Exploring Java

Gabe Byk & Tyler Slussar

Programming Languages

April 26, 2022

1. Introduction

Java is a compiled language that uses Just-in-Time compiling to compile its code to machine code. Its syntax is fairly similar to C++, so anyone who knows C++ can probably read Java code and figure out more or less what it's meant to do. The primary IDE for Java is Eclipse, which helps simplify file management and creating projects and classes correctly. To program in Java, you'll need to download and install both the Java Development Kit (JDK) and Eclipse. This essay will discuss Java's variables, data types, expressions, control structures, subprograms, classes, and exception handling. After we discuss Java, we will discuss the program we decided to make to demonstrate Java's various features, evaluate Java as a language, and finally reflect on our experiences.

2. Variables

Java defines four types of variables. Those are instance variables, class variables, local variables, and parameters. Inside a class, if a variable is not declared static then it is an instance variable. Instance variables are instances within a class. They are generated and assigned when a new class object is assigned. Every class has their own set of instance variables. Inside a class, if a variable is declared static then it is a class variable. A class variable is generated and assigned when a new class object is assigned. Every class shares the class variable. Therefore, if one instance of the class changes the class variable then all other instances of the class will have that changed class variable. This is usually an issue. The class variable should not change and remain constant throughout the class. Java does use the keyword *final* which means a class variable cannot change. This is useful as then a written method will receive an error if it is trying to change the class variable.

Local variables are stored temporarily. These variables are assigned by their type and their variable name. The scope of the variable is in between curly braces ({}). Once a method, class, or function is completed the variable that was initialized inside will no longer be available to anything on the outside of that function, class, or method. This remains true for any form of a loop. In a for loop if there is a variable initialized such as *i* in *for (int i; i < 3; i + +)*, i will only be accessible in the loop itself and changing it in the loop will change the counter-controlled count that the for loop already counts. While a loop is convenient and may be necessary, if the loop is executing code that is needed for a function. A variable should be assigned outside the loop to be updated in the loop. If the variable is assigned within the loop, the variable does not have the scope to be accessed outside the loop. Making it virtually pointless and doing work for no apparent reason.

Parameters are the variables that are passed into a function. The actual parameters are the values that are passed into the function when the function is called. Actual parameters are the parameters when the function is written and how those variables are manipulated. There are no default parameters in Java. However, there is a way to overload a method to have it use default parameters.

Just like other languages, Java has its own convention when it comes to variable names. All variable names are case-sensitive. Variable names can be as long as possible and the only symbols they could start with are *$* (dollar sign) and _ (underscore). Under convention, the dollar sign is never used and the names should always start with a letter. Variable names cannot start with a number but numbers can be in between letters. Variable names also cannot be a keyword.

The convention for multiple word variable names is camel case which is represented as *camelCase*.

Variable names and conventions:

```
final public static int _classVariable;
public int _instanceVariable;
int $dollarsign;
int _underscore;
int veryLongNameWith2334AndCamelCase;
```

Formal parameter, scope, local variable:

```
// formal parameters
public void parameterReport(int x, double y) {

    int localVariable;

    // i is only visible in the loop
    // scope is only inside the for loop only
    for (int i; i < 3; i++) {
        i = i + 1;
    }
```

Actual parameter:

```
public void run() {
    // actual parameters
    parameterReport(2, 3);
```

3. Data Types

Java provides eight primitive types and a number of other built-in types. The primitive types include several integer and floating point types that differ by the range of values they can hold, a boolean type, and a character type. The integer types are *byte*, *short*, *int*, and *long*, which use one, two, four, and eight bytes of memory respectively to store an integer in two's complement. The smaller types would be used when you are concerned with the memory usage of your program, whereas the larger types would be more concerned with the range of possible values. Below are some examples of how you might declare and initialize some of the integer types.

```
byte b = 127;
short s = 32_767;
int i = 2_147_483_647;
long l = 9_222_372_036_854_775_807L;
```

Note that in order for an integer value to be considered a *long* and not an *int*, you need to include the *L* on the end (you can also use a lowercase *l*, however it can be difficult to distinguish from other characters and symbols).

The floating point types include two types instead of four: the *float* and the *double*. *Float*s use a 32-bit version of the IEEE 754 floating point standard, whereas *double*s use the 64-bit version. Below is an illustration of how to declare and initialize *float*s and *double*s.

```
float f = 2.0f;
double d = 12.04372;
```

In order for a decimal to be considered a *float*, the *f* after the decimal is required, as *double*s are the default. If you wish to specify that a particular decimal value should be a *double*, you can end the value with a *d*.

