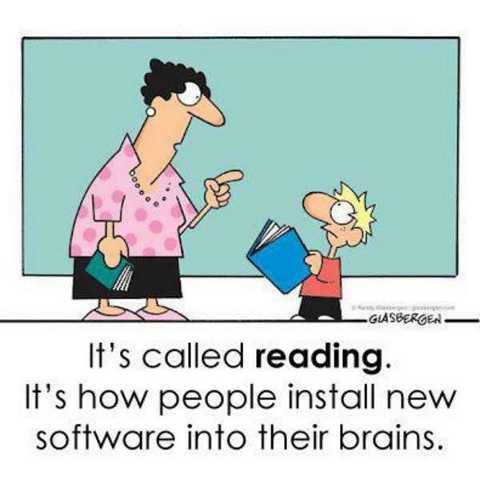
****

**Just Enuf Java**

**© 2014 Jason Leigh**

**University of Hawaiʻi at Mānoa**

**Version** February 23, 2016

1. Primitives 2

Declaring and assigning variables 2

Operators 2

Short cut operators 3

2. Comparison/Boolean Operators 3

If statement 4

Else Statement 4

3. Loops 4

While loops 4

Do loops 5

4. Classes and Object Oriented Programming 5

Defining a class 5

Declaring a variable of that class and Instantiating an object 7

5. Reading Typed Information from the Keyboard into Your Program (the Scanner and InputStreamReader) 7

6. Arrays 8

7. The For Loop 10

8. Switch Statement 11

9. Static Final Members 12

10. Main() & Static Member Functions 14

11. Scope 15

12. Reading from a File and Text Parsing (the Scanner & FileReader) 16

13. Another File Reading and Text Parsing Method (the BufferedReader) 18

14. Writing to a File (the BufferedWriter & FileWriter) 21

15. Recursion 21

16. 2D Arrays 23

17. Privacy in Objects (Public / Private / Protected / Package Private) 24

18. ArrayList 26

19. Inheritance 29

20. Polymorphism 32

21. Try/Catch/Finally – Error Handling 33

22. The End? 35

# Primitives

## Declaring and assigning variables

In every programming language variables are used to store information for the program.

A variable is like a thumbdrive or memory stick that you can **only put one thing** at a time into. If you put more than one thing into the thumbdrive, the last thing you put in will simply replace what was already there.

Variables come in different types to represent the type of information they can store. E.g. The int type can only store integers; the float type can only store floating point numbers.

When you store something in a variable, the term is *assigning a value to a variable.*

Here are examples of declaring and assigning values to variables:

int foo; // declare foo as an integer

int foo = 5; // declare foo as an integer and give it a 5

float foo; // declare foo as a floating point number

float foo=3.1416f; // declare foo as a floating point number and

// assign it 3.1416. The f indicates it's a float.

String myName; // declare myName as a string

String myName = “Jason”; // declare myName as a string and assign it “Jason”

## Operators

Operators are, well, operations you can do with variables.

e.g. foo = foo + 1; // take the value of foo and add 1 to it,

// then store the result back in foo

Other operators include:

+ means plus

- means minus

/ means divide

\* means multiply

e.g.

bar = foo \* baz; // take the value of foo and multiply by baz and store

// the result into bar

Note: the order in which the operators are performed is as follows: multiply and divide are done before plus and minus.

e.g.

bar = foo \* baz + cow / bell;

foo and baz are multiplied together, cow is divided by bell, then finally the two parts are added together.

Itʻs like saying:

bar = (foo\*baz) + (cow/bell); where the parentheses help group the order stuff is done.

So if you are ever in doubt about the precedence ordering, just put a set of parentheses in to force the order you want.

e.g.

bar = foo\*(bar + cow) / bell;

this adds bar to cow first, then foo is multiplied by the result and lastly it is divided by bell.

## Short cut operators

++ means increment (increase by 1)

-- means decrement (decrease by 1)

e.g.

foo++; // increase the value of foo by 1.

Same as saying: foo = foo + 1;

e.g.

foo--; // decrease the value of foo by 1.

Same as saying: foo = foo – 1;

+= means add to

-= means subtract from

e.g.

foo+=5; // add 5 to foo

Same as saying foo = foo + 5;

e.g.

foo-=5; // take 5 away from foo

Same as saying foo = foo – 5;

The ++ and –- operators can also be used in the following way:

foo++;

++foo;

When the ++ is before a variable (such as foo), it means increment the variable first then return its value.

When ++ is after the variable, it means return its value and then increment the variable.

Consider the following two examples to see the difference:

E.g. 1

int a = 0;

int x = 0;

a = ++x;

This is the same as saying:

x = x + 1;

a = x;

In this case a will be 1, and x will be 1.

E.g. 2

int a = 0;

int x = 0;

a = x++;

This is the same as saying:

a = x;

x = x + 1;

In this case a will be 0, and x will be 1.

# Comparison/Boolean Operators

A comparison operator lets you compare two variables, and the result is a *boolean*. A boolean value is either true or false.

Here are examples of Java comparison/boolean operators and their meanings:

x == y means x equals y

x != y means x not equals y

x > y means x is greater than y

x < y means x is less than y

x >= y means x is greater than or equal y

x <= y means x is less than or equal y

|| means OR – see below for example

&& means AND – see below for example

Boolean operators arenʻt very useful on their own. They need to be used as part of an **if** statement or loop statement, which are described next.

## If statement

The **if** statement is program statement to allow you to perform an action when a certain condition in your program becomes true or false.

e.g.

If (**condition** is true) {

//do your thing here if the conditional was true.

}

e.g.

if (x > y) {

// if X is greater than Y then this part of the program is executed.

}

The curly braces { } is often called a *code block.* I.e. a block of code. Think of it as a small musubi of program code wrapped not in seaweed but curly braces.

Another e.g.

if ((myName **==** “Jason”) **&&** (height < 4)) {

// if my name is Jason AND my height is less than 4 then print something…

System.out.println(“I am a Hobbit”);

}

## Else Statement

**Else** is an extension of the **if** statement and lets you write code to deal with situations if an if condition is not true. Here is the general form of how else statements are written:

if (conditional is true) {

// if the conditional was true then this part of the program is done.

} else {

// otherwise this part is done.

}

# Loops

Loop statements let you repeat the actions of your program if or until a particular condition is met.

## While loops

while (conditional is true) {

// do your thing here if the conditional was true.

}

e.g.

while (count < 100) {

// while count is less than 100 then do stuff in here.

}

while(true){

// This basically says the condition is always true so always

// go into this while loop. In other words it is an INFINITE loop.

}

## Do loops

Do loops are essentially the same as while loops except the condition to return to the top of the loop is tested at the end of the { }

do {

// do something here

} while (conditional is true);

e.g.

do {

System.out.println(“Hello my name is Jason”);

count = count + 1;

} while (count < 10); // only if count is less than 10 will we return to

// the top of the do statement.

# Classes and Object Oriented Programming

Javaʻs most powerful feature is a programming **paradigm (fancy word for technique or approach or methodology)** called Object Oriented programming. Basically you try to represent the information in your program by mimicking objects in the real world- like a watch or a TV. A watch stores information (like the time) and it has functions like showing you the time or letting you set its time.

To start object oriented programming you need to *define* a **class.** A class is like a *blueprint* which you can use over and over again to make the same thing (called objects) again and again. Inside a class you need to *declare* variables that the class will need to hold important information about itself. In our what example below you might have variables for time. Inside the class you will also typically declare functions that you think a class should be able to perform. In our watch example we said showing you the time or letting you set its time are typical functions of a watch. Lastly once you have the class declared you want to actually use that blueprint to start making objects. The technical term for “making” an object is to “instantiate” it. So hereʻs an example of declaring a class for a watch.

