CSCI 1320 Java Commands

Hello, World!

```java
// imports go here
public class Hello {

   public static void main(String [] args) {
      System.out.println("Hello, World!"); // yay!
   }

   /*
   additional methods go here
   ... after main() but inside the class
   ... but not inside a comment!
   */
}
```

Link to Code

Expected output (on screen):

```
Hello, World!
```

It is a time-honored tradition in computer science to begin any new endeavor by creating the simplest possible working example. When writing a program that often involves getting the computer to display something. So, this is a Java "Hello, World" program. The idea is that if you can get this program to compile and run on your computer then you have all the pieces you need to begin writing more complex progams. Some things to note:

- Comments are used to provide notes/annotation to be read by other programmers. Comments are ignored by the compiler. To create a multi-line comment use `/*` at the beginning and `*/` at the end. A single line comment starts with `//` and goes to the end of the line.
- By convention each Java source file contains a single class that has the same name as the file. So, this program would be saved in a text file named `Hello.java`.
- When you run a Java program, it always starts executing statements in the method called `main()`.
- You can have other methods besides `main()`. Those will appear *after* `main()` (not inside of it) and *inside* the class (not after it).
- If you need to import any classes, the `import` statements go before the class (but not inside of comments).

Declaring Variables

```java
float x;
double y,z;
int ans;
char let;
String name;
boolean isFun;
```

x	?	ans	?
y	?	let	?
z	?	name	?
isFun	?		

Declaring a variable specifies its type and name and reserves a location in the computer's memory to hold the variable. A variable whose type is `float` or `double` is used to hold numbers with decimal points. Variables whose type is `int` holds numbers without decimal places. A `char` variable holds a single character which can be a letter, symbol or a digit. A `String` variable holds a series of individual characters. A `boolean` variable hold the value `true` or `false`. Reserving space for a variable does not specify what value is to be held in the variable.

Initializing Variables

```
y= 0.1;
z= 7.5 + y;
name= "Fred"; // String use double quotes
let= 'R';      // chars use single quotes
a= (int) x;
a= a + 1;
isFun= true;
```

y	0.1	name	"Fred"
z	7.6	isFun	true
a	8		

Initializing a variable means "giving it a value for the first time." It is possible to both declare and initialize a variable on the same line, though there are times you want those actions to be separated. Initializing a variable is accomplish through use of the *assignment operator* (i.e., =). The assignment operator takes a value (on the right) and places it into the variable (on the left). The statement `a= (int) x;` is an example of type-casting (temporarily changing the type of an expression). When we *type-cast* a `double` to be an `int` the result is that the decimal places get chopped off. The statement `a= a + 1;` highlights the different between the assignment operator and the mathematical concept of "equals". In this case the variable `a` is incremented by 1.

Displaying Values

```
System.out.println("Hello");
System.out.print("Answer is: ");
System.out.println(a);
System.out.println("Name is: "+name);
```

Output of statements assuming variables have values from previous examples:

```
Hello
Answer is: 8
Name is: Fred
```

The `println` (pronounced "print line") and `print` commands are used to display results to the console/screen. Both commands require you provide a string in the parentheses and both will display the provided string. After displaying the value `println` performs an extra step of displaying a newline character (so future output will appear on the following line).

Getting Input

```
import java.util.Scanner; // at top of file
. . .
Scanner kb;
kb= new Scanner(System.in);
int a;
double x;
String str1,str2;

System.out.print("Enter age: ");
a= kb.nextInt();
kb.nextLine(); // toss out extra newline

System.out.print("Enter num: ");
x= kb.nextDouble();
kb.nextLine(); // toss out extra newline

System.out.print("Enter word: ");
str1= kb.next();
kb.nextLine(); // toss out extra newline

System.out.print("Enter sentence: ");
str2= kb.nextLine();
// .nextLine() tosses out extra newline
```

a	?
x	?
str1	?
str2	?

```
Enter age: 7
Enter num: 2.5
Enter word: hi
Enter sentence: How are you?
```

a	7
x	2.5
str1	"hi"
str2	"How are you?"

The `Scanner` class can be used to obtain input from various sources. In this example we are reading from keyboard (`System.in`). The commands `.nextInt()`, `.nextDouble()`, and `.next()` all read the next available token and stop reading at any trailing whitespace characters; the read value is then converted to the specified type (int, double, String, respectively). To get the data into the input buffer to be read the user must type the data and then press then Enter key (which generates a newline character … which is classified as whitespace). So, `kb.nextInt()` reads the number, but stops at the newline character. To prevent possible issue later it is good form to toss out the extra character. The `.nextLine()` command reads more than a token; it reads an entire line and tosses out the trailing newline character.

