Unit 2: Libraries, Packages, Components, Types, and Variables

Preview of Coming Attractions

In this unit be sure to look for

- different types of libraries
- packages
- **`import`** command
- primitive types and component types
- variables and their declaration
- input and output components and statements.

They're in a Library

Off-the-shelf software components live in libraries that in Java are organized into *packages*. Which packages are available to a software developer depends on things like where the developer might be working or which classes the developer might be taking. In the CSE 2221/2231 sequence, you will be using packages from three distinct libraries:

- the course libraries,
- the standard Java libraries, and
- lab and project libraries.

The *course libraries* contain a special package called `components`, which is an extensive collection of components to be used throughout the course sequence. The *standard Java libraries* contain a large number of packages that are part of the standard Java distribution and provide a vast selection of components available to all Java programmers. The *lab and project libraries* will contain components needed for specific assignments; for example, there might be a 2221Lab1 library containing special components needed for Lab 1 in CSE 2221.

Ordering Components from a Library

When developing software, you'll want to use specific components from the available libraries. The process of "ordering from a library" is accomplished in Java through use of **`import`** commands. The following code segment shows an example of a Java main program ordering components from libraries:

Code Segment 2-1

```
import java.util.Date;
import components.text.Text;
import components.text.Text1L;

public class Example {
  public static void main(String[] args) {
    // code for body of main goes here
  }
}
```

Notice that Code Segment 2-1 uses three separate **`import`** commands. To get some understanding of these statements, let's begin with the obvious — the names `java.util.Date`, `components.text.Text`, and `components.text.Text1L` don't make a bit of sense. Not to worry, since you'll eventually be able to decipher names like these and, when you can, you'll be well on your way to computer "geekdom". Just for now, here's a short explanation of these statements. **`import`** `java.util.Date` orders a component named `Date` from the package `java.util` (one of the Java standard libraries). **`import`** `components.text.Text` and **`import`** `components.text.Text1L` order, respectively, a component named `Text` and a component named `Text1L` from the `components.text` package (which is part of the course libraries). For the time being, any further explanation of these components can remain shrouded in mystery with no damage done.

Primitive Types

Informally, a *type* defines a set of values and a set of operations that can be performed on those values. The Java language provides built-in types for some common kinds of

values and it also provides new types in the form of components. *Primitive* types (as the built-in types are usually known) differ from component types in a couple of significant ways: first, Java provides special syntax for the operations (e.g., operators for integer or real arithmetic such as +, –, *, /, are built-in to the language), and second, because these types are part of the language itself, we do not have to order them with **`import`** commands. Here are the primitive types available in Java:

- **`int`** — for working with integer-valued data like 13 and –24601
- **`double`** — for working with real-number-valued data like 13.24601
- **`boolean`** — for working with logical-valued data (true or false)
- **`char`** — for working with character-valued data like `'a'`, `'b'`, and `'?'`

(There are 4 more primitive types in Java: **`byte`**, **`short`**, and **`long`** which simply support different ranges of integer values, and **`float`** which represents real-numbered-valued data with lower precision than **`double`**; for now, we will only use the types listed above.)

To summarize, types in Java come in two flavors: built-in, primitive types and types from components. Primitive types can be used directly while component types must usually be ordered from a library by importing them from the appropriate package. By the way, the official technical term for ordering a component from a library is *bringing a component into scope*. There is one more thing that is worth mentioning at this time. The Java standard libraries include a special package called `java.lang` which contains a number of components that are used very commonly in Java programs. What is special about it is that any components defined in this package can be used without the need to explicitly import them. A particularly useful component in this package is the `java.lang.String` component which defines strings of characters. We'll see an example of how to use this component in the next section.

And, one final bit of technical detail — how does the compiler know where to find the various libraries used by a program? The Java compiler uses what is known as the *classpath* (or build path) to find and retrieve the needed components. The classpath

simply defines a list of locations on the computer file system where the compiler should look for the imported components. The compiler simply looks at each location in turn and as soon as the desired component is found it's used and the compiler stops snooping around.

Still Awake?

2-1. What is the purpose of the **`import`** command in Java?