## Defining a class

e.g. This is a class to define a simple watch that can store hours and minutes. It also lets you ask the time and change the current time.

public class watch { // this is starts by defining a class called watch

int hours; // this declares a variable for the watch called hours

int minutes; // this declares a variable for the watch called minutes

int getHours() { // this declares a function of the watch called getHours

return hours; // which gives you the current value stored in hours

}

int getMinutes() {

return minutes;

}

void setTime(int thehour, int theminute) { // this sets the time

hours = thehour;

minutes = theminute;

}

watch() { // this is called a **constructor**

setTime(12,0);

}

};

The variables in a class (hours and minutes) are called *member variables,* or *instance variables.*

The functions of a class (getHours, getMinutes, setTime) are called *member functions,* or *methods.*

The constructor of a class is a special kind of member function that is called whenever an object of that class is created/instantiated.

**Every time you define a class you will almost always need member variables, member functions, and constructors.**

So in the above example the member variables are: hours and minutes.

The member functions or methods are: getHours(), getMinutes(), setTime(), watch().

The constructor is: watch().

Note for example in getHours:

int getHours () {

return hours;

}

The int in front of getHours means that the programmer intended the member function to return an integer value as a result of calling (or executing) the function. And sure enough inside the block of code, there is a statement that says: return hours. Which basically means *return* whatever is stored in hours as the result of calling the getHours() function.

Now look at setTime:

void setTime(int thehour, int theminute) { // this sets the time

hours = thehour;

minutes = theminute;

}

The *void* in setTime() means that the programmer would not like the function to return any values when it is called. Hence there is no *return* statement. But look at whatʻs inside the parentheses of setTime(), it says: (int thehour, int theminute)

That means setTime is expecting you to give it two integers and the function store those two integers inside the variables thehour, and theminute. These variables inside ( ) they are called *parameters.*

So setTime has two parameters of type int called thehour and theminute.

What setTime does with those values is seen in the code block, which says:

hours = thehour;

minutes = theminute;

This copies the values from the parameters into the member variables of the watch. So now watch holds the new time.

## Declaring a variable of that class and Instantiating an object

So now that you have defined a class, it is time to use it. First you need to declare a variable, and then create (or instantiate) an object of that class type.

e.g.

watch myWatch; // this declares a variable of type watch called myWatch

myWatch = new watch(); // this instantiates an object of type watch and

// assigns it to myWatch

Now lets print out the time of my watch by using the member functions getHour() and getMinute().

System.out.println(“The time is: Hours: “ + myWatch.getHours() + “ and Minutes: “ + myWatch.getMinutes());

Notice: myWatch.getHours()

This basically says: using the myWatch object, call the getHours() function. The “.” reaches into the myWatch object to let you access the getHours() function.

If you do the above the resulting printout will be:

The time is: Hours: 12 and Minutes: 0

Where did the 12 come from? Whenever an object is created/instantiated, its *constructor* is called. In this case the constructor calls the member function setTime() and gives it two parameters 12 and 0.

watch() { // this is called a **constructor**

setTime(12,0);

}

… just like a microwave oven thatʻs had its power pulled, it reads 12 oʻclock.

If you were to instantiate another watch, like this:

watch mySecondWatch;

mySecondWatch = new watch();

It will also be set to 12 oʻclock by default.

What if we wanted to set the time to something else? Then we would use our setTime() member function and give it our own values.

e.g.

mySecondWatch.setTime(5,35);

This calls mySecondWatchʻs setTime function and gives it 5 and 35 as parameters, which are then used to set the watchʻs time.

# Reading Typed Information from the Keyboard into Your Program (the Scanner and InputStreamReader)

Here we going to introduce you to how to get your program to read text typed in from the command prompt. This was used in the Guess A Number program.

The following two lines go at the top of your program file. They tell Java to import two Java libraries, the InputStreamReader and the Scanner.

**import java.io.InputStreamReader;**

**import java.util.Scanner;**

The InputStreamReader is the library you use to read from the keyboard. The Scanner allows you to parse the text from the InputStreamReader.

So now look at these 4 lines of code:

InputStreamReader keyboardInput = **new** InputStreamReader(System.*in*);

Scanner myScanner = **new** Scanner(keyboardInput);

System.*out*.print("Guess a number from 0 to 100? ");

int guess = myScanner.nextInt();

The first line: InputStreamReader keyboardInput = **new** InputStreamReader(System.*in*);

Tells Java to create an InputStreamReader object and assign it to the variable keyboardInput. The parameter System.in just tells the InputStreamReader that we want to create an input stream that reads directly from System.in. In Java System.in is essentially the command line.

The second line: Scanner myScanner = **new** Scanner(keyboardInput);

Tells Java to make an object of type Scanner and assign it to myScanner. This scanner object will be used to interpret the text received by the keyboardInput object.

The System.out.println command merely prints the message to the screen.

Lastly: int guess = myScanner.nextInt();

Means to try to read an integer from the myScanner object and give that integer value to a variable called guess.

You can also read floats too, if you are expecting the user to type in a floating point number, like this:

float myFloatGuess = myScanner.nextFloat();

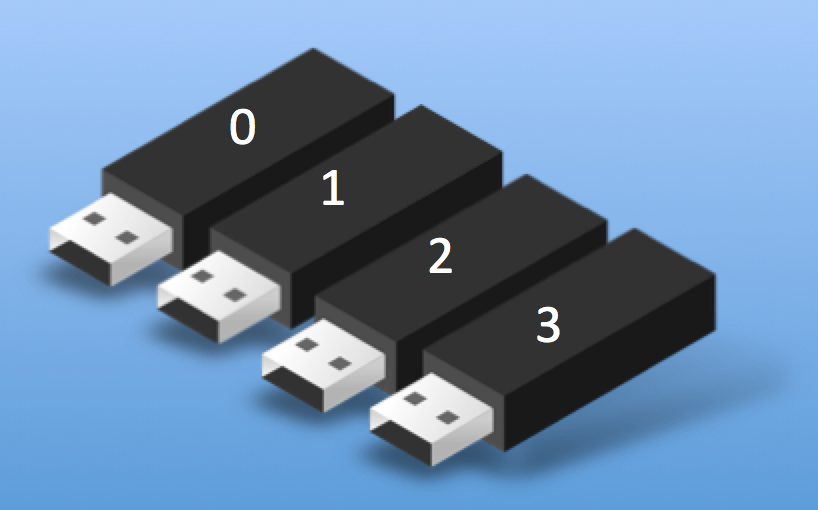
If you just wanted to get the next item as a String then you would do something like this:

String myStringInput = myScanner.next();

This isnʻt the last time you will encounter Scanner. You will see it again later when we deal with reading from a file.

# Arrays

If a variable is like a thumbdrive you can only store one thing into, an *array* is a collection of thumbdrives of the same type.



E.g.

int[] foo= new int[4];

This makes an array called foo that can hold 4 integers. Notice the int[]. This is how you tell Java that the foo variable is of type integer array. And notice it says foo = new int[4];

This tells Java to make an array of size 4 of type integer and foo holds onto that array for you so you can do interesting things with it.

So once you have an array you want to store things in it and also read from it.

e.g. if I wanted to store 100 into each of the 4 array entries / items I could do this:

foo[0] = 100;

foo[1] = 100;

foo[2] = 100;

foo[3] = 100;

Notice that arrays are **indexed** starting with zero. I.e. the first array item is foo[0]. And the last array item in array of 4 integers is foo[3].

If I were really clever I would use a loop to make this task easier, like this:

int[] foo = new int[4];

int currentItem = 0;

while (currentItem < 4) {

**foo[currentItem] = 100;**

currentItem++;

}

This loop would be especially useful if you wanted to assign 100 to an array with thousands of entries, for example. It would save you typing in thousands of lines of code, unless you want typing practice.

What if I want to make an **array of objects** rather than just primitive variables?

E.g. make an array of watches.

watch[] myWatches = new watch[4];

int currentItem = 0;

while(currentItem < 4) {

myWatch[currentItem] = new watch();

currentItem ++;

}

This little program made an array variable called myWatches with 4 entries that each hold 1 watch.

After declaring the array you have to create/instantiate a watch object and put it into each entry, like this:

myWatch[currentItem] = new watch();

What if I want to print the time in each of these 4 watches.

watch myWatches[4];

int currentItem = 0;

while(currentItem < 4) {

myWatch[currentItem] = new watch();

System.out.println(“The time is: Hours: “ + **myWatch[currentItem].**getHours() +

“ Minutes: “ + **myWatch[currentItem].**getMinutes());

currentItem ++;

}

# The For Loop

A very popular kind of loop in Java (and C++) is the For loop.

e.g.

for (int count = 0; count < 5; count++) {

// do something inside the body of the for loop.