Integer Arithmetic

```java
int a,b,c,d,e;
a= 7;
b= -1;
c= a + b * 3; // 4
d= b - a / 3; // -3
e= a % 3;     // 1
```

a	7
b	-1
c	4
d	-3
e	1

Integer arithmetic in Java is fairly simple. You can use *arithmetic operators* ($-$, $+$, $*$, $/$) to perform addition, subtraction, multiplication, and division, respectively. The order of operations works as in algebra. NOTE: Dividing integers has a wrinkle in that *in Java, an* `int` *divided by an* `int` *is always an* `int`. In the example, $a/3$ results in 2 (because 3 goes into 7 two times with a reminder of 1). The *modulus operator* can be used to extract the remainder portion of integer division: `a % 3` produces 1 (the remainder when 7 is divided by 3). Parentheses can be used to override order of operations.

Floating Point Arithmetic

```java
double a,b,c,d;
a= 4.0;
b= -2.2;
c= a + b * 3;          // -2.6
d= Math.sqrt(a) + b; // -0.2
```

a	4.0
b	-2.2
c	-2.6
d	-0.2

Floating point arithmetic works like integer arithmetic except that division does not produce an integer. In addition there are a host of functions provided in the `Math` library to provide many useful result. This example shows how to find the square root of a number. For a complete list of `Math` functions see: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html.

Strings

```java
int num1,num2;
String one, two, three;
one= "Amy";
two= "Hi, there!";
if (one == "Amy") { // nope!
   System.out.println("Greetings!");
}
if (one.equals("Amy")) { // yep!
   System.out.println("Hi, Amy!");
}
num1= one.length();
System.out.println(one.charAt(2));
num2= one.indexOf("the");
System.out.println(one.toUpperCase());
three= two.substring(4,7)+"n";
```

num1	3
num2	4
one	"Amy"
two	"Hi, there!"
three	"then"

Expected output:

```
Hi, Amy!
y
FRED
```

Variables of type `String` are actually complex objects that require special treatment. They also come with quite a few built-in methods for manipulating them. To compare strings do not use `==` because you will always get a false result (because it compares their addresses). Instead use the `.equals()` method as illustrated. For explanations of the methods used in this example and for a complete list of methods available for `String` variables see: https://docs.oracle.com/javase/8/docs/api/java/lang/String.html. NOTE: When the `+` operator is applied to strings is performs *string concatenation*, which is the fancy computer science terms that means "stick them together".

If Statements

```java
if (a < 0) {
   // do this if true
}
else {
   // do this if false
}
```

The *if-statement* allows you to select one of two alternate actions based on a comparison. After the word "if" you must have, in parentheses, a boolean expression which evaluates to either true of false. If the expression evaluates to true the statements in the first section will happen, otherwise the statements in the `else` part will happen. Keep in mind the statements in either section could be if-statements themselves (which is called "nested ifs"). The `else` portion of the statement is optional. If else is omitted and the boolean expression is false then the statements are simply skipped.

Boolean Expressions

```java
int a,b;
boolean c,d,e,f,g;
a= 7;
b= -1;
c= true;
d= a<b;
e= a<b || a>0;
f= a<=b && a!=0;
g= !isCool && a==7;
```

a	7
b	-1
c	true
d	false
e	true
f	false
g	true

A *boolean expression* is an expression that evaluates to either true or false. based on a comparison. They can be built using a combination of variables, relational operators, boolean operators, and constants. *Relational operators* ($<$, $<=$, $>$, $>=$, $==$, $!=$) compare values of the same type and produce a boolean result. NOTE: $==$ compares values for equality and is different than $=$ (which is the assignment operator). Also, $!=$ means "not equal". *Boolean operators* (`&&`, `||`, `!`) act on boolean values and produce a boolean result. NOTE: `&&` is read as "and" and is true only if both parts are true. The boolean operator `||` is read as "or" and is false only if both parts are false. The operator `!` is read as "not" and when placed in front of a boolean expression toggles its value. The order in which boolean expressions are evaluated is 1) `!`, 2) `<`, `<=`, `>`, `>=`, 3) `==`, `!=`, 4) `&&`, 5) `||`. Of course, parentheses override everything so used them if unsure.

