Lecture 5: Wrapping Up Java

Sam McCauley
Data Structures, Spring 2026

Wrapping up Java Basics

Logical Operators and Short-Circuit Evaluation. In your past experience programming, you have probably seen ways to combine multiple conditionals in, say, an `if` statement. For example, maybe we have code we want to execute if either `x < 0` or `x >= 100`.

In Java, there are three operators that are useful in these contexts.

- && (and): true if *both* conditional expressions are true
- || (or): true if *at least one* of the conditional expressions is true. (Could be both, or could be just one.) The key to press to make this character is located above the Enter key.
- ! (not): converts true into false and false into true

Let's look at some examples.

```java
if(x >= 10 && x <= 99) {
    System.out.println("x has two digits");
}
```

```java
if( !(x >= 10 && x <= 99) ) {
    System.out.println("x does not have two digits");
}
```

```java
//c has type char
if(c == 'a' || c == 'e' || c == 'i'
|| c == 'o' || c == 'u') {   //this line break is legal Java
    System.out.println("c is a vowel");
}
```

Let's say that a number is "interesting" if it is positive, and it is either a prime number or a perfect square. Let's say we have already created methods `isPrime()` and `isSquare()` that take in an integer parameter, and return a `boolean` determining if the number is prime or square. Then the following code determines if a number is interesting.

```java
if(x > 0 && (isPrime(x) || isSquare(x)) ) {
    System.out.println("Look at this interesting number: " + x);
}
```

One important hint relating to the above example: ***always use parentheses if you are using multiple logical operators.*** How precedence works with these operators can be subtle and counterintuitive—parentheses ensure that your code is doing what you expect.

Short-Circuit Evaluation. Java takes a shortcut internally which occasionally has implications on how code is run. This mostly comes up when your conditionals have *side effects*: for example, you call a method to test a variable, and that method also prints to the screen (see the final example below).

If two expressions are separated by &&, and the first is false, we already know that the final expression is false—we don't need to look at the second one. Java takes this shortcut, and does not evaluate the rest of the expression.

If two expressions are separated by ||, and the first is true, we already know that the final expression is true—we don't need to look at the second one. Java takes this shortcut, and does not evaluate the rest of the expression.

Let's look at the following example.

```
 1  public static boolean isEven(int x) {
 2      System.out.println("Testing if " + x + " is even.");
 3      return (x % 2 == 0);
 4  }
 5  public static boolean isLarge(int x) {
 6      System.out.println("Testing if " + x + " is large.");
 7      return x > 10;
 8  }
 9  public static void main(String[] args) {
10      int num = 21;
11      if(isEven(num) && isLarge(num)) {
12          System.out.println(num + " is even and large");
13      }
14      int num2 = 6;
15      if(isEven(num2) || isLarge(num2)) {
16          System.out.println(num2 + " is either even or large, or both");
17      }
18  }
```

Let's trace through the execution of this code. Execution starts at the beginning of the `main` method, on line 10. Java creates num and stores 21 in num.

On line 11, Java calls `isEven(num)` to determine if num is even. This prints "Testing if 21 is even" to the screen. This method evaluates to false. Java looks at the && and determines that the whole `if` statement is false: it does not call `isLarge(num)`.

The program proceeds to line 14; Java creates num2 and stores 6.

On line 15, Java calls `isEven(num2)` to determine if num2 is even. This prints "Testing if 6 is even" to the screen. This method evaluates to true. Java looks at the `||` and determines that the whole `if` statement is true: it does not call `isLarge(num2)`. The program proceeds to line 16, and prints to the screen "6 is either even or large, or both."

Overall, the output of the program is:

```
Testing if 21 is even
Testing if 6 is even
6 is either even or large, or both.
```

Java Types

We've already seen that each variable must be assigned a *type* in Java, but we have not said much about how these types interact.

Mixing Types. What happens when an expression has multiple types?

```java
int x = 4;
double y = 2.1;
System.out.println(x + y);
```

We actually already saw one example of this: if one operand is of type `String`, the other is converted to a `String` and they are then concatenated.

The above example is similar. If a `double` and `int` are summed (or subtracted, multiplied, divided, etc.), the result is a `double`.

Mixing `double`s with `int`s, or arbitrary values with `String`s, is the most common way you'll see types mixed. The Java documentation contains the full list of rules for how this works in general: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.1.2

Similar rules work for assignment. The following code runs in Java.

```java
int x = 4;
double y = x + 1;
System.out.println(y); //prints 5.0
```

However, note that Java will give an error if you are going to *lose* information due to type change.

```java
double x = 4.2;
int y = x - 1; //the .2 is dropped; the compiler gives an error
```

To make this work, you need to explicitly change the type of the data (in this case, turn `x` into an integer). See the discussion below on "Casting."

Types of Constants. An important sticking point is that constants have types in Java as well. Usually this is fairly clear: for example, `"Hello"` is a `String`. Do bear in mind that single quotes, such as: `'A'`, represent a `char` rather than a `String`.