System.out.println(“COUNT IS: “+count);

}

This program will print out:

0

1

2

3

4

Now look more closely at the *for* part of the for loop: **for ( int count = 0; count < 5; count++)**

The first part says: int count = 0; this declares a variable called count and assigns it a zero.

The second part says: count < 5; like in the while loop this is a conditional. Ie do the stuff inside the {} of the for loop as long as this condition is true.

The third part says: count++; which means to increment the counter.

Here is another example of using a for loop:

for (int j = 1; j <=5; j=j+2) {

System.out.println(“COUNT IS: “+j);

}

This program starts with j at 1 and continues as long as j <=5. It also increments j by 2 each time through the loop.

So the program would print:

1

3

5

Hereʻs yet another example. This time I will use the earlier example from the array of 4 watches and rewrite it with a for loop. Notice now that the code is more compact.

watch myWatches[4];

for (int currentItem = 0; currentItem < 4; currentItem++){

myWatch[currentItem] = new watch();

System.out.println(“The time is: Hours: “ + myWatch[currentItem].getHours() +

“ Minutes: “ + myWatch[currentItem].getMinutes());

}

For comparison, here is the old code still using the while loop:

watch[] myWatches = new watch[4];

int currentItem = 0;

**while(currentItem < 4) {**

myWatch[currentItem] = new watch();

System.out.println(“The time is: Hours: “ +

**myWatch[currentItem].**getHours() + “ Minutes: “ +

**myWatch[currentItem].**getMinutes());

currentItem ++;

}

# Switch Statement

A switch statement is like having a series of if-else statements one after another.

An example illustrates the idea:

int a;

a=2;

if (a == 1) {

System.out.println(“January”);

} else if (a == 2) {

System.out.println(“February”);

} else if (a == 3) {

System.out.println(“March”);

} else if (a == 4) {

System.out.println(“April”);

} else if (a == 5) {

System.out.println(“May”);

} else if (a == 6) {

System.out.println(“June”);

} else if (a == 7) {

System.out.println(“July”);

} else if (a == 8) {

System.out.println(“August”);

} else if (a == 9) {

System.out.println(“September”);

} else if (a == 10) {

System.out.println(“October”);

} else if (a == 11) {

System.out.println(“November”);

} else if (a == 12) {

System.out.println(“December”);

} else {

System.out.println(“Thereʻs no such month!”);

}

You can replace all this with a simple switch statement like this:

int a;

a = 5;

switch(a) {

case 1:

System.out.println(“January”);

break;

case 2:

System.out.println(“February”);

break;

case 3:

System.out.println(“March”);

break;

case 4:

System.out.println(“April”);

break;

case 5:

System.out.println(“May”);

break;

case 6:

System.out.println(“June”);

break;

case 7:

System.out.println(“July”);

break;

case 8:

System.out.println(“August”);

break;

case 9:

System.out.println(“September”);

break;

case 10:

System.out.println(“October”);

break;

case 11:

System.out.println(“November”);

break;

case 12:

System.out.println(“December”);

break;

default:

System.out.println(“Thereʻs no such month!”);

}

The syntax for the switch statement is as follows:

**switch** (***variable***) {

**case** *value*:

… code …

**break;**

**default**:

… code …

}

Note: if you wish to use a String variable in a switch make sure you have at least Java 8. Anything before Java 8 does not accept a String as a variable in a switch statement.

# Static Final Members

Static Final Member variables are generally used to store constant values that might be reused often in your program.

Take PI for example- which is 3.141592653589793238.

Lets say you are writing a fancy math program and your program makes use of that number a lot. You wouldnʻt want to type in 3.141592653589793238 every time you want to use PI. You could make a mistake and mistype one of the digits. I suppose you could do a cut and paste job. But then lets say you wanted to change all the PIs in your program to just 3.1415. Then youʻd have to make sure your cut and paste job didnʻt miss any of the 3.141592653589793238 out there in your code.

So naturally it makes sense to want to assign it to a variable like this:

float PI = 3.141592653589793238;

So now in your program you can just do this:

To calculate the area of a circle, which is, PI \* radius2 the code might look like this:

areaOfACircle = **PI** \* radius \* radius;

And to calculate circumference itʻs PI \* diameter

So your code might look like this:

circumference = **PI** \* diameter;

This is certainly better than typing 3.141592653589793238 every time. But itʻs not fool proof.

By that I mean that since you have made PI a variable, it is quite easy for you to accidentally change that number in your code like this:

PI = 9;

and then mess up any future calculations that depend on PI. Java wonʻt stop you. It couldnʻt care less what PI *really* should be. It can only do what you tell it.

To help guard against this, what you really want instead is to treat the PI variable as a *constant* that cannot change, and if you tried to change it Java will barf at you for trying.

So hereʻs how you do it, using static final member variables:

**static final** float PI = **3.1415**92653589793238;

**static final** long speedOfLight = 299792458;

Ok so whatʻs with this *static* and *final* business?

*Final* means the value stored in the variable wonʻt change once itʻs been given a value the first time. So PI cannot be changed by any other part of a program. If you decide later to write something like this:

PI = 2.1;

Java will barf at you because you said it was **final**, and final is final. Java wonʻt let you change it.

But what about *static*? In Java *static* variables are variables that **belong to the class only** **and are shared by all objects of the class.** In contrast non-static variables (i.e. regular member variables) are variables that belong to each individual object and are therefore independent of each other. They are not shared. Confused?

Hereʻs an example to hopefully clarify this concept:

We have 2 identical classes (exampleA and exampleB except that foo is declared differently in each case):

class exampleA {

int foo;

};

class exampleB {

**static** int foo;

};

exampleA object1 = new exampleA();

exampleA object2 = new exampleA();

object1.foo = 5;

object2.foo = 10;

exampleB object3 = new exampleB();

exampleB object4 = new exampleB();

object3.foo = 100;

object4.foo = 200;

System.out.println(“OBJECT 1 foo is: “ + object1.foo);

System.out.println(“OBJECT 2 foo is: “ + object2.foo);

System.out.println(“OBJECT 3 foo is: “ + object3.foo);

System.out.println(“OBJECT 4 foo is: “ + object4.foo);

This program would print:

OBJECT 1 foo is: 5

OBJECT 2 foo is: 10

OBJECT 3 foo is: 200

OBJECT 4 foo is: 200

The foo in OBJECT 3 and OBJECT 4 share the same value because it refers to exactly the same variable- the *static* variable. The static variable belongs to the class and is shared by all the objects- hence they have the same value.

Whereas foo in OBJECT1 and OBJECT2 have their own individual variables called foo. As a result they can hold different values for foo.

Still confused?

Just stick to this rule of thumb:

**If you are declaring a variable which every object of a class should share then make the variable *static*. And if the value should be a constant then make it *final*.**

**CAVEAT**:

While the above code example will run in Java, we donʻt normally use static member variables in the way shown. I.e. in the above example we referred to the static member variable using the following syntax:

object3.foo = 100;

object4.foo = 200;

In fact the proper way to refer to static member variables is to write this:

class\_name.variable\_name

for example:

exampleB.foo = 100;

exampleB.foo = 200;

The reason is that, remember **a static member variable belongs to a class NOT an object** and so it makes sense that to reference it you should use the name of a class (in this case the class name was exampleB) and not an object.

# Main() & Static Member Functions

Every time you create a new program in Java you have to define a class.

When you use Eclipse and create a new project and your first class it will typically look something like this:

**public** **class** HelloWorld {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

}

}

In this example the class created is called HelloWorld. The class has a member function called **main().** Java uses this to figure out where to begin execution of your program.

The **public static void** in front of main() takes some explaining. You already know what *void* means. It means the main function is not expected to return anything after it is done executing.

Public is something we will explain later. But for now just think of it as meaning that it is publically accessible.