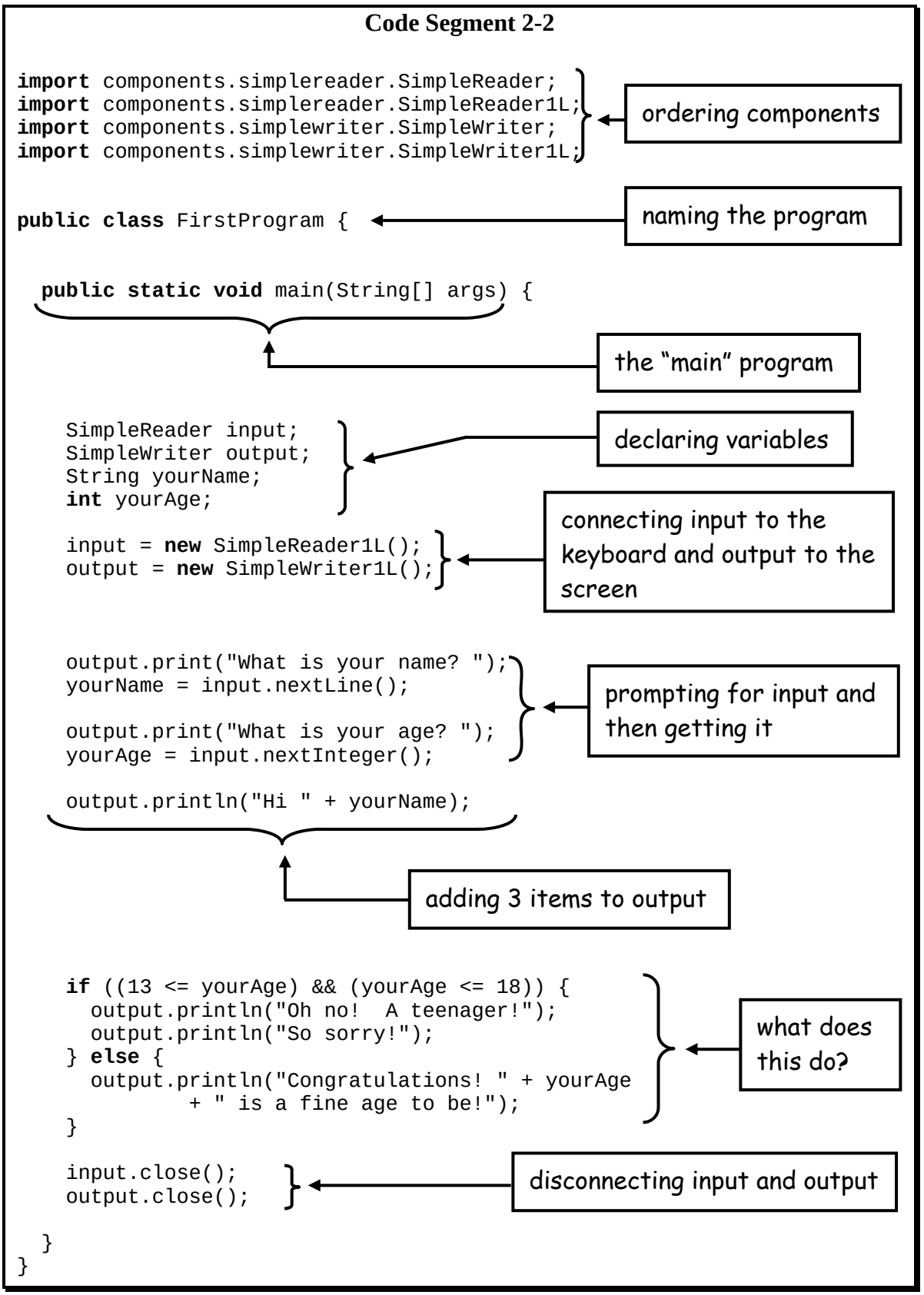
2-2. What are the fundamental differences between primitive types and component types in Java?

2-3. Suppose a program wanted to keep track of whether a water valve is open. For this purpose, which (primitive or component) type would the programmer choose?

Using Components

If you go to the trouble of ordering a component, then probably you would like to put it to use. Here is a short example showing the use of a primitive type (**`int`**), some components imported from the class libraries (`SimpleReader`, `SimpleReader1L`, `SimpleWriter`, and `SimpleWriter1L`) and also the `String` component (implicitly imported from the `java.lang` package).