Numerical values with no decimal are assumed to be `int`s. This leads to the following situation:

```
1 int feet = 10;
2 double yards = feet/3;
3 System.out.println(yards);  //prints 3.0
```

What happened here? Let's look at what happens in line 2. Java divides `feet` by 3. That's dividing an `int` by an `int`, so Java uses *integer division*, and obtains 3. Then, Java happily stores this integer value in `yards`.

To avoid this issue, we need to make sure that `feet/3` is not integer division. We could change the type of `feet` to `double` and we'd be all set. Alternatively, we can instead divide by 3.0.

```
1 int feet = 10;
2 double yards = feet/3.0;
3 System.out.println(yards);  //prints 3.33333...
```

Here, when Java sees `feet/3.0`, it sees an operation between a float and an integer. Java first turns `feet` into a `double`, then does the division for `doubles` (retaining the fractional part), storing the result in `yards`.

Below, we will see that we can use casting to solve this problem as well.

Casting. Occasionally we will want to change the type of data. Note that a *variable* cannot have its type changed: if we instantiate `double x`, then x will stay a `double` for as long as it is around. However, it may be useful to change the type of the contents—the information itself. This is called *casting*. We say something like: "cast it to an `int`" to mean "take the value, and convert it to an integer."

The notation to change the type of data is to put the new type in parentheses immediately before the data. Consider the following example (fixing the error from above)

```
1 double x = 4.2;
2 int y = (int)x - 1;
3 System.out.println(y);
4 System.out.println(x);
```

When Java sees `(int)x`, it takes `4.2` and converts it into an `int`, with value `4`. It subtracts 1 to obtain 3, and stores 3 in y (this was integer subtraction, since both sides are `int`s). Note that the value of x is unchanged: Java looked at x to get the value 4.2, and it was this value that was cast to an integer.

Casting is perhaps the clearest way to explicitly tell Java to avoid integer division as well—compare this code with the example above.

```java
int feet = 10;
double yards = (double)feet/3;
System.out.println(yards);  //prints 3.33333...
```

Casting also works with objects. We will be seeing quite a number of examples of how casting works with objects soon. Oftentimes, this casting is done to and from the `Object` class.

Wrapping up Control Flow in Java

break and continue. Two keywords you may have seen in other languages are `break` and `continue`. The `break` keyword in a loop forces the control flow to "jump" out of the loop: the surrounding loop is exited, and control resumes immediately after the loop. The `continue` keyword in a loop causes control to stop processing the current loop iteration, and immediately proceed to the next iteration of the loop.

Use these keywords sparingly: often the loop condition can be written more clearly instead. They are entirely allowed, and can sometimes be invaluable. But before you use either of these keywords, think momentarily if your code can be refactored to avoid their usage. Given the option, it is better to have the loop exit because the condition is false than to use a `break` instead. Let's look at some examples, which try to explain both how these keywords are used, and when they're best applied (or when they should be avoided).

Here is an example where a `break` statement is reasonable to use. The following two code examples accomplish the same thing: print every element of an array, but if you see a `-1`, do not print any further values.

```java
for(int x = 0; x < arr.length; x++) {
    System.out.print(arr[x] + " ");
    if(arr[x] == -1) {
        break;
    }
}
```

Refactoring this without a `break` would be messy! Here's one attempt.

```java
boolean foundANeg1 = false;
for(int x = 0; x < arr.length && !foundANeg1; x++) {
    System.out.print(arr[x] + " ");
    if(arr[x] == -1) {
        foundANeg1 = true;
    }
}
```

Here is a pair of code snippets which both print each element of an array that is an odd integer; one with and one without a `continue` statement. In this example, it is better if the `continue` is refactored out of the code.

Here's an example that uses `continue`:

```java
for(int x = 0; x < arr.length; x++) {
    if(arr[x] % 2 == 0) {
        continue;
    }
    System.out.print(arr[x] + " ");
}
```

Rather than testing for what we don't want and then skipping the loop, we should direct the code using what we *do* want.

```java
for(int x = 0; x < arr.length; x++) {
    if(arr[x] % 2 == 1) {
        System.out.print(arr[x] + " ");
    }
}
```

switch Statements. `switch` statements are used when a variable can be equal to one of several values, and you want to do something different for each. In its simplest form, it can replace a complicated sequence of `if/else if/else` statements—though it is in some ways more expressive. Python does not have an equivalent to this keyword.

A `switch` statement starts with the keyword `switch`, with an expression (usually a variable) in parentheses. In the curly braces for the switch statement, there are a sequence of `case:` statements, listing out possible values for the variable. Java will check which one the variable is equal to, and jump to that part of the code. There is also a `default:` statement, which is what will be executed if the variable matches none of the `case:` statements.

Here's an example of a method that could reasonably use a `switch` statement.[1] The following method is a part of a card game program; the goal is to take in an `int` containing a number that represents a suit, and then print the name of the suit.