The **static** is used in the same context as **static final members** that we encountered in the previous section. **Static basically means the member function belongs to the class and not to the object that is created/instantiated from the class.** That basically means there is only 1 main function and not an individual main function for each object of the class. Member functions that are declared as static are usually called **static member functions.**

Since a static member function belongs to the class and NOT an object it is possible to call a static member function without an object.

For example, Java has a library for Mathematics functions like sin(), cos() and sqrt().

double squareroot = Math.sqrt(100);

Math is the class and sqrt is the static member function.

You do NOT call sqrt by creating an object first such as in this **incorrect** example:

Math obj = new Math();

double squareroot = obj.sqrt(100); 🡨 WRONG

This is the method for calling member functions that are NOT static. I.e. the traditional member functions youʻve been using up to this point.

Regarding Main(): Something to be aware of is: when the pros write Java programs they will typically keep the main program simple, but create lots of different classes that the main program might access. So it is common to see one java file representing the main program, and many other java files representing all the other classes you define to support your program. So it is usually a good habit to follow this convention. In some of the examples presented in the class we didnʻt follow this convention, only because we hadnʻt sufficiently introduced the notion of classes yet. Now that you know what a class is you should try to follow this convention.

# Scope

Scope refers to the lifetime and accessibility of a variable. How large the scope is depends on where a variable is declared. For example, if a variable is declared at the top of a class then it will be accessible to all of the class member functions. If it’s declared in a member function then it exists only in that member function and can only be used in that member function.

For example:

class myClass {

int foo;

myClass() { // The constructor initializes foo to 5

foo = 5;

};

void mf1() {

int bar = 77;

System.out.println(“In mf1 Foo contains: “+foo);

System.out.println(“In mf1 Baz contains: “+baz);

}

void mf2() {

int baz=9;

System.out.println(“In mf2 Foo contains: “+foo);

System.out.println(“In mf2 Bar contains: “+bar);

}

};

If I were to use this class as follows:

myClass anObject = new myClass();

anObject.mf1();

The program would print 5 and then crash because foo is declared as a member variable and therefore all the member functions have access to foo. But baz is declared only within the scope of mf2, so mf1 isnʻt even aware of bazʻs existence. As a result Java will flag an error in your code with a message to the effect that baz does not exist.

If we were to run this code:

myClass anObject = new myClass();

anObject.mf2();

The program would again print 5 for foo. But since bar is declared in the scope of mf1 and not mf2, bar would be an error.

**So in general whenever you see { } and you decide to declare variables within those curly braces, those variables only exist within those braces.** They disappear from existence the moment the program leaves those braces. They are like sub-universes you can enter into. But the moment you leave everything that was declared inside that universe disappears.

For example in this code:

{

int foo = 5;

{

int bar;

bar = foo;

}

foo = bar;

}

bar = foo is ok and bar would hold 5, because bar is in a scope surrounded by foo and therefore is aware of fooʻs existence.

However after the closing of the {}, bar no longer exists and therefore foo=bar would yield an error, because at that point bar no longer exists.

# Reading from a File and Text Parsing (the Scanner & FileReader)

Sometimes rather than giving the computer input from the keyboard or from a mouse you want the program read it from a file.

Lets say you have a file containing text like this:

Jason 21 M 5.11

Jennifer 10 F 5.8

Joseph 15 M 5

In this example, as you might guess, the first entry consists of the name, the next is age, then sex and lastly height.

So lets save this file away and call it something like: **peopledata.txt**

Now lets try to read the file and assign it to variables in our program.

First you need to include the following two lines of code in your program:

import java.io.\*;

import java.util.\*;

These tell Java that we are going to make use of the java.io and java.util libraries. Donʻt worry whatʻs in those libraries. Weʻll get to that in a moment.

In Java, to open a file to read, you need to do something like this:

Scanner fs = new Scanner(new FileReader("peopledata.txt"));

This line of code declares a variable called fs of type Scanner and assigns it an object of the same type, which takes as a single parameter, a file reader that opens a text file called peopledata.txt

Youʻve already encountered Scanner earlier (to jog your memory check out section 5).

FileReader is the Java library class that actually opens a file to be read. The Scanner uses the File Reader to read a chunk of text at a time (parse) under your programʻs control.

So lets say you wanted to read a bunch of information in a text file called peopledata.txt, which happens to contain, people data like this:

Jason 21 M 5.11

Fred 32 M 5.3

Jennifer 25 F 4.5

You would write something like this:

Scanner sc = new Scanner(new FileReader("peopledata.txt"));

String line;

while (sc.hasNextLine()) {

String name;

int age;

String sex;

float height;

name = sc.next();

age = sc.nextInt();

sex = sc.next();

height = sc.nextFloat();

System.out.println(“NAME: “ + name + “ AGE: “ + age +

“ SEX: “ + sex + “ HEIGHT: “ + height);

}

sc.close();

The line that says: while(sc.hasNextLine()), basically checks to see if there are any lines of text available to be read. If the answer is true then it proceeds.

Following this you see lines of code that say:

name = sc.next();

age = sc.nextInt();

sex = sc.next();

height = sc.nextFloat();

This should be pretty straightforward. The first line attempts to read the next token as a string and assign it to the String variable name. The second line attempts to read an Integer and assign it to age. The third reads another text string, and the last reads a floating point number and assigns it to the variable height.

Lastly, the System.out.println prints the contents of the variable out so you can verify that everything was behaving as you expected.

Something that might be useful to you is the following:

Scanner also has additional member functions to do look ahead. E.g.:

if (fileScanner.hasNextFloat()) {

aFloatValue = fileScanner.nextFloat();

}

Basically this code looks ahead to see if there is a float coming next before attempting to read it.

Other look ahead member functions include: hasNext(), hasNextInt(), hasNextDouble(), etc..

For more on the Scanner see:

<http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html>

One last thing: make sure to add the following code as part of the member function declaration that performs file reading (or writing) operations:

**throws** java.io.IOException

Hereʻs how you would add it. Lets say your member function is called readSomeData(), you would declare your member function as follows:

void readSomeData()**throws** java.io.IOException {

}

This is needed by Java because the file operations will need you to be prepared to catch any errors as a result of performing a file operation. By issuing the *throws* command in the declaration of your member function you are essentially saying that your member function is going to pass on the responsibility to someone else rather than deal with it. When you are writing code that involves writing to a file (see next section) you are going to have to do this also. Thatʻs all we are going to say about this in ICS111. For the purposes of ICS111 you can assume that all the data files you need to read will not throw exceptions. You will learn more about exception handling in ICS211. If you want to dig deeper look here: <http://tutorials.jenkov.com/java-exception-handling/basic-try-catch-finally.html#propagating-exceptions>

# Another File Reading and Text Parsing Method (the BufferedReader)

Previously you used Scanner to parse a file while reading from a FileReader object.

Here we will introduce another type of Scanner, called a BufferedReader. We are introducing it because later you will see the BufferedReaderʻs counterpart, the BufferedWriter.

So to open a file to read using a BufferedReader you might do something like this:

BufferedReader br = new BufferedReader(new FileReader("peopledata.txt"));

This line of code declares a variable called br of type BufferedReader and assigns it an object of the same type, which takes as a single parameter, a file reader that opens a text file called peopledata.txt

So you might be asking why do we want to use a BufferedReader when the Scanner was serving us just fine? Lets postpone this for later.

In the meantime lets first understand the difference between a Buffered Reader and a File Reader. In simple terms, the File Reader is the Java library class that actually opens a file to be read. The Buffered Reader uses the File Reader to read a large chunk of text all in one gulk so that your program doesnʻt have to keep going back to the File Reader everytime it wants more text.

Now that the Buffered Reader has been created you can begin to use it to read from the text file.

So here is how you would then read a line of text from the text file:

String line;

line = br.readLine();

// Lets print it to make sure

System.out.println ("The line is " + line);

So if you were to read a single line of text from the above text file you would get:

Jason 21 M 5.11

But what if you now want to extract / **parse** each individual “piece” of information from that line. In other words what if you want to extract the name, then the age, then the sex, and the height and store it in individual variables? By the way each “piece” of the text line separated by a white space is called a “token” in computer lingo. And the process of breaking up a line of text into tokens is generally called “parsing”.