While other languages may shorten the name to just *bool*, Java keeps the full length *boolean* for its implementation of the type. Below is an example of how to declare and initialize a *boolean*.

```
boolean B = true;
```

Besides the longer name, Java's *boolean* seems fairly similar to C++'s implementation of the type.

The last primitive data type is the *char*, which seems to be the Unicode for whichever character corresponds to it. For example, arithmetic and relational expressions work on *char*s as though they were the corresponding integer. The following lines of code show two ways to initialize a *char* to hold "a".

```
char c1 = 'a';
char c2 = 97;
```

It is important to note that *char*s use single quotes (') surrounding the character, as double quotes (") are reserved for *String*s.

While *String*s are "not technically" a primitive type according to Oracle's Java documentation, the way Java chooses to implement them makes it feel like they are. Below is an example of a string.

```
String string = "Hello World!";
```

While this *String* seems fairly similar to the types we've covered so far, it works slightly differently. For example, when comparing two *String*s, the == operator does not work as expected; instead, you must use the *String*'s *equals* method to compare them.

Java also supports enumerations, arrays, associate arrays, records, lists, and references, which we will cover here. I don't believe there are any built-in enumeration types besides perhaps the *boolean* primitive type, however Java makes it fairly simple to define your own. Below is an example using the days of the week.

```
1 // Day is an enumeration type for the days of the week
2 public enum Day {
3     MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
4 }
```

After declaring your enumeration type like this, making an instance of it is very simple:

```
Day day = Day.MONDAY;
```

If you followed this line of code with a statement printing *day*'s value to the console, it would show *MONDAY*. Java does not support treating enumeration types as integers, so trying to set *day* equal to 0 yields an error.

Arrays in Java seem fairly similar to C++'s arrays. The syntax for creating and initializing arrays is slightly different, however any basic array operation you can do in C++ seems to be supported in Java. Below are two different ways of initializing an array of *int*s.

```
int[] array1 = new int[10];
int[] array2 = {1, 2, 3, 4, 5};
```

This is the first time we've seen the *new* keyword in code, but it essentially just means that we need to call a constructor. This is because everything other than Java's primitive types are objects, and so need to call constructors when they are created. Once you've created an array, you can use the standard bracket notation to access any element in the array that is in bounds.

Associate arrays in Java are more explicitly objects. There are a few different options, however the one that seems most familiar is the *Hashtable*, which is shown below.

```
Hashtable<Integer, Integer> dict = new Hashtable<Integer, Integer>();
for (int num: array1) {
    dict.put(num, num * num);
}
```

Unfortunately, the *Hashtable* doesn't play nicely with primitive types, so to make an array that maps *int*s to *int*s, we need to use Java's *Integer* object. Once you've set up your *Hashtable*, you can use its *put* and *get* methods to store and retrieve data.

Like enumeration types, records are easily defined by the user. Below is a complete declaration of a record that stores a number and its square.

```
public record TestRecord(int num, int square) {}
```

When declared in this way, Java will automatically create a constructor and methods to retrieve the instance variables as shown below.

```
TestRecord r = new TestRecord(2, 4);
r.num();
r.square();
```

Java also supports list types in the form of the *ArrayList*, among other possible types. Just like the *Hashtable*, the *ArrayList* doesn't play nicely with primitive types, so you will need to use the corresponding object for the type you want to use. Below is an example of using an *ArrayList*.

```
List<Integer> L = new ArrayList<Integer>();
for (int j = 0; j < 20; ++j) {
    L.add(j);
}
```

As with the *Hashtable*, bracket notation is not supported, so you will need to use the *add* method as shown above (which either puts it on the end, or inserts it at a given index) and the *get* method to retrieve your data.

Finally, Java supports reference types which seem similar to C++'s smart pointers. Below is an example of instantiating some *Reference*s.

```
Reference<Integer> I = new WeakReference<Integer>(4);
Reference<Integer> J = new SoftReference<Integer>(5);
ReferenceQueue<Integer> refQueue = new ReferenceQueue<Integer>();
Reference<Integer> K = new PhantomReference<Integer>(6, refQueue);
```

*Reference*s don't play nicely with primitive types just as *Hashtable*s and *ArrayList*s, and don't support the syntax C++'s smart pointers use, opting for a *get* method to dereference it.

