The core language for Java, though larger and more intricate than the core language for Scheme or for C, is still not very complicated, as programming languages go. The design of the language reinforces the idea of information hiding: Almost all of the complexity is encapsulated inside class definitions. Java’s standard libraries provide many thousands of classes, and an application programmer working in Java is likely to have access to thousands more that are not part of the standard.

What makes the complexity manageable is that all of these classes are completely compatible with the core language and that their application programming interfaces are reasonably well documented. Programmers don’t need an encyclopedic knowledge of all of those classes. In order to do useful work once they know the core language, they can easily learn about the classes they need as they go along and as they encounter them in other programmers’ code.

In this course, therefore, we’ll study only a few dozen of the most commonly used classes from the standard libraries, such as java.lang.String, java.lang.System, java.lang.Math, and java.util.Random.

1. You can examine the application programming interfaces for these four classes, including descriptions all of their public constructors and members, in the on-line documentation mentioned in the “General Information” handout. Bring up the front-door page for the documentation site at [https://docs.oracle.com/en/java/javase/11/docs/api/index.html](https://docs.oracle.com/en/java/javase/11/docs/api/index.html) and use the “All Classes” list along the lower left side of that page to find the documentation for java.lang.String. Skim the summary descriptions of the public fields, constructors, and methods near the top of the page.

2. Some of the public methods of java.lang.String, such as valueOf, are static, while others, such as toUpperCase, are not. State the difference that this makes in the way the methods are invoked and explain why it was appropriate to make these particular methods static.

3. The rest of the page provides more detailed descriptions of the members of the class. Find the method that you would use to determine whether an instance of the class has a prefix (an initial substring) that exactly matches a given string, and write a method call that has the value true if the value of the variable str has a prefix that matches the string "java/" and false if it does not.

4. Some of the available methods are marked as “deprecated,” and the documentation explains why they should not be used in new Java programs. In some cases, it appears that the methods received this opprobrious classification in versions of the Java standard dating back as much as twenty years. Why have implementers of Java persisted in retaining methods that are known to be incorrect or unsafe to use for all these years? Why not just remove them from the standard and stop requiring implementations to support them?

5. Bring up the documentation for the java.lang.System class and skim the summary descriptions of the public members. Write a method call that outputs "Assertion failed.", followed by a line terminator, to the standard error output stream err.

6. Write a method call that asks the operating system to provide the (string) value of the environment variable LANG and returns that value.

7. The documentation header claims that “the System class . . . cannot be instantiated.” Why can’t a programmer simply write ‘new java.lang.System()’ to get an instance of this class?
8. Skim the documentation for the `java.lang.Math` and `java.util.Random` classes. The `random` method in `java.lang.Math` and the `nextDouble` method in `java.util.Random` both return a pseudorandom `double` value between 0.0 and 1.0. The values returned in either case are uniformly distributed over that range. Why might a programmer for a particular application that needs only random numbers of type `double` want to use random-number generators that are encapsulated in instances of `java.util.Random` rather than just using the static method in `java.lang.Math`?

9. Implement and test a new Java class called `StringVault`. Each instance of `StringVault` should have a private field `secret` for a string value that should be accessible only to methods that can provide the password — another string, stored in another private field (`password`) of the `StringVault` class. The password should be set by the constructor at the time the `StringVault` instance is created. Provide public methods `getSecret` and `setSecret` for (respectively) accessing or replacing the value of the `secret` field. Each of these methods should have an additional parameter for the password. If the password doesn’t match, `getSecret` should return `null` and `setSecret` should have no side effect.

10. Reimplement the `StringVault` class to provide for a separate administrator-only password and two new methods, `getPassword` and `changePassword`, for (respectively) accessing or replacing the value of the `password` field. These should likewise take an additional parameter and succeed only if the additional parameter matches the administrator password. The administrator password should also be set by the constructor and should not be mutable.

11. Add a private, static `String` field `backdoor` to the `StringVault` class and rewrite the methods so that a string that matches the value of `backdoor` is acceptable as an alternative user password and as an alternative administrator password in any instance of `StringVault` whatever.

12. (This exercise is optional and open-ended.) The `backdoor` field that you implemented in the preceding exercise may be a little too visible, since the `StringVault.class` file actually contains the string literal you used to initialize it. Find a way of hiding it better. (Hint: `javax.crypto.spec.SecretKeySpec` and `javax.crypto.Cipher` are in the standard libraries.)