So here is some code you can use to parse a line of text into tokens, and then store their values in separate variables. Read the comments in green carefully as they explain whatʻs going on:

// First we need something to give us each token of a line of text.

// Fortunately Java has a class for doing this called StringTokenizer.

StringTokenizer st = new StringTokenizer (line);

// Here are some variables we are going to store the values from the tokens into.

String name;

int age;

String sex;

float height;

// This while loop will continue to loop if the tokenizer

// sees more tokens in the line of text.

while (st.hasMoreTokens()){

String token;

// Retrieve a token, and since it is a string, assign it to name.

token = st.nextToken();

name = token;

// The next token will be an integer so we need to convert the string

// token into an integer.

token = st.nextToken();

age = Integer.parseInt(token); // This converts the token to an integer

// The next token is again text.

token = st.nextToken();

sex = token;

// The last token is a float and therefore we need to convert the

// string token into a float.

token = st.nextToken();

height = Float.parseFloat(token); // This converts the token to a float

}

So the above while loop parses just one line of text. In order to read multiple lines of text you are going to have to wrap the whole thing into a loop sort of like this:

BufferedReader br = new BufferedReader(new FileReader("peopledata.txt"));

String line;

while ((line = br.readLine()) != null) {

StringTokenizer st = new StringTokenizer (line);

String name;

int age;

String sex;

float height;

while (st.hasMoreTokens()){

String token;

token = st.nextToken();

name = token;

token = st.nextToken();

age = Integer.parseInt(token);

token = st.nextToken();

sex = token;

token = st.nextToken();

height = Float.parseFloat(token);

}

}

br.close(); // This closes the buffered reader.

Notice that in the while loop I have:

while ((line = br.readLine()) != null)

This basically says: read a line of text, if the result is not null then continue into the while loop.

When the result *is* null then that means br.readline() had no more lines of text to read and therefore it should stop the loop.

So lets go back to addressing the question of why you would use this BufferedReader over the Scanner. The advantage of using the Scanner is that the code is simpler to use. But some programmers argue that the StringTokenizer used with the BufferedReader is more efficient. If you google around the Internet you can easily find debates on the forums over which is better. For the purposes of ICS111 we recommend you keep things simple and use the Scanner over the BufferedReader.

# Writing to a File (the BufferedWriter & FileWriter)

Of course the opposite of reading text from a file is to write text to a file.

The code you write isnʻt too different from the code for Reading from a File. Instead of using a FileReader and BufferedReader, we use something called a FileWriter and BufferedWriter.

Below we created a buffered writer object and assigned to its constructor an object of type FileWriter which will open the file “output.txt” for writing into:

BufferedWriter bw = **new** BufferedWriter(**new** FileWriter("output.txt"));

Now writing to the text file is relatively simple. Here is an example:

String myString=”This is my string”;

bw.write(myString,0,myString.length());

bw.newline();

In the line: bw.write, the first parameter is the string to write. The second parameter is the location within the string to begin reading when writing the text to the file. Zero means we begin from zeroth position (i.e. the beginning) of the string. myString.length() retrieves the length of myString. So this effectively writes the whole string to the text file.

If instead I were to say:

bw.write(myString,5,2);

The code would start on the 5th character which is “i” and end after writing 2 characters, which means it would write “is”.

Lastly the bw.newline() call adds a text newline to the file.

In the above example I am writing a string (myString) to the file. But what if I wanted to take a float variable and write it to a file. First you need to convert the float to a string before you can write it. You can do this as follows:

float f = 858.48;

String s = Float.toString(f); // This converts a float into a string

bw.write(s,0, s.length());

Likewise for Integer and Double you would do something like:

String s = Integer.toString(f); // Convert an int to a string

String s = Double.toString(f); // Convert a double to a string

# Recursion

Recursion in Computer Science means to solve a problem by taking a problem, and successively solving smaller versions of the problem.

First a non-computer science example:

Lets say a teacher walks into a class with a pile of 1000 handout sheets and wants to give every student a sheet, including himself. He could take one sheet, give a sheet to the 1st student, then walk to the 2nd student and give him/her a sheet, and keep doing this until all 1000 students are given a sheet. Solving the problem in this way is called *iteration*. Youʻve already been doing this whenever you use a loop (like a for loop, a while loop or do loop).

Of course no teacher ever does this because it would take too much work. Instead, teachers being smart (and lazy) will just take one sheet and give the pile of 999 sheets to a student and tell him/her to distribute the 999 sheets to the rest of the class. The student isnʻt stupid either. So he/she does exactly what the teacher does. He/she takes one sheet and hands the remaining 998 sheets to the next student. And this repeats until every student has a sheet. This technique is called *recursion* and it is considered an advanced computer science technique.

Recursion is used to solve many problems in Computer Science that involve searching for the best solution among a number of possible alternatives. For example, teaching a computer to play Chess basically involves teaching the computer how to search for the best move among many. If you can do this enough times you can potentially have a computer play a game of chess better than most humans, because the computer can look ahead many more moves than most average chess players. You will learn more about recursion in your future Computer Science classes but for now lets go through a simple concrete example.

In math Factorial is defined as follows:

n! = 1 \* 2 \* 3 \* .... \* (n-1) \* n

What this says in English is that the factorial of n (written as n!) is calculated as n times (n-1) (n-2) (n-3) and so forth. So for example, the factorial of 4 is calculated as: 4 x 3 x 2 x 1, which is 24.

This is what the program might look like in Java:

public class Factorial {

   public static void main(String[] args) {

       int n = 4;

       int result = 1;

       for (int i = 1; i <= n; i++) {

           result = result \* i;

       }

       System.out.println("The factorial of 4 is " + result);

   }

}

Now if we were to think about this problem in terms of **recursion**, we would describe Factorial like this:

factorial ( 0 ) = 1

factorial ( n ) = n \* factorial ( n - 1 )

Like the example in the class room and the teacher, we solve factorial by taking n and multiplying by what the factorial of n-1 is. In order to calculate the factorial of n-1 we take n-1 and multiply that by the factorial of n-1-1, and so on. We stop *recursing* when we reach zero, where the factorial of zero is defined as 1.

Here then is the equivalent code in Java:

public class Factorial {

   public static void main(String[] args) {

       int n = 4;

       int result = factorial(n);

       System.out.println("The factorial of 4 is " + result);

   }

   public static int factorial(int n) {

       if (n == 0) {

           return 1;

       } else {

           return n \* factorial(n - 1);