4. Expressions

Java expressions are all left to right associativity except for unary operators, prefix increment/decrement, assignment operator, and the ternary operator. The precedence rules in Java are as follows:

Postfix increment/decrement	++, - -
Prefix increment/decrement, unary	++, - -, +, -, ~, !
multiplicative	*, /, %
additive	+, -
shift	<<, >>, >>>
equality	= =, ! =
Bitwise AND	&
Bitwise exclusive OR	^
Bitwise inclusive OR	|
Logical AND	&&
Logical OR	| |
Ternary	? :
Assignment	=, + =, - =, * =, / =, % =, & =, ^ =, | =, << =, >> =, >>>=

Assigning a variable is right to left associativity. Where it is variable name = value of data type. *int x = 37.* That statement is also considered a simple and declarative statement. Parentheses may also be needed to make statements unambiguous. For example *x + y / 100*, will evaluate using mathematical PEMDAS. Therefore, if the addition is needed to be executed first using parentheses the expression *(x + y) / 100* becomes unambiguous. Java does not support exponentiation.

```
// simple assignment and declaration
int j = 0;
int k;
// right associativity and unambiguous statement
k = (j + 200) % 3;

// unary operator for i
for (int i = 0; i < 2; i++) {
    // assignment using +=, -=
    k += 2;
    j -= 2;
```

5. Control Structures

Even a very simple program might need control structures to control the flow of its execution. Without control structures, any task that requires something to happen sometimes but not others, or something to happen multiple times, would be impossible. In this sense, there are two main types of control structures: selection statements and iterative statements. Java implements several different versions of each of these depending on what particular task you want to accomplish.

For selection statements, Java provides all the standard options. Its two-way control statement is formatted identically to C++'s, as shown below.

```
if (g % 2 == 1) {
    // code goes here
}
else {
    // other code goes here
}
```

If the condition in parentheses after the *if* statement is true, the code in the first set of brackets will be executed. Otherwise, the code in the brackets after the *else* statement will be executed. Java also allows another *if* on the same line as the *else* to allow for multiple-way control structures. An alternative implementation of multiple-way control structures is the *switch* statement, shown below.

```
switch (i) {
    case 1: System.out.println("test");
    break;
    case 3 + 4: break;
}
```

If you don't use the *break* keyword in a given case, execution will proceed through every case until either the *switch* statement ends or it finds a *break*. In this sense, *switch* is a lot more

versatile than *if ... else if ... else* blocks, however forgetting a single *break* will have unintended consequences.

As for iterative statements, Java provides *for* loops, *while* and *do while* loops, and allows for iteration over a data structure. Java's *for* loops are counter controlled, however the counter variable can be changed within the loop if so desired. Its syntax is pretty much identical to C++'s *for* loops, as shown below.

```
for (char g = 65; g < 100; g++) {
    if (g % 2 == 1) {
        continue;
    }
    System.out.print(++g);
}
```

Java's *while* loops are logically controlled loops that test the given condition before each iteration. An example of one is given below.

```
i = 0;
while (i < 10) {
    System.out.println(i);
    ++i;
}
```

This example would print the numbers from zero through nine (inclusive) to the console, as the body of the loop isn't performed when *i* is equal to 10. Java's *do while* loops are also logically controlled, however the condition is checked after the body is executed. An example of a *do while* loop is shown below.

```
i = 0;
do {
    System.out.println(i);
    ++i;
}
while (i < 10);
```

This example would first print all the numbers the previous example did, and then print 10, as the condition is only checked *after* the body of the loop has run.

Java also supports *break* and *continue* statements. The *break* statement exits the loop immediately and puts execution after the end of the loop, whereas *continue* skips this execution of the loop and continues execution at the start of the next iteration, if applicable. An example of *continue* and *break* being used in a loop are shown on the next page.

```
for (char g = 65; g < 100; g++) {
    if (g % 2 == 1) {
        continue;
    }
    else if (g == 70) {
        break;
    }
    System.out.print(++g);
}
```

This will skip any execution where *g* is odd, and stop the loop when *g* gets to 70.

Finally, Java also supports iteration over a data structure using the *for* keyword. The exact syntax is shown below.

```
for (int num : array1) {
    dict.put(num, num * num);
}
```

As the loop executes, *num* will hold each value in *array1*, which will then be used in the body of the loop.

