For the exam, make sure you understand that the constructor and any instance initializations defined in the serialized class are ignored during the deserialization process. Java only calls the constructor of the first non-serializable parent class in the class hierarchy.

Finally, let's add a subclass:

```
public class BabyChimpanzee extends Chimpanzee {
   private static final long serialVersionUID = 3L;

   private String mother = "Mom";

   public BabyChimpanzee() { super(); }

   public BabyChimpanzee(String name, char type) {
      super(name, 0, type);
   }
   // Getters/Setters/toString() omitted
}
```

Notice that this subclass is serializable because the superclass has implemented `Serializable`. We now have an additional instance variable. The code to serialize and deserialize remains the same. We can even still cast to `Chimpanzee` because this is a subclass.

Interacting with Users

Java includes numerous classes for interacting with the user. For example, you might want to write an application that asks a user to log in and then prints a success message. This section contains numerous techniques for handling and responding to user input.

Printing Data to the User

Java includes two `PrintStream` instances for providing information to the user: `System.out` and `System.err`. While `System.out` should be old hat to you, `System.err` might be new to you. The syntax for calling and using `System.err` is the same as `System.out` but is used to report errors to the user in a separate I/O stream from the regular output information.

```
try (var in = new FileInputStream("zoo.txt")) {
   System.out.println("Found file!");
} catch (FileNotFoundException e) {
   System.err.println("File not found!");
}
```

How do they differ in practice? In part, that depends on what is executing the program. For example, if you are running from a command prompt, they will likely print text in the same format. On the other hand, if you are working in an integrated development environment (IDE), they might print the `System.err` text in a different color. Finally, if the code is being run on a server, the `System.err` stream might write to a different log file.

Real World Scenario

Using Logging APIs

While `System.out` and `System.err` are incredibly useful for debugging stand-alone or simple applications, they are rarely used in professional software development. Most applications rely on a logging service or API.

While many logging APIs are available, they tend to share a number of similar attributes. First you create a `static` logging object in each class. Then you log a message with an appropriate logging level: `debug()`, `info()`, `warn()`, or `error()`. The `debug()` and `info()` methods are useful as they allow developers to log things that aren't errors but may be useful.

Reading Input as an I/O Stream

The `System.in` returns an `InputStream` and is used to retrieve text input from the user. It is commonly wrapped with a `BufferedReader` via an `InputStreamReader` to use the `readLine()` method.

```
var reader = new BufferedReader(new InputStreamReader(System.in));
String userInput = reader.readLine();
System.out.println("You entered: " + userInput);
```

When executed, this application first fetches text from the user until the user presses the Enter key. It then outputs the text the user entered to the screen.

Closing System Streams

You might have noticed that we never created or closed `System.out`, `System.err`, and `System.in` when we used them. In fact, these are the only I/O streams in the entire chapter that we did not use a try-with-resources block on!

Because these are `static` objects, the `System` streams are shared by the entire application. The JVM creates and opens them for us. They can be used in a try-with-resources statement

or by calling `close()`, although *closing them is not recommended*. Closing the `System` streams makes them permanently unavailable for all threads in the remainder of the program.

What do you think the following code snippet prints?

```
try (var out = System.out) {}
System.out.println("Hello");
```

Nothing. It prints nothing. The methods of `PrintStream` do not throw any checked exceptions and rely on the `checkError()` to report errors, so they fail silently.

What about this example?

```
try (var err = System.err) {}
System.err.println("Hello");
```

This one also prints nothing. Like `System.out`, `System.err` is a `PrintStream`. Even if it did throw an exception, we'd have a hard time seeing it since our I/O stream for reporting errors is closed! Closing `System.err` is a particularly bad idea, since the stack traces from all exceptions will be hidden.

Finally, what do you think this code snippet does?

```
var reader = new BufferedReader(new InputStreamReader(System.in));
try (reader) {}
String data = reader.readLine();  // IOException
```

It prints an exception at runtime. Unlike the `PrintStream` class, most `InputStream` implementations will throw an exception if you try to operate on a closed I/O stream.

Acquiring Input with *Console*

The `java.io.Console` class is specifically designed to handle user interactions. After all, `System.in` and `System.out` are just raw streams, whereas `Console` is a class with numerous methods centered around user input.