       }

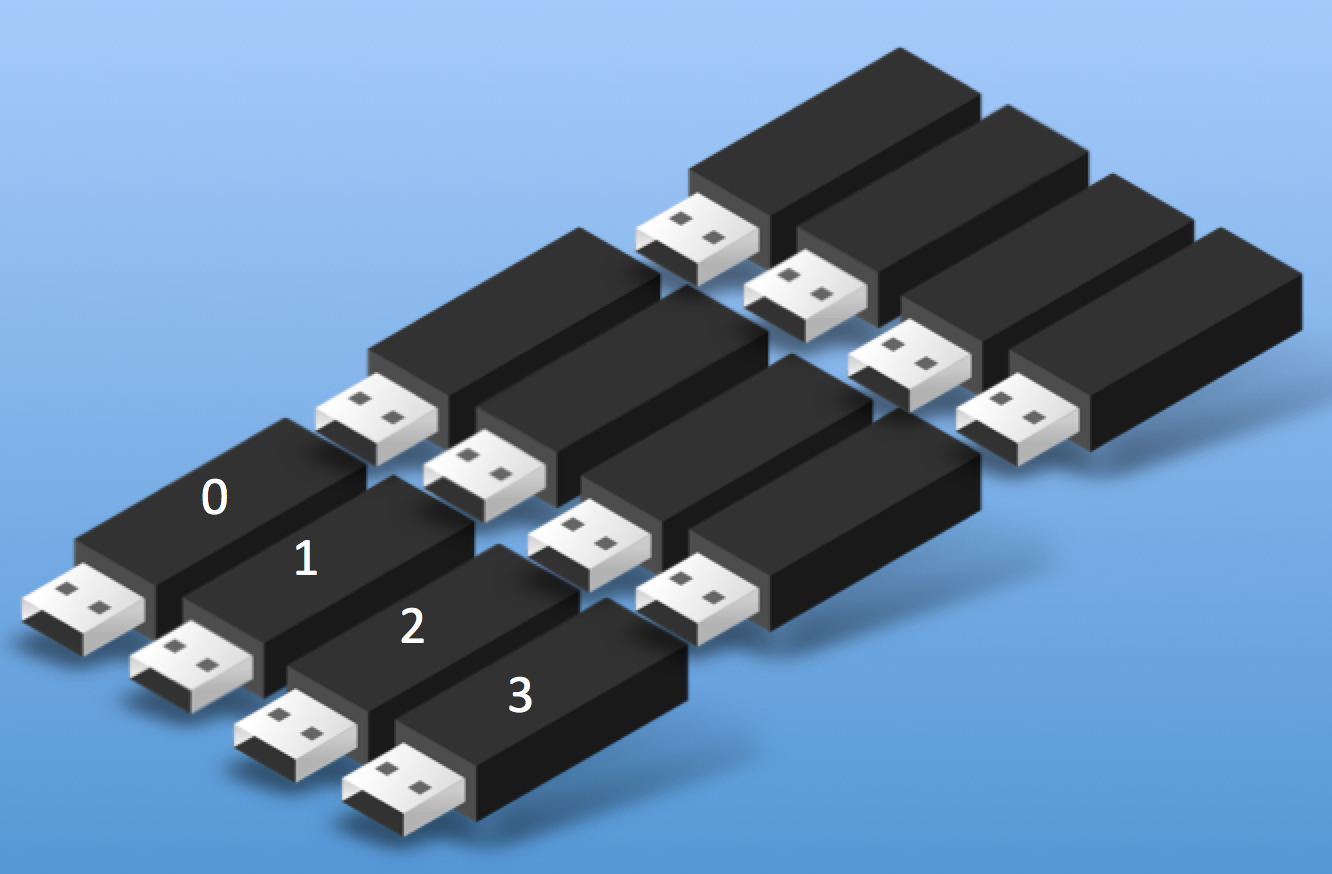
   }

}

# 2D Arrays

In section 6 you learned a little about arrays. The arrays you learned about there are called 1 dimensional arrays. 2 dimensional arrays are like having multiple 1D arrays collected together.

For example the picture below illustrates a 2D array that is 3 rows high and 4 columns across, or 4 rows high and 3 columns across.



To declare a 2D array of integers like the one pictured above you would write code like this:

int [][] myArray = new int[4][3];

To store a number into a particular entry in the array (for example the 2nd row and 3rd column) you would write code like this:

myArray[3][2] = 999;

Here for example is a loop you can write to assign 999 to every array entry:

for (int c = 0; c < 4; c++) {

for (int r = 0; r<3; r++) {

myArray[c][r]=999;

}

}

Notice we have one for loop inside another for loop. This is called a **nested loop.**

Lastly here is an example of how you might take the sum of every array entry:

int sum = 0;

for (int c = 0; c < 4; c++) {

for (int r = 0; r<3; r++) {

sum = sum + myArray[c][r];

}

}

Note: even though in the above examples we have associated the first *index* in the array with the column in the 2D array, and the second as the row in the array, you could have easily reversed it and it would be perfectly ok, as long you remember to consistently follow that convention throughout your code.

E.g.

int [][] myArray = new int[3][4];

int sum = 0;

for (int c = 0; c < 4; c++) {

for (int r = 0; r<3; r++) {

sum = sum + myArray[r][c];

}

}

# Privacy in Objects (Public / Private / Protected / Package Private)

In section 11 we talked about **scope**, which determines whether a variable or object is accessible to a program.

In Java when you declare a class you can also declare the extent to which member variables and member functions are accessible by different parts of a program. Often you will hear programmers refer to member variables and member functions as being *private* or *public* or *protected*. These are called *modifiers*. The two *modifiers* you will encounter the most are *public* and *private.*

*Public* means a member variable or member function is accessible to any part of your program.

*Private* means a member variable or member function is only accessible within the member functions of the **same** class.

When no modifier is used (sometimes referred to as package private) a member variable or member function is only accessible within the class it was defined and by other classes within the same package. A package is a collection of classes that are grouped together for distribution. For example, the Scanner class youʻve used in the past is part of the java.util package, and File reader and writer classes are part of the java.io package. Whenever you google for information on a particular Java class, the documentation will often tell you which package it belongs to. You need to make sure to **import** them in order to use them, just as with Scanner you had to write:

import java.util.Scanner;

The table below summarizes the accessibility of member variables and member functions from within a Class, Package, Subclass and the World (the rest of your program).

Modifier | Class | Package | Subclass | World

————————————+———————+—————————+——————————+———————

public | y | y | y | y

————————————+———————+—————————+——————————+——————— y: accessible

protected | y | y | y | n

————————————+———————+—————————+——————————+——————— n: not accessible

no modifier | y | y | n | n

————————————+———————+—————————+——————————+———————

private | y | n | n | n

For example, private member variables and functions are only accessible within the class it was defined. It is not accessible even if it is part of a package. It is not accessible from a subclass, and it is certainly not accessible to the rest of the program.

So why would you want to declare your member variables or member functions with these modifiers? It is generally considered good programming practice to only publicize your member variables or member functions as little as necessary to enable your program to work. This reduces the chances of another part of the code breaking things by incorrectly poking around inside your code and making use of member variables and member functions it has no business of using.

Hereʻs a real world analogy that might make things a little clearer:

When you use an ATM machine the bank lets you do several things. They let you see your balance, they let you deposit a check into your account, they let you take out cash. These can be thought of as *public* member functions. Internally however, the bank will change your bank balance depending on the above 3 operations you choose. The bank will never let you change the balance directly, otherwise you could enter 1,000,000 and become an instant millionaire. So your balance could be thought of as a private member variable. Only the bank has direct access to that, not you.

All that said, how do you actually use these modifiers? Here is an example of what a BankAccount class might look like:

public class BankAccount {

private int balance;

public void deposit(int dollars) { balance = balance + dollars; }

public int withdraw(int dollars) {

if ((balance – dollars) >= 0) {

balance = balance – dollars;

return dollars;

} else {

return balance;

}

}

public int getBalance() {

return balance;

}

}

A main program code that uses BankAccount might look like:

BankAccount myAccount;

myAccount.deposit(1000);

myAccount.deposit(1000);

System.out.println(“My balance: “ + myAccount.getBalance());

int withdrawn = 0;

withdrawn = myAccount.withdraw(1500);

System.out.println(“Withdrawn: “ + withdrawn);

System.out.println(“My balance: “ + myAccount.getBalance());

withdrawn = myAccount.withdraw(999);

System.out.println(“Withdrawn: “ + withdrawn);

System.out.println(“My balance: “ + myAccount.getBalance());

System.out.println(myAccount.balance);

myAccount.balance = 10000000;

The last two lines of code are illegal because balance is a private member variable, which is only accessible to the BankAccount class and no body else.

The program itself (without the last two lines of code) would print:

My balance: 2000

Withdrawn: 1500

My balance: 500

Withdrawn: 500

My balance: 500

# ArrayList

One of the disadvantages of arrays is that you have to declare their size ahead of time and you canʻt change its size later. For example, what if you created an array of 100 items and decide when the program is running that you need a bigger array of 200 items. Using traditional arrays youʻre stuck. There is nothing you can do except to stop the program, change the code to make the array bigger, and restart it.

The smarter approach is to use a special kind of Array that can automatically grow in size while the program is running, hence ArrayLists. ArrayLists are like arrays but with no fixed size. ArrayLists are created with an initial size. When this size is exceeded, it is automatically enlarged. When items are removed, the array will automatically shrink.

The most salient member functions of ArrayList are:

add(Object o) add an object to the ArrayList

get(int index) get the object at a specific index (location in the array list)

set(int index, Object o) replaces the object at a specific index with a specific object

remove(int index) remove an object at an index

size() determine the size of the ArrayList

clear() clear the ArrayList of all objects

There are many more member functions and you can check them out here:

<http://www.tutorialspoint.com/java/java_arraylist_class.htm>

Here is an example of using the most salient member functions to store an array of integers. Even though this example uses integers one can store just about any object in an Arraylist.