6. Subprograms

In Java, subprograms within a class can be declared public, private, or protected. However, subprograms without an instance of the class must be declared static. They can be declared public, private, or protected, but that serves virtually no difference in the execution. Since Java code is written inside of a class without the *static* keyword a instance of the class is needed to call a subprogram. This is also why the main function must be declared with the static keyword. After the static and type declaration the user must provide a return type, even if it is just void. Followed by the parameters, which can be left empty if no parameters are needed. In Java the subprograms are passed-by-value and in mode. This means that the values passed to the subprogram will be used inside the program based on the formal parameters and the contents within the subprogram will not change anything from the outside. To have values used and get a result from the subprogram it must return a value. Returns work by using the keyword *return* and the subprogram can only return one of the data types it is declared to return.

```
// subprogram that can be called without
// class instance
static boolean isEven(int num){

    if (num % 2 == 0) {
        return true;
    }
    return false;
}
// main function with static keyword
public static void main(String[] args) {

    boolean response = isEven(3452);
    System.out.println(response);
```

7. Classes

Classes in Java seem to be a main feature: pretty much nothing can exist in Java if it's not a class or a member of a class. Java supports nested classes (classes inside of classes) and inheritance, including multiple inheritance. Three keywords are related to the creation of classes in Java besides *class*: *public*, *abstract*, and *final*.

Classes marked as *public* are accessible to any class in the entire project you're working in. Classes not marked as *public* are package-private, meaning they are only accessible inside the package they are created in. A package is essentially a bundle of related classes, and in general packages allow for a more granular level of access control. Classes marked as *abstract* are Abstract Classes: you cannot instantiate an instance of a class marked *abstract*. In a sense, *abstract* classes are classes that you *must* inherit from in order to use (although you can define class methods in *abstract* classes that can still be used). In contrast, classes marked *final* cannot be inherited from.

Interestingly, in addition to being able to declare variables and methods as *public*, *private*, *protected*, or package-private (by not using a keyword), you can also declare nested classes as any of these, and the same rules apply. Something marked as *public* is accessible to any file in the same project, something marked as *private* is only accessible in the class itself, something marked as *protected* is accessible in any subclass of the given class in any package, and something that's package-private is accessible anywhere inside the package. Additionally, you can declare variables, methods, and nested classes as *static* and *final*.

As we discussed in Section 2, variables marked *static* are class variables, and if they're marked *final* that means you can't change their value. We also discussed in the previous section that methods marked *static* are class methods, meaning you don't need to create an instance of the class to use it, and methods marked *final* cannot be overridden by any subclass. When nested classes are marked *static*, you call the constructor by using *OuterClass.InnerClass* as the class name, whereas a non-*static* nested class requires an instance of *OuterClass* to call the constructor. As with non-nested classes, a nested class marked *final* cannot be inherited from.

Below are some examples of how to declare classes. Note that the keyword *extends* is used to inherit from another class.

```
1 public class TestClass {
2     protected final class Test1 {}
3     public abstract class Test2 {}
4     private static class Test3{}
5     class Test4 extends Test2 {}
6 }
```

8. Exception Handling

Java uses *try, catch, throw,* and *throws* keywords with its exception handling. In Java there are three kinds of exceptions. The first kind is the checked exception. These are written by the user and the program should recover from them, if written correctly. The second kind of exception is error. These are not caused by the program itself, rather typically hardware or a system malfunction. The program cannot recover from this. However, the user could set up a try-catch exception for it to display the error from the program. The third and final exception is a

runtime exception. These are problems that occur during runtime that are not anticipated. The program cannot recover from an runtime exception. Runtime and errors are also known as "bugs" and are considered to be unchecked exceptions. Debugging the program is the best solution to solve an error or runtime exception.