While Loops

```java
while (a < 0) {
   // loop body
}

do {
   // loop body
} while (a < 0);
```

A *while loop* is used to repeat a group of actions and begins with the keyword `while` followed by a boolean expression in parentheses. The Java statements between the {}'s are run if the boolean expression evaluates to true. When the statements have been executed the boolean expression is revaluated to determine whether or not the statements should be performed. If the expression evaluates to false the statements are skipped. For this reason the boolean expression must contain a variable that gets changed by the statements in the loop (or else suffer the fate of a so-called "infinite loop"). The *do-while loop* works in exactly the same way as a while loop except that the loop body is performed once no matter what and the evaluation of the boolean expression happens at the end of the loop. The keyword `do` marks the beginning of the section to be repeated in the case the boolean expression evaluates to true.

For Loops

```java
int i;
for (i=0; i<5; i=i+1) {
   System.out.print(""+i+",");
}
```

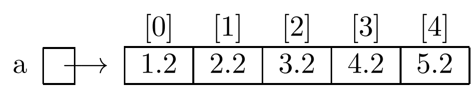
Expected output:

```
0,1,2,3,4,
```

A *for loop* is a specialize version of a while loop that begins with the word `for` and contains three statements in parentheses: 1) a statement to be executed once before the loop begins, 2) a boolean expression which when true causes the loop body to be run (just like a while loop), and 3) an "update" statement to be run each time the loop body completes. Normally, for-loops are used in situations where the number of times the loop should happen is known prior to the start of the loop. The example above would execute the statements in the loop body 10 times.

Arrays

```java
1 int i;
2 double [] a;      // declare array name
3 a= new double[5]; // reserve 10 doubles
4 a[0]= 1.2;
5 for (i=1; i<5; i++) {
6    a[i]= a[0]+i;
7 }
```



An *array* is a "big" variable that combines multiple variables of the same type under one name. The individual variables are distinguished from one another by their position in the array (starting with 0). Arrays are often used in conjunction with for-loops. In the example above the variable `a` refers to 5 double variables whose names are `a[0]`, `a[1]`, ..., `a[4]`. NOTE: The variable `a[0]` is a variable of type double and can be used like any other variable of type double. To act on all values in an array you must do it one variable at a time. The variable `a` (created on line 2) is separate from the block of 5 variables (created on line 3). The variable `a` is a *reference* (i.e., contains an address/arrow) to the array block.

Methods

```java
public static void showSum(double a, double b) {
   double c;
   c= a + b;
   System.out.println("Sum is: "+c);
}

public static double calcSum(double a, double b) {
   double c;
   c= a + b;
   return c;
}

// To call these methods:
double x= 7.5;
showSum(x,1.0);
showSum(x*2.0,x-2.0);
x= calcSum(1.0,2.2); // x= 3.2
System.out.println(x);
```

Expected output:

```
Sum is: 8.5
Sum is: 20.5
3.2
```

Writing a method allows you to give a name to a section of code. The section of code can then be invoked at any time by simply using the name of the method. It is useful to name sections of code to organize a large program or for sections that will need to be invoked multiple times. To define a method start with keywords `public static` followed by the return type followed by the name of the method followed by the formal parameter list. Names of methods should be verbs that describe the action of the method. The *formal parameter list* declares variables that must be filled in order for the method to do it job. In this example the methods specify two formal parameters of type double named `a` and `b`. The methods must be *called* in order for them to run. This example shows two calls to `showSum` and one call to `calcSum`. When calling a method you must specify the *actual parameters* that will be used to fill in the formal parameters for that particular call. So, in the first call the formal parameter `a` takes on the value 7.5 and the formal parameter `b` takes on the value 1.0. The `calcSum` method demonstrates a method that returns a value. The return type is specified prior to the method name. In method that returns a value you must 1) have a return statement to indicate the value to the returned, and 2) do something with the returned value. In the example, the local variable `c` is returned and the returned value is stored into the variable `x`.

Formatted Output

```java
double x= 7.5;
String str= "Fred";
int a= 2;

System.out.printf("%s is %d\n",str,a);
System.out.printf("A%5d %4.2fB\n",a,x);
System.out.printf("A%-5d %4.2fB\n",a,x);
```

Expected output:

```
Fred is 2
A    2 7.50B
A2    7.50B
```

It can be useful to to produce formatted output using `.printf()`. The `printf()` method's first parameter is a format string that contains placeholders: `%s` for strings, `%d` for ints, and `%f` for doubles/floats. The remaining parameters are values that will be plugged in to the placeholders. A number number right after the `%` in a format string is called a *field width specifier* and is used to specify how much space to leave for the value. In this example, `%4.2f` leaves 4 spaces for a floating point number that will display 2 decimal places. Placeholders with a field width specifier right justify the value by default. To left justify place a value put a negative sign in front of the number.