**import** java.util.\*; // ArrayLists are part of java.util package.

// create an array list

ArrayList<Integer> al = **new** ArrayList<Integer>();

System.***out***.println("Initial size of al: " + al.size());

// add elements to the array list

al.add(**new** Integer(5));

al.add(**new** Integer(10));

al.add(**new** Integer(2));

al.add(1, **new** Integer(999));

System.***out***.println("Size of al after additions: " + al.size());

System.out.println("Item at index 1: " + al.get(1));

// display the array list

System.***out***.println("Contents of al: " + al);

// Remove elements from the array list

al.remove(**new** Integer(999));

al.remove(2);

System.***out***.println("Size of al after deletions: " + al.size());

System.***out***.println("Contents of al: " + al);

This code would produce the following output:

Initial size of al: 0

Size of al after additions: 4

Item at index 1: 999

Contents of al: [5, 999, 10, 2]

Size of al after deletion: 2

Contents of al: [5, 10]

There are a couple things to be aware off in the code above.

Firstly look at how the ArrayList is declared:

ArrayList<Integer> al = **new** ArrayList<Integer>();

Where it says ArrayList<Integer> it is telling Java that the variable al will be declared with an ArrayList of type Integer. Note: this Integer is NOT the same as the int primitive youʻve come across so far. int is a primitive in Java. But Integer is a class in Java. Therefore you can use Integer to create **objects** of type Integer. And so ArrayList <Integer> is expecting you to place **objects** of type Integer into the ArrayList, not primitive values of int.

Here's another example. Lets say you wanted to make an ArrayList to store floating point numbers. float is a Java primitive. But Float is a Java **class** for floating point numbers. So in order to create an ArrayList of floating point numbers you need to do something like:

ArrayList<Float> al = new ArrayList<Float>();

You cannot say:

ArrayList<float> al = new ArrayList<float>();

because float is not a Java class.

So lets say you wanted to make an ArrayList of Strings. Since String is a proper class in Java you can declare an ArrayList of Strings as follows:

ArrayList<String> al = new ArrayList<String>();

Ok now look at the code that says:

al.add(**new** Integer(5));

This simply adds an object of type Integer with the value 5 to the array list (called al).

Now look at this line of code:

al.add(1, **new** Integer(999));

This adds an object of type Integer with the value 999 to location 1 of the array list. It is attempting to insert 999 into the array list and as a result push items already in the array list further down the list.

So while this code:

al.add(**new** Integer(5));

al.add(**new** Integer(10));

al.add(**new** Integer(2));

produces an array list with the following items: 5 10 2

The addition of this code:

al.add(1, **new** Integer(999));

changes the array list so that the items are now: 5 999 10 2

999 has been **inserted** into position 1 of the array list.

Lastly, since ArrayList are part of the package java.util, make sure to add this at the top of your program. It tells Java to include the java.util package when attempting to compile and run the code.

**import** java.util.\*; // ArrayLists are part of java.util package.

Ok now that you understand ArrayList I need to give you a warning. Sometimes you will find ArrayList code without the “<” “>”.

Like you might see in the following code:

ArrayList al = new ArrayList();

// add elements to the array list

al.add(5);

al.add(10);

al.add(15);

// Retrieve the item at index 1

System.out.println(“Item at index 1: “ + al.get(1));

// Change the item at index 1 to the value 111

al.set(1, 111);

// Remove item 2 from the array list

al.remove(2);

In this version of the code the omission of “<” and “>” results in Java attempting to guess at what type ArrayList you want it to automatically create.

For example the code:

al.add(5);

al.add(10);

al.add(15);

seems to imply that the items/objects that will be added to the ArrayList are integers. So behind the scenes (hidden from you), the Java compiler will convert the code that looks like this:

ArrayList al = new ArrayList();

// add elements to the array list

al.add(5);

al.add(10);

al.add(15);

into code that looks like this:

ArrayList<Integer> al = **new** ArrayList<Integer>();

// add elements to the array list

al.add(**new** Integer(5));

al.add(**new** Integer(10));

al.add(**new** Integer(2));

In general, to avoid confusion it is recommended that you use the form of coding style with the “<” and “>”. These marks are Java’s way of creating what are called **templates** or **generic types.** Generic types are beyond the scope of ICS 111. You will likely encounter them in a more advanced programming class whenever you are writing program codes that can *generically* work with any class, rather than with only one specific class (hence the name, **generic** **types**). You will also encounter this in C++ (which is where Java originally borrowed the concept from).

# Inheritance

Inheritance is a means of extending existing Java classes to make them more specialized or powerful. It is also a means to reduce the amount of code that may end up having to be written redundantly.

For example lets say you are writing a video game involving farm animals, and in the video game you had to simulate the behavior of these animals. For example, they can sleep, walk, eat, and poop.

So lets say you want to make a class to represent a sheep. You could perhaps write something like this:

class Sheep {

void sleep() {

System.out.println(“zzz”);

}

void walk() {

System.out.println(“patter patter patter”);

}

void eat() {

System.out.println(“chomp chomp”);

}

void poop() {

System.out.println(“splat”);

}

}

Now lets say you wanted to write a class for a farm dog. You could perhaps write something like this:

class Dog {

void sleep() {

System.out.println(“zzz”);

}

void walk() {

System.out.println(“patter patter patter”);

}

void eat() {

System.out.println(“chomp chomp”);

}

void poop() {

System.out.println(“splat”);

}

}

Lastly, lets say you wanted to write a class for a chicken. You could perhaps do this:

class Chicken {

void sleep() {

System.out.println(“zzz”);

}

void walk() {

System.out.println(“patter patter patter”);

}

void eat() {

System.out.println(“chomp chomp”);

}

void poop() {

System.out.println(“splat”);

}

}

I hope you are starting to see a tedious pattern forming. While the code above is perfectly ok in that it will run just fine, much of the code is repeated because all dogs, sheep, and chicken sleep, walk, eat and poop. It would save us a lot of work coding if there was a way for our dog, sheep and chicken to just share those member functions. In fact there is. The way to do that is to use **Inheritance**. To use inheritance you first declare a class (say call it Animal) and then have subclasses (like Dog, Sheep and Chicken) inherit the properties of the Animal superclass. So Animal is the parent class (superclass), and Dog, Sheep and Chicken are child classes (subclass).

Since sheep, dogs and chicken are all animals, and all animals walk, sleep, eat and poop.

We can create a superclass called Animal like this:

class Animal {

void sleep() {

System.out.println(“zzz”);

}

void walk() {

System.out.println(“patter patter patter”);

}

void eat() {

System.out.println(“chomp chomp”);

}

void poop() {

System.out.println(“splat”);

}

}

Now when it comes time to define the classes for Sheep, Chicken and Dog you can simply **Inherit** the member functions from Animal just by doing this:

class Dog **extends** Animal {

}

class Chicken **extends** Animal {

}

class Sheep **extends** Animal {

}

The **extends** keyword is Javaʻs way of letting you make a class by extending from an existing class. In this case, the subclass Chicken is an extension of the superclass Animal. That means that when you create a Chicken object, it automatically acquires the member functions Sleep, Walk, Eat and Poop without having to explicitly define them. Note: itʻs not just member functions that are acquired, member variables are also inherited (with some caveats- see later).