To use a try-catch in a program, the code must use the keyword *try*. Using *try {}*, inside the brackets the program must try to do something. If the try cannot be completed it will then move to the *catch (error type) {}*. Catch is written with parentheses and inside the parentheses the error type must be declared and assign it to a variable. Within the brackets of a catch, the code must have error output written as *System.err.println*. The message that should output should be the error that is caught and typically the message of the error itself from the Java language itself. That can be achieved by the *variable name.getMessage()*. Typically written as *e.getMessage()*. Using the *finally* keyword and brackets, this will execute after the try has been executed. No matter if an exception is caught or not, the code within the finally brackets will execute.

```
int[] num = {1, 2, 3};
boolean result = true;
// trying to index out of array
try {
    System.out.println( num[3] );
}
// index out of bounds exception
catch (IndexOutOfBoundsException e){
    // gets the message
    System.err.println("Caught IndexOutOfBounds Exception: " +
 e.getMessage());
    result = false;

}
finally {
    // out puts the message as to why
    if (result == false) {
    System.out.println("The index of num does not exceed 2");}

    else {
        System.out.println("try sucessful, index is in bounds");
    }
```

In this example, the program is trying to print out from the array with an index that is not possible. The catch, grabs the error and outputs the message. The finally statement then executes the reasoning written by the programmer. The output is as follows:

```
Caught IndexOutOfBounds Exception: Index 3 out of bounds for length 3
The index of num does not exceed 2
```

The next example will be a try-catch, where an error does not occur.

```
int[] num = {1, 2, 3};
boolean result = true;
// trying to index out of array
try {
    System.out.println( num[0] );
}
// index out of bounds exception
catch (IndexOutOfBoundsException e){
    // gets the message
    System.err.println("Caught IndexOutOfBounds Exception: " +
 e.getMessage());
    result = false;

}
finally {
    // out puts the message as to why
    if (result == false) {
    System.out.println("The index of num does not exceed 2");}

    else {
        System.out.println("try sucessful, index is in bounds");
    }
```

Since the catch never occurs, because the try successfully executes, the result never changes to false and prints that the try is successful, because the index is in bounds. The output is below:

```
1
try sucessful, index is in bounds
```

To throw an exception, the program must use the *throw* keyword. Using throw differs from try-catch, as instead of trying something, the program will automatically call the throw the exception if written as so. The reason to throw an exception and not to catch it, allows the programmer to automatically throw an exception without trying it. Therefore, the program can throw an exception without having to attempt code to catch the exception. To throw an exception the exception must be a descendant of the *Throwable* class. Meaning the object being thrown must inherit from the *java.lang.Throwable* class. It is possible for the user to create their own Throwable object. There are also many built in Throwable objects.

```
throw new ArithmeticException("Cannot buy, too young");
```

This will throw the *ArithmeticException* no matter what since it is written to throw it. The output is below:

```
Exception in thread "main" java.lang.ArithmeticException: Cannot buy, too young
        at Report.main(Report.java:37)
```

Typically, when using the *throw* keyword, logical reasoning is used. This way an exception will not be thrown unless a criteria is not met.

```
int age = 6;

if (age < 21) {
    throw new ArithmeticException("Cannot buy, too young");
}
else { System.out.println("Can buy, old enough");
```

This has the same output as throwing the exception without any logic, since the age is not greater than 21. The same message it shown:

```
Exception in thread "main" java.lang.ArithmeticException: Cannot buy, too young
        at Report.main(Report.java:37)
```

Here is an example when the age is old enough and the exception is not thrown:

```
int age = 22;

if (age < 21) {
    throw new ArithmeticException("Cannot buy, too young");
}
else { System.out.println("Can buy, old enough");
```

Since the code never makes it into the if statement, the exception is never thrown. The output is below:

```
Can buy, old enough
```

The keyword *throws* is used in the signature of a method to indicate that the method may throw an exception.

```
static void checkAge(int age) throws ArithmeticException{
    if (age < 21) {
        throw new ArithmeticException("Cannot buy, too young");
    }
    else { System.out.println("Can buy, old enough");
}
}
public static void main(String[] args) {

    int age = 16;
    checkAge(age);
```

The keyword *throws* is not necessary but it helps identify if a function or method is going to throw an exception. The above example executes exactly as before, with both an age less than 21 and an age greater than 21.

```
static void checkAge(int age){
    if (age < 21) {
        throw new ArithmeticException("Cannot buy, too young");
    }
    else { System.out.println("Can buy, old enough");
}
}
public static void main(String[] args) {

    int age = 16;
    checkAge(age);
```

This example above will also execute exactly as before, even without the *throws* keyword.