The `Console` class is a singleton because it is accessible only from a factory method and only one instance of it is created by the JVM. For example, if you come across code on the exam such as the following, it does not compile, since the constructors are all `private`:

```
Console c = new Console();  // DOES NOT COMPILE
```

The following snippet shows how to obtain a `Console` and use it to retrieve user input:

```
Console console = System.console();
if (console != null) {
   String userInput = console.readLine();
   console.writer().println("You entered: " + userInput);
} else {
   System.err.println("Console not available");
}
```

The Console object may not be available, depending on where the code is being called. If it is not available, System.console() returns null. It is imperative that you check for a null value before attempting to use a Console object!

This program first retrieves an instance of the `Console` and verifies that it is available, outputting a message to `System.err` if it is not. If it is available, the program retrieves a line of input from the user and prints the result. As you might have noticed, this example is equivalent to our earlier example of reading user input with `System.in` and `System.out`.

Obtaining Underlying I/O Streams

The `Console` class includes access to two streams for reading and writing data.

```
public Reader reader()
public PrintWriter writer()
```

Accessing these classes is analogous to calling `System.in` and `System.out` directly, although they use character streams rather than byte streams. In this manner, they are more appropriate for handling text data.

Formatting Console Data

In Chapter 4, you learned about the `format()` method on `String`; and in Chapter 11, "Exceptions and Localization," you worked with formatting using locales. Conveniently, each print stream class includes a `format()` method, which includes an overloaded version that takes a `Locale` to combine both of these:

```
// PrintStream
public PrintStream format(String format, Object... args)
public PrintStream format(Locale loc, String format, Object... args)

// PrintWriter
public PrintWriter format(String format, Object... args)
public PrintWriter format(Locale loc, String format, Object... args)
```

For convenience (as well as to make C developers feel more at home), Java includes `printf()` methods, which function identically to the `format()` methods. The only thing you need to know about these methods is that they are interchangeable with `format()`.

Let's take a look at using multiple methods to print information for the user:

```
Console console = System.console();
if (console == null) {
   throw new RuntimeException("Console not available");
} else {
```

```
   console.writer().println("Welcome to Our Zoo!");
   console.format("It has %d animals and employs %d people", 391, 25);
   console.writer().println();
   console.printf("The zoo spans %5.1f acres", 128.91);
}
```

Assuming the `Console` is available at runtime, it prints the following:

```
Welcome to Our Zoo!
It has 391 animals and employs 25 people
The zoo spans 128.9 acres.
```

Using *Console* with a *Locale*

Unlike the print stream classes, `Console` does not include an overloaded `format()` method that takes a `Locale` instance. Instead, `Console` relies on the system locale. Of course, you could always use a specific `Locale` by retrieving the `Writer` object and passing your own `Locale` instance, such as in the following example:

```
Console console = System.console();
console.writer().format(new Locale("fr", "CA"), "Hello World");
```

Reading Console Data

The `Console` class includes four methods for retrieving regular text data from the user.

```
public String readLine()
public String readLine(String fmt, Object... args)

public char[] readPassword()
public char[] readPassword(String fmt, Object... args)
```

Like using `System.in` with a `BufferedReader`, the `Console readLine()` method reads input until the user presses the Enter key. The overloaded version of `readLine()` displays a formatted message prompt prior to requesting input.

The `readPassword()` methods are similar to the `readLine()` method, with two important differences:

- The text the user types is not echoed back and displayed on the screen as they are typing.
- The data is returned as a `char[]` instead of a `String`.

The first feature improves security by not showing the password on the screen if someone happens to be sitting next to you. The second feature involves preventing passwords from entering the `String` pool.

Reviewing Console Methods

The last code sample we present asks the user a series of questions and prints results based on this information using many of various methods we learned in this section:

```
Console console = System.console();
if (console == null) {
   throw new RuntimeException("Console not available");
} else {
   String name = console.readLine("Please enter your name: ");
   console.writer().format("Hi %s", name);
   console.writer().println();

   console.format("What is your address? ");
   String address = console.readLine();

   char[] password = console.readPassword("Enter a password "
      + "between %d and %d characters: ", 5, 10);
   char[] verify = console.readPassword("Enter the password again: ");
   console.printf("Passwords "
      + (Arrays.equals(password, verify) ? "match" : "do not match"));
}
```

Assuming the `Console` is available, the output should resemble the following:

```
Please enter your name: Max
Hi Max
What is your address? Spoonerville
Enter a password between 5 and 10 characters:
Enter the password again:
Passwords match
```

Working with Advanced APIs

Files, paths, I/O streams: you've worked with a lot this chapter! In this final section, we cover some advanced features of I/O streams and NIO.2 that can be quite useful in practice—and have been known to appear on the exam from time to time!