1. The following program segment was proposed by a CS 61B student to print out whether or not a given year is a leap year. A leap year is either divisible by 400, or divisible by 4 but not by 100.

```java
if (year % 400 != 0) {
    System.out.println(year + " is not a leap year");
} else if (year % 100 != 0) {
    System.out.println(year + " is a leap year");
} else if (year % 4 != 0) {
    System.out.println(year + " is not a leap year");
} else {
    System.out.println(year + " is a leap year");
}
```

For which values of `year` given below does the program correctly identify the year as a leap year or not a leap year? How would you fix the program?

1. 1904
2. 1900
3. 1905
4. 2000

**Solution:**

1. 1904 - It will incorrectly say that it is not a leap year.
2. 1900 - It will correctly say that it is not a leap year.
3. 1905 - It will correctly say that it is not a leap year.
4. 2000 - It will correctly say that it is a leap year.

A fixed version:
```java
if (year % 400 == 0) {
    System.out.println(year + " is a leap year");
} else if (year % 100 == 0) {
    System.out.println(year + " is not a leap year");
} else if (year % 4 == 0) {
    System.out.println(year + " is a leap year");
} else {
    System.out.println(year + " is not a leap year");
}
```

2. Suppose I run the following code:
   1. String a = "", b, c, d;
   2. b = "I'm a string!";
   3. c = b;
   4. a = new String();
   5. b = c;

   How many variables, references, objects, and orphaned objects are there after each line of code? (An orphaned object is one which has no references pointing to it, which means that it can no longer be accessed in the program.)

<table>
<thead>
<tr>
<th>Variables</th>
<th>References</th>
<th>Objects</th>
<th>Orphaned objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>After line 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After line 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After line 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After line 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>After line 5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Solution:** A variable is a “box” into which you can put a value (either a primitive value, like an `int`, or a reference to an object).

It should be clear what an object is. The `new` keyword is the only way to create new objects of *user-defined* classes. For built-in classes like `String`s and arrays, there is special syntax for creating new objects (double quotes and array initializers respectively).

A reference is an “arrow”. All references begin inside of a variable and point to an object.

<table>
<thead>
<tr>
<th>Variables</th>
<th>References</th>
<th>Objects</th>
<th>Orphaned objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>After line 1</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>After line 2</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>After line 3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>After line 4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>After line 5</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

3. Suppose arrays were not built-in to Java, and you had to write an `Array` class. How would you declare the instance variable `length`? (For example, would it be `private` or `public`? What type of variable would it be? Would it be `static`?)

An example of a valid (but incorrect) answer:

```java
private static int length;
```

**Solution:** It should be clear that the type of `length` should be numeric. We’ll default to `int`, but `long` would work as well. (`byte` and `short` would be too small.)

Then, the question is what qualifiers should be added. Specifically, what access modifier (`public/private/protected/package`) should it have, should it be `static`, and should it be `final`? (Note: The `protected` and `package` modifiers are rarely used in 61B, so don’t worry if you don’t know them.)

Consider access modifiers. We can write code like `i < nums.length` in Java, which means that the `length` field must be accessible from any class. Thus, it
must be public.

However, this leads to a problem. If it is public, then anyone can read it, but anyone can also change it. For example, it would be legal to say `nums.length = 3`. This is certainly incorrect behavior which we need to prevent. This is done using `final`, which means that once length is initialized, its value can never be changed afterwards. Note this means that an array’s length can never change (not even as a result of code inside the `Array` class) - this is fine, because anyway there is no way to change the size of a Java array.

Now, should length be static? A static variable (also known as a class variable) would mean that the variable belongs to the class, and all objects of that class would share the variable. So, if we made `length` static, then all arrays would have the same length. This is ridiculous - clearly length should be an instance variable, and so we do not use static.

So, our final answer is:

```java
public final int length;
```

4. Write a method `selectionSort` that sorts an array of integers in place (that is, without creating a new array). It uses the following algorithm:

1. Consider the entire array as our subarray.
2. If the length of the subarray is \( \leq 1 \), return.
3. Find the smallest item in the subarray.
4. Switch it with the first item in the subarray.
5. Set the subarray to be all but the first element of the subarray (since that element is in the right place). Go back to step 2.

Make sure you do not create any new arrays!

```java
public static void selectionSort(int[] nums) {
    Solution:
    for (int start = 0; start < nums.length - 1; start++) {
        int minPos = start;
        for (int i = start + 1; i < nums.length; i++) {
            if (nums[i] < nums[minPos]) {
                minPos = i;
            }
        }
    }
}
```
5. Consider the following classes:

```java
public class Mystery {
    public int theMystery = 42;
    public static void printTheMystery () {
        System.out.println(42);
    }
    public void printAgain () {
        System.out.println(theMystery);
    }
}

class Foo extends Mystery {
    public int theMystery = 13;
    public static void printTheMystery () {
        System.out.println(13);
    }
    public void printAgain () {
        System.out.println(theMystery);
    }
}
```

What will be printed out in the following cases? (“Compile-time error”, “Run-time error” and “No output” are valid answers.)

> Mystery m = new Foo();

> System.out.println(m.theMystery);

> m.printTheMystery();

> m.printAgain();

> System.out.println(((Foo) m).theMystery);
Solution: This is a question about static and dynamic types. (Side Note: Don’t get confused by “static” - it has two completely separate meanings. One meaning of static is to make class variables and class methods - see question 3 for more details. Here, we are talking about static types, which refers to the types that the Java compiler infers about your code.)