9. Demo

Our initial idea for our program was some kind of interactive text-based story, similar to the *Choose Your Own Adventure* books. However, trying to write a story on top of the program we would use to tell it would have been difficult, so we decided to make a small quiz program as a sort of proof of concept. When you run the program, you are presented with a question, such as the one you can see below.

```
What is 1 + 1?
a) 1
b) 2
c) 3
d) 4
Enter the letter that corresponds to your answer:
```

When you respond to the question, your answer will be checked to see if it is correct, incorrect, or invalid. If your response was valid (“a”, “b”, “c”, or “d”), the code will tell you if you were correct or not (telling you the correct answer if you got it wrong), and proceed to the next question. If your response was invalid (anything but “a”, “b”, “c”, or “d”), the program will tell you so, and ask for another response. After answering each question, the program will tell you how many questions you got right out of the total number of questions, and give you the corresponding percentage, rounded down to the nearest integer.

Our implementation of this program uses two classes: a *Question* class and a *QuestionAsker* class. The *Question* class has a question as a String, answers as an array of Strings of any length, and an integer indicating the index of the correct answer. The *Question* class also has an *ask* method, which we did not use for this project. The *QuestionAsker* class keeps track of valid responses as an array of Strings, the questions it should ask as an array of our *Question* class, and an integer that keeps track of how many questions the user has answered correctly. It holds the *main* function that runs when the program runs, a *quiz* method that starts the interactive portion of the program, a helper method *getInput* which acts similar to Python’s *input* function, and a *reportScore* method that prints the user’s score to the console.

The *main* function (shown below) initializes an instance of the *QuestionAsker* class and instances of the *Question* class to give a hard-coded quiz. This function primarily demonstrates defining and initializing local variables in Java, as once the *QuestionAsker* class has the information, the *quiz* method does most of the heavy lifting.

```
14    public static void main(String[] args) {
15        String[] as = {"a", "b", "c", "d"};
16        String[] temp1 = {"1", "2", "3", "4"};
17        String[] temp2 = {"1.414", "1.000", "3.141", "2.718"};
18        Question[] qs = {new Question("What is 1 + 1?", temp1, 1),
19                new Question("Which is closest to the square root of 2?", temp2, 0)};
20        QuestionAsker test = new QuestionAsker(qs, as);
21        test.quiz();
22    }
```

Once the *quiz* method is called, it iterates over each of the questions in its array of *Question*s (*qs* in *main*) and asks each of them, using a *do while* loop to check if the input is valid (for which it compares the response to *as*). After it's done asking the questions, the *quiz* method cleans up and reports the score using the *reportScore* method, shown below.

```
76    public void reportScore() {
77        System.out.println();
78        String answers = _correctAnswers == 1 ? "answer" : "answers";
79        System.out.println("You got " + _correctAnswers + " " + answers + " correct out of " + _questions.length + ".")
80        System.out.println("That's about " + _correctAnswers * 100 / _questions.length + "%!");
81    }
```

The *reportScore* method makes use of the ternary operator and the String's + operator to format the output correctly. While Java's primary method of printing to the console takes only one parameter, its overloading of the + operator for concatenating Strings with other types makes putting information in the middle of text simple and easy, at least for what we used it for.
The only remaining method for the *QuestionAsker* class is the *getInput* class method, shown below.

```
70    public static String getInput(String prompt, Scanner in) {
71        // output the prompt and return the response
72        System.out.print(prompt);
73        return in.nextLine();
74    }
```

While simply using these two lines of code anywhere we would use *getInput* would work, having a method to abstract it away makes it much nicer to read the code, and takes away much of the hassle of using the *Scanner* class, which has a similar interface to reading from a file in Python.

Java's implementation of record classes makes storing information in a class much simpler and easier than I know how to in C++. The entire body of the *Question* class is shown below, including the unused *ask* method.

```
1  import java.util.Scanner;
2
3  public record Question(String question, String[] answers, int correctAnswer) {
4
5      public String ask(Scanner in) {
6          // ask the question
7          System.out.println(question);
8          // give possible answers
9          for (String answer : answers) {
10             System.out.println(answer);
11         }
12         // ask for a response
13         System.out.print("Enter your response and press Enter: ");
14         // give it to the caller
15         return in.nextLine();
16     }
17 }
```

Simply declaring the *Question* class as a record automatically implements a constructor and getter methods for each of the parameters given on line 3 (each getter's name is the name of the parameter it gets). This makes writing a class just intended for storing information almost as easy as using a *Tuple* in Python, with the convenience of a data type and named instance variables.