Code Segment 2-2



After ordering several components and naming the program (`FirstProgram`), we enter the "main" procedure, the one that will be executed when we run the program. The first thing that must be done is to *declare* the *variables* needed by the program. Code Segment 2-2 has four variable declarations:

```
SimpleReader input;
SimpleWriter output;
String yourName;
int yourAge;
```

In Java, you can think of a variable as a container for a value. This container has a name through which we can gain access to the value kept in the container, and a type that determines the values that can be stored in the container. Both the name and the type of a variable are specified when the variable is declared.

Note 1: For convenience we will often use the expression "value of a variable" to refer to the value stored in the variable.

Note 2: Java variables differ from mathematical variables in that a mathematical variable stands for exactly one, possibly unknown, but well defined value, while a Java variable can hold different values at different times in the execution of a program.

Now back to the variables declared in Code Segment 2-2:

- `yourAge` is the name of a variable that can hold an integer value,
- `yourName` is the name of a variable that can hold a text-string value,
- `input` is the name of a variable that can hold an input-stream value (where we will get the input), and
- `output` is the name of a variable that can hold an output-stream value (where we will send the output).

In general, variables in Java are not initialized automatically. In other words, after declaring a variable, the variable does not contain any specific value, i.e., it is uninitialized. This is unfortunate but, on the positive side, the Java compiler will complain whenever you try to use the value of a variable that may be uninitialized. The fix, of course, is to always initialize variables *before* using their values.

Also note that once a component has been imported with its full name (including the package name, e.g., `components.simplereader.SimpleReader`), it can be used with its simpler, shorter name (e.g., `SimpleReader`).

Here's a quick explanation of the other statements in Code Segment 2-2:

```
input = new SimpleReader1L();
```

This statement initializes the variable `input` and "connects" it to the keyboard. Values entered by the user through the keyboard will become part of the value of `input` and can be extracted from `input` by applying appropriate operations to `input`. An explanation of the notation and the use of the **`new`** operator is beyond the scope of this unit, but for the time being, just note the use of another imported component, namely, `SimpleReader1L`, to provide a value for the input variable. Similarly,

```
output = new SimpleWriter1L();
```

initializes the variable `output` and "connects" it to the screen (using a similar notation and the imported component `SimpleWriter1L`). When the program adds values to `output`, these values will appear on the screen.

```
output.print("What is your name? ");
output.print("What is your age? ");
```

The first statement adds the text string `"What is your name? "` to the value of `output`, and `"What is your name? "` also appears on the screen since `output` is connected to the screen. The effect of `output.print("What is your age? ")` is similar.

```
yourName = input.nextLine();
yourAge = input.nextInteger();
```

One of the earlier statements prompts the user to enter a name. The text string the user enters, at the keyboard, in response to the prompt becomes part of the value of `input`, since `input` is connected to the keyboard. The statement `yourName = input.nextLine()` extracts from `input` the line typed by the user and assigns it to `yourName` (thus initializing this variable). The effect of `yourAge = input.nextInteger()` is similar except that the string of characters entered by the user is converted to the corresponding integer value which is then assigned to `yourAge` (so that now this variable is also initialized).

```
output.println("Hi " + yourName);
```

This statement adds three things to the value of `output`: the text string `"Hi "`, the value of the variable `yourName`, and a single *newline* or *end-of-line* character to terminate the line of output. Since `output` is connected to the screen, `"Hi "` and the value of the variable `yourName` appear on the screen. Subsequent output will appear on a new line because we are using the `println` operation (instead of the `print` operation employed previously). Note the use of the `+` operator to concatenate the string `"Hi "` and the value of the variable `yourName`.

The last statement in Code Segment 2-2 is an interesting *if-else* statement.

See if you can figure out for yourself the effect of this statement, including explaining the various output statements. Note that the cryptic && symbol in the *if* condition stands for the logical *and* operator.

Still Awake?

2-4. Modify Code Segment 2-2 so that if the user enters an age between 0 and 3 inclusive, the program outputs a message like

```
"My, just 2 years old!
What a cute little baby."
```

For all other ages, the program just outputs the boring message

`"Thanks for entering your age."`.

2-5. In Code Segment 2-2, suppose the condition

```
(13 <= your_age) && (your_age <= 18)
```

is changed to

```
(13 <= your_age) || (your_age <= 18)
```

where the equally cryptic symbol `||` stands for the logical *or* operator. What message will the program output if the user enters 4 for their age? 15 for their age? 102 for their age?