But what if you wanted to say that Chickens walk differently from regular animals. For example, instead of “patter patter patter” you want a Chicken to walk like “tipi toe”. To do that you simply add a walk member function that will **override** the walk member function that you inherited from the Animal class. You can do that as follows:

class Chicken extends Animal {

void Walk() {

System.out.println(“tipi toe”);

}

}

Now what if you wanted to also let a Chicken fly by introducing a Fly() member function? Since sheep and dogs canʻt fly you wouldnʻt put that member function in the Animal class otherwise Sheep and Dog would inherit that ability too. Instead you put that member function ONLY in the Chicken class, like this:

class Chicken extends Animal {

void Walk() {

System.out.println(“tipi toe”);

}

void Fly() {

System.out.println(“Flap flap”);

}

}

So now you have a Chicken class which lets you create Chicken objects which are able to Sleep, Eat and Poop like regular animals, but walks differently from other animals, and can also fly. In other words we have been able to extend the capabilities of the original superclass by adding new member functions that are specific only to the Chicken. Remember what I said in the first paragraph of this section? **Inheritance is a way to extend the capabilities of classes and to reduce the amount of redundant code that needs to be written.**

One thing to keep in mind is that while member variables and member functions are inherited from a superclass to a subclass, the Constructor is NOT inherited. If for some reason you want to call the superclassʻs constructor, you would have to do so by calling: super();

Another thing to keep in mind is that member variables that are marked **private** are also NOT inherited (to review what a Private member variable is see Section 17.) Instead if you wish for member variables to be inherited then declare them either as public or protected.

# Polymorphism

Now that you have created a Sheep, Dog, and Chicken class that inherits some of its properties from an Animal superclass, lets say you want to make 4 chickens, 4 sheep and 4 dogs. Then after making those animals you want them all to Walk in an infinite loop.

One way to do it is like this:

Dog [] myDogs = new Dog[4];

for (int i = 0; i < 4; i++) {

myDogs[i] = new Dog();

}

Sheep [] mySheep = new Sheep[4];

for (int i = 0; i < 4; i++) {

mySheep[i] = new Sheep();

}

Chicken[] myChickens = new Chickens[4];

for (int i = 0; i < 4; i++) {

myChickens[i] = new Chickens();

}

while (true) {

for (int i = 0; i < 4; i++) {

myDogs[i].walk();

}

for (int i = 0; i < 4; i++) {

mySheep[i].walk();

}

for (int i = 0; i < 4; i++) {

myChickens[i].walk();

}

}

A smarter way would be to do something like this:

Animal [] myAnimals = new Animal[12];

for (int i = 0; i < 4; i++) {

myAnimals[i] = new Dog();

myAnimals[i+4] = new Sheep();

myAnimals[i+8] = new Chicken();

}

while (true) {

for (int i = 0; i < 12; i++) {

myAnimals[i].walk();

}

}

Take some time to look over the differences between the first and second approach. Right away you should notice the first approach is a lot lengthier than the second. You are essentially creating three arrays, one for Dog, one for Sheep and one for Chicken. In the second approach you are instead making just an array of Animal that is large enough to accommodate 4 sheep, 4 dogs, and 4 chickens- hence: Animal [] myAnimals = new Animal[12];

So in the first for loop:

for (int i = 0; i < 4; i++) {

myAnimals[i] = new Dog();

myAnimals[i+4] = new Sheep();

myAnimals[i+8] = new Chicken();

}

You are essentially filling the myAnimals array so that the first third is filled with dogs, the second third is filled with Sheep, and the last third is filled with Chickens.

With that in place the next body of code simply goes through the 12 elements of the array and calls the Walk function in each item stored in the array.

while (true) {

for (int i = 0; i < 12; i++) {

myAnimals[i].walk();

}

}

The printout of such a program would read:

patter patter patter

patter patter patter

patter patter patter

patter patter patter

patter patter patter

patter patter patter

patter patter patter

patter patter patter

tipi toe

tipi toe

tipi toe

tipi toe

Notice that as each myAnimals[i].walk() is performed Java cleverly knows that the objects are actually **extensions** of the Animal class. So when it gets around to calling Walk for the last 4 animals (which are Chicken), instead of printing “patter patter patter”, it prints “tipi toe”. Even though the myAnimals array is an array of type Animal, Java somehow knew that it wasnʻt dealing with objects purely of the class Animal, but instead was dealing with objects of type Dog, Sheep and Chicken, which are all children of Animal. This behavior is called **Polymorphism.** In essence myAnimal can temporarily become the type of one of its subclasses, like shapeshifters in science fiction movies can assume the form of other people. Thatʻs essentially what Polymorphism means- **poly** means many; and **morphism** means forms- many forms.

Polymorphism is the corner stone of object-**oriented** programming. Without polymorphism Java would not truly be an object-oriented programming language. Instead it would only be an object-**based** language. You will learn more about Polymorphism in ICS211 and beyond. This introduction is just a taste of it.

# Try/Catch/Finally – Error/Exception Handling

When a user makes a mistake using an application, the application shouldnʻt just crash. Instead the programmer has to take the responsibility of making sure that any errors that might occur are caught and dealt with so that the program can continue to operate (if possible). This is called **defensive programming.** The connotation isnʻt too different from the term defensive driving.

Java provides a mechanism for error handling using the **try**, **catch** and **finally** statements. In Java another word often used synonymously with error is **exception**. And when an error occurs, usually it is phrased as: “an **exception** has been **thrown**.”

The syntax for try/catch/finally is as follows:

try {

// code that might throw an exception goes in here…

}

catch (exception-type *identifier*) {

//code that gets executed when an exception occurs …

}

finally {

// code that always gets executed regardless of whether an exception has occurred or not …

}

Often you will see try/catch used without finally.

Here is a quick and simple example of using try/catch

int a = 5;

int b = 0; // you know this won’t work

try

{

int c = a / b; // but you try it anyway

}

catch (ArithmeticException e)

{

System.out.println("Can't do that! Here’s the exception: "+e);

}

This program throws a divide-by-zero exception, which is an Arithmetic Exception, which is why we used ArithmeticException to catch the exception. When that exception occurs the program prints the message: “Canʻt do that! Hereʻs the exception: java.lang.ArithmeticException: / by zero“

If you had some code that might throw multiple different types of exception you could write it like in the following example:

try

{

// statements that might throw

// FileNotFoundException

// or IOException

}

catch (FileNotFoundException **|** IOException e)

{

System.out.println(e.getMessage());

}

The above example will catch two classes of exceptions whenever a file operation is performed: a FileNotFoundException- which is thrown whenever there is an attempt to open a file that does not exist; and an IOException- which is thrown whenever there is any other type of file related exception- like if you tried to read beyond the end of the file (EOFException).

So for example if you wanted to specifically deal with the EOFException you could do something like this:

catch (FileNotFoundException | IOException e)

{

System.out.println(e.getMessage());

if (e == EOFException) {

System.out.println(“You are at the end of the file!”);

}

}

Keep in mind that variables that might be declared in the try block are not visible in the catch block. I.e. they follow Javaʻs standard scoping rules. For a review of scoping rules see Section 11.

By the way, if you want to just ignore the exceptions altogether you could do this:

try

{

// Statements that might throw

// FileNotFoundException

}

catch (FileNotFoundException e){

}

This is called “swallowing the exception” and is generally considered bad practice because it means you are making the exception go away and assume everything is ok. This can result in unpredictable behavior in your program.

Finally, what about the **finally** statement?

The finally block is optional and is code that you want to always be executed after a try/catch regardless of whether an error occurred. Finally is usually a good place to put whatʻs called **cleanup code**. For example, lets say you opened a file at the beginning of your code block, and in your try block an error occurred which got caught by the catch block. Your code will then enter the finally block where you could do things like closing the file properly. And even if an error did not occur your program would still enter the finally block to close the file properly. Closing a file after opening is considered good programming practice because when a file is left opened it can potentially prevent other programs from gaining access to them.

# The End?

Believe or not that covers all the most fundamental aspects of the Java language. In fact since Java is so similar to other languages like C# and C++, many of the things you learned will carry over.

You also now have the necessary knowledge to use Javaʻs more advanced libraries. If you plan to be a long term Java programmer you should hang on to this URL as it contains the main reference documentation for Javaʻs enormous library of classes:

<http://docs.oracle.com/javase/7/docs/api/>

The EZ graphics library that you used for this class is also built from these libraries.

Now that you know the basics of Java there is only one thing missing- experience. Just as you canʻt become a Michael Jordan by learning to play basketball from reading a book, you canʻt become a good Java programmer by reading this document and just listening to lectures in class. You gotta get out there and practice, practice, practice. But once you are fairly fluent in one language it becomes much easier to learn a new programming language- much more so than learning a different foreign language- like French, Chinese or Samoan!