The current implementation of the program definitely leaves room for improvement. For example, there currently isn't a way to read questions or answers from a file (although a program that used these classes could do so), and while the question is output correctly for questions with fewer than 4 responses, the input validation doesn't work properly for such questions ("c" or "d" are considered valid inputs for a question with only two answers provided). A full on *Choose Your Own Adventure* style game would still require a decent amount of work from here, but the fundamental features needed for it are more or less working.

10. Language Evaluation

If the user is familiar to C++, then Java will be quite easy to learn. The syntax and semantics are similar to C++. Java reads and writes just like C++ does. Java is reliable as there are features in place that allow the user to build classes with security. Methods and variables in Java classes can be declared private or protected, meaning the data cannot be changed outside the class. Java also has the keyword *final,* which completely protects something from being changed, allowing things to maintain stability and not messing up the proper usage. As long as the user knows which exception to throw or catch, Java is reliable in either recovering and still executing or terminating a failed program. Java does not support multiple inheritance, so for larger programs that may need it, Java might not be the language to use. Java is completely object-oriented, it has active garbage collection, which allows it to be allocated on the heap without any repercussions, just as a memory leak. Java uses the keyword *new* to instantiate a new instance of a class, this allows the heap memory to be allocated, while Java takes care of the rest of the work. Java is passed by value, which is not necessarily a problem as long as the programmer is experienced. With that being said, there are no pointers in Java, but there is a way to make reference objects that point to the same data. That is shown below:

```
Date d1 = new Date(4, 22, 2022);
Date d2 = d1;
d1.printDate();
d2.printDate();

if (d1 == d2) {
    System.out.println("true");
}
d2.changeYear(2020);

d1.printDate();
d2.printDate();
```

The output for the execution is below:

```
April 22nd, 2022
April 22nd, 2022
true
April 22nd, 2020
April 22nd, 2020
```

The reference object to d1 is d2, as the program is executed, it is shown that the do equal each other. If the user changes the year of the dates, as they did in the example. Changing d2 also changed d1. These references only work with objects, as variables passed into functions will still be passed-by-value. Java does not support pass-by-reference. One final thing to mention about Java is that it is both a compiler and an interpreter making it a hybrid.

11. Reflections

11a. Tyler's Reflection

Java is extremely easy to use and learn. Having decent knowledge on C++ really helped while learning Java. At the beginning using Eclipse was difficult, but towards the end it started to make more sense and it was not a problem. Also at the beginning, trying to initialize a class was challenging. Using the *new* keyword was not difficult, but when you have other languages in your head from other classes, it took me a second to realize my error when running code was trying to initialize a class. I am still learning many things about Java. For the presentation, I briefly looked up exception handling, but writing this report had me diving into the exception handling material. I found that quite interesting. There are also other things I wish I would have known earlier. For example, the keyword final. During the assessment, I had a class variable. The keyword final was not technically needed, but for writing a class it was probably in my best interest to use the keyword final for my class variable. Overall, I enjoyed learning Java, and I am glad Gabe and I chose this language. Before this project, I never looked up documentation for a language, but as I was researching, I found many cool things Java could do. I may look into them in the future during my free time.

11b. Gabe's Reflection

In my opinion, Java feels as if C++ took a step towards Python. Its syntax and general style of programming are similar to C++, however a lot of how things work under the hood (especially with regards to non-primitive data types) work like Python does. I think it was interesting to learn a language by trying things out and looking up when things didn't work like I thought they should. This more trial and error based approach from trying to create a program to do something is definitely different from the classroom based approach where I learned C++ and Python, although I feel my knowledge of those languages is more comprehensive than Java (but that may just be up to not having used it as much yet). I feel like trying to learn this way from scratch would be a lot more difficult because I wouldn't have the vocabulary to search for exactly what I needed, but now that I have a good level of knowledge relating to both these languages, it's probably the easiest way for me to learn more languages going forward. If you haven't tried to learn a language by sitting down and trying to solve a problem (or if it's just been a while since you have), I would recommend it.

References

[1] https://www.geeksforgeeks.org/java-and-multiple-inheritance

[2] https://www.javatpoint.com/reference-data-types-in-java

[3] https://www.geeksforgeeks.org/throw-throws-java

[4] Robert W. Sebesta, Concepts of Programming Languages, 12th edition

[5] https://docs.oracle.com/javase/tutorial/java/index.html

Work Percentages:
Gabe 50%
Tyler 50%