There are two key ideas to keep in mind in any such question:

1. **The dynamic type must always be a subtype of the static type.** (Note that being the same type counts as a “subtype”.) We will call this “The Invariant”.

2. Static types are determined by the compiler and are used for the sole purpose of making sure that certain kinds of errors don’t happen at runtime (and are instead compile-time errors). *There are no static types at runtime.* On the other hand, dynamic types are the actual types of objects, and are known at runtime.

```
> Mystery m = new Foo();
No output
```

Let’s check The Invariant. The dynamic type is Foo, since the object being created is clearly a Foo. The static type of m is Mystery, since that’s what we declared it to be. Is Foo a subtype of Mystery? Yes, because Foo extends Mystery. So, the compiler accepts this line, and this line will not produce any output.

```
> System.out.println(m.theMystery);
```
These are tricky. Let’s start with the last one - `m.printAgain()`. The compiler knows that the static type of `m` is `Mystery`. So it checks to see if `Mystery` has a method called `printAgain`, and in fact it does. So, the compiler is happy.

However, at run-time, Java looks at the object inside `m`. (Remember, the compiler couldn’t do this.) It realizes that the dynamic type is `Foo`, not `Mystery`, and since `Foo` overrides the `printAgain` method, it calls `Foo`’s `printAgain` method instead of the one in `Mystery` and prints 13. This is called dynamic method lookup.

Dynamic method lookup only applies to non-private instance methods. For private methods, static methods, and variables, the static type is used to figure out which method/variable to use. So, for `m.printTheMystery()`, since we are calling a static method, we look at the static type of `m` which is `Mystery`, and so the compiler decides that we are calling `Mystery`’s `printTheMystery` method instead of the one in `Foo`, even though at runtime the dynamic type is `Foo`. So, `m.printTheMystery()` prints out 42. For the same reason, `m.theMystery` evaluates to 42.

This is basically the same as before, except now we are casting `m` to `Foo` before looking up variables/methods. Casting changes the static type of an expression. There is no way to change the dynamic type of an object. (You can however change the dynamic type of a variable by assigning a different object to it.) This should make sense if you remember that dynamic types are the actual types - why would we be able to change the actual types of objects? Let me say it again - Casting changes the static type of an expression.

So, in the three lines above, we are forcing the static type to be `Foo`. Remember that the dynamic type was already `Foo`, so The Invariant is satisfied. So, in these lines, both the static and the dynamic types are `Foo`, and so in each case we look at the variable/method in `Foo`, and so they all print out 13.
Here, the cast is changing the static type of m from Mystery to... Mystery. In other words, the cast does nothing, and so it this is just like m.printAgain(), and so again due to dynamic method lookup it prints out 13.

> m = new Mystery();
No output

Let’s check The Invariant. The dynamic type is Mystery, while the static type is Mystery, which is fine. So this works and there is no output.

(Note that here, we are changing the dynamic type of m from Foo to Mystery, which happens because we have put in a different object into m. We haven’t changed the dynamic type of any object.)

> m.printAgain();
42

The static and dynamic types are both Mystery, so it calls printAgain in Mystery, which prints out 42.

> ((Foo) m).printAgain();
Run-time error (ClassCastException)

Here we are changing the static type of m from Mystery to Foo. Let’s check The Invariant - the static type is now Foo, and the dynamic type is Mystery, which is not a subtype of Foo. So, The Invariant has been broken by the cast, which results in a ClassCastException (which is a run-time error).

Side note: Why didn’t the compiler realize that you did something bad with your types? Isn’t that its job? The answer is yes, ordinarily the compiler would do this. However, a cast is an explicit promise by the programmer saying “Compiler, I know you can’t prove that the static type is X, but I, the smart human programmer, am telling you it will be. Trust my infinite wisdom.” Of course, the compiler knows better than to trust programmers, and so it says “Fine, I won’t error on you just yet, but I’m going to tell the runtime environment to check that you’re right. If you aren’t right, sucks to be you.” So, even though The Invariant was broken, we only find this out through a runtime error.

> Foo f = new Mystery();
Compile-time error
Let’s check The Invariant. The static type is \texttt{Foo} while the dynamic type is \texttt{Mystery}. This violates The Invariant, and so we get a compile-time error.

(Side note: I said that the compiler doesn’t know about dynamic types. How then did it know to error here?

The answer is that the compiler actually assigns static types to \textit{all expressions}. It assigned the static type \texttt{Mystery} to the expression \texttt{new Mystery()}. It assigned the static type \texttt{Foo} to \texttt{f}. Then, in the assignment, it realized that you were assigning something of static type \texttt{Mystery} to something of static type \texttt{Foo}, and it realized that this could lead to something bad, and so it throws an error.)

\begin{verbatim}
> Foo.printTheMystery();
13
\end{verbatim}

The only trick here is that we are using a \textit{class} to call a method. Is this fine? Yes, as long as the method is \textit{static} (recall that \texttt{static} means that variable/method belongs to the \texttt{class} and does not need an object). This is the case here, and so it works fine and prints 13.

\begin{verbatim}
> Foo.printAgain();
Compile-time error
\end{verbatim}

This is the same case as before, but here the method is not \texttt{static}, and so we are not allowed to use \texttt{printAgain} from the \texttt{Foo} class directly. So, this causes a compile-time error.

Check your knowledge: Notice that the \texttt{printAgain} method in \texttt{Foo} is directly copied from \texttt{Mystery}. However, since \texttt{Foo} extends \texttt{Mystery}, it already inherits \texttt{printAgain}. If we remove the definition of \texttt{printAgain} in the \texttt{Foo} class, will anything change? Test your answer.