Reading Text Files: Scanner

```java
  // requires some imports:
import java.util.Scanner;
import java.io.FileNotFoundException;
import java.io.FileInputStream;

public static double sumFile(String filename) throws FileNotFoundException {
  Scanner numFile;
  double num,total=0.0;

  numFile= new Scanner(new FileInputStream(filename));
  while (numFile.hasNextLine()) {
    num= numFile.nextDouble();
    total= total + num;
    numFile.nextLine(); // toss out \n
  }
  numFile.close();
  return total;
}
```

Suppose datafile looks like this:

```
1.5
10.2
2.3
5.2
```

Expected return value: 19.2

The `Scanner` class can be used to read data from files in addition to reading values from a keyboard. When opening the file simply indicate the filename as a `FileInputStream` object rather than using `System.in`. The `Scanner` method `.hasNextLine()` will return if there is more to be read and false otherwise. Then proceed to read from the file just as if you reading from the keyboard. Be sure to close the file when you are finished. Java requires that we announce how we are going to handle the situation if the file we try to open is not there. The statement `throws FileNotFoundException` tells Java that we intend to pass the error along to whatever method happens to call this one. This method accepts the name that presumably contains one number per line of file. We then read the numbers one at a time and add them up. The sum is returned.

Reading Text Files: BufferedReader

```java
  // requires some imports:
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public static double sumFile(String filename) throws IOException {
  BufferedReader numFile;
  double total=0.0;
  String str;

  numFile= new BufferedReader(new FileReader(filename));
  while ((str= numFile.readLine())!=null) {
    total= total + Double.parseDouble(str);
  }
  numFile.close();
  return total;
}
```

Suppose datafile looks like this:

```
1.5
10.2
2.3
5.2
```

Expected return value: 19.2

The BufferedReader class can be used to read data from files. It runs much faster than using Scanner but has the disadvantage that it can only read a single line at a time as a string. So, lines from the file may need to be broken apart and converted to various types. The open command can generate an IOException which must be handled. This example accepts the name of a text file that presumably contains one number per line of the file. We then read one line at a time and put it into the string **str**. The value comes back a **null** we know the read failed and it is time to stop reading. We convert the string to a double value and then add it to our running total.

Saving Text Files

```java
  // requires some imports:
import java.io.FileNotFoundException;
import java.io.PrintStream;

public static void saveData(String filename) throws FileNotFoundException
{
  PrintStream f= new PrintStream(filename);
  double y= 3.8234;

  f.print("This is so fun!");
  f.print("How are you?\n");
  f.println("What do we do now?");
  f.printf("I am am %8.1f feet tall\n",y);
  f.close();
}
```

Expected output (in junk.txt):

```
This is so fun!How are you?
What do we do now?
I am    3.8 feet tall
```

To write to a text file use the PrintStream class which is the same type of object as System.out. So, you can use .print(), .println(), and .printf() to write to a file just as if you were displaying it to the screen. When you open the file by creating the PrintStream object it will create a new file if it doesn't exist; if the file did exist it is destroyed!

Try-Catch

```java

int num;
Scanner kb= new Scanner(System.in);

try {
  System.out.print("Enter number: ");
  num= kb.nextInt();
  System.out.println("You entered: "+num);
}
catch (InputMismatchException e) {
  System.out.println("Must enter an integer");
  System.out.println(e);
}
```

Expected output if user enters 7.

```
Enter number: 7
You entered: 7
```

Expected output if user enters "frog".

```
Enter number: frog
Must be an integer
java.util.InputMismatchException
```

A *try-catch block* can be used to prevent a program from crashing on an error condition. In this example, if the user types a double or String instead of an int the program would normally crash with an InputMismatchException. With the try-catch block we ask the program to try the statements listed. Then we catch exceptions that occur using one or more catch blocks. If the listed exception occurs control is transferred to the catch block rather than crashing. NOTE: This is not a looping structure! If you want your program to go back and let the user try entering the number again then you need to put the try-catch block inside a loop. NOTE: try-catch can be used to handle code that works with files so you no longer need to announce throws FileNotFoundException in the header.