First, let's look at how to write code using a sequence of `if/else if` statements. If we call `printSuit(2)` it will print "Clubs"; if we call `printSuit(5)` or `printSuit(-1)` it will print "Invalid Suit!"

```java
public static void printSuit(int suit) {
    if(suit == 1) {
            System.out.println("Spades");
```

[1] If you're familiar with Java you may notice that there are better ways to write this code; probably an enum would be best.

```java
    } else if (suit == 2) {
            System.out.println("Clubs");
    } else if (suit == 3) {
            System.out.println("Diamonds");
    } else if (suit == 4) {
            System.out.println("Hearts");
    } else {
            System.out.println("Invalid suit!");
    }
}
```

Now, let's look at how it is written with a `switch` statement. This version is arguably more readable.

```java
public static void printSuit(int suit) {
    switch(suit) {
        case 1:
            System.out.println("Spades");
            break;
        case 2:
            System.out.println("Clubs");
            break;
        case 3:
            System.out.println("Diamonds");
            break;
        case 4:
            System.out.println("Hearts");
            break;
        default:
            System.out.println("Invalid suit!");
    }
}
```

Note the `break` statements scattered throughout the `switch` statement. This is because in a `switch` statement, we do not *only* run the code for the current case. Rather, the flow of execution jumps to the corresponding case, and then continues until the end of the `switch` statement. The `break` statements are there to prevent this behavior: when encountering a `break`, control goes to the end of the `switch` statement.

Consider the above code without a break statement (this does ***not*** do the same thing as the `switch` statement above.

```java
public static void printSuit(int suit) {
    switch(suit) {
        case 1:
            System.out.println("Spades");
        case 2:
            System.out.println("Clubs");
        case 3:
            System.out.println("Diamonds");
        case 4:
            System.out.println("Hearts");
        default:
            System.out.println("Invalid suit!");
    }
}
```

If we call `printSuit(3)` then it will print:

```
Diamonds
Hearts
Invalid suit!
```

The way a `switch` statement executes all remaining cases is perhaps counterintuitive, but can be used to our advantage. Consider the following code to test if a character is a vowel:

```java
public static boolean isVowel(char c) {
    switch(c){
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            return true;
        default:
            return false;
    }
}
```

Do-while. A do-while loop is much like a while loop, but the condition is checked at the *end* of the loop. This means that the code always runs at least once.

Do-while loops are not very common, but occasionally come in very handy.

Here is the notation for do-while loops in Java. Note that do-while loops, unlike other loops in java, *do* have a semicolon.

```java
do {
    //this code is executed once,
    //and then continues to be executed while the
    //condition is true
}while (/*condition*/);
```

The following is an example use case for do-while loops. Let's say we want to ask the user for input using a method `askUserForMove()`. We check if the move is legal using `isLegal()`; if it's not legal, we ask again. In this case, a do-while loop makes things quite easy:

```java
do {
    move = askUserForMove();
} while(!isLegal(move));
```

The above example is equivalent to the following code. Note that we need to call `askUserFor-Move()` twice—the code is a little less succinct.

```java
move = askUserForMove();
while(!isLegal(move)){
    move = askUserForMove();
}
```

Getting Input in Java

We've discussed how the program can output information to the user using `System.out.print()` and `System.out.println()`. Now let's talk about the reverse: how the program can ask the user for input.

To get user input, we need to import a Java library. At the *top* of the file (before the line saying `public class ...`), we need a line with the following:

```java
import java.util.Scanner;
```

Then, we need to "instantiate" the Java object that looks for input using the line `Scanner reader = new Scanner(System.in);`. Let's look at what we are doing here: we're creating an object, called `reader`, of the class `Scanner`. This is an example of how "(almost) everything in Java is an object." If you want input from the user, you need to instantiate an object to handle this input. I should state explicitly that you can call the object whatever you want; it does not need to be called `reader`.

Once this is run, we can find the next integer the user inputs using `reader.nextInt();` Note that this is a method call for the `reader` object.

If you want a double you can alternatively use `reader.nextDouble();`. Similarly, `reader.nextLine()` gets the next entire line of input (as a `String`).

Let's put this all together. The following program asks the user to input integers until they enter -1; then, it outputs the total of all integers input this way.

```java
import java.util.Scanner;

```

```java
public class SimpleInput {

    public static void main(String[] args){
        Scanner reader = new Scanner(System.in); //set up user input
        int input = 0;  //dummy value to ensure we enter the loop
        int total = 0;
        while(input != -1) {
            total += input;
            System.out.println("Enter the next int: ");
            input = reader.nextInt();
        }
        System.out.println("Total is: " + total);
    }
}
```

Scope in Java

Scope refers to the "lifetime" of a variable; in other words, when can you access certain data in the program? In Java, as in many other languages, the main ideas of scope are quite intuitive—but the details can be quite tricky.

What we Mean by Lifetime. When you declare a variable in Java, your computer allocates memory to hold the variable: somewhere inside your device, there are 1s and 0s that keep track of your variable.

```java
int x;  //declaring a variable x
```

After the variable is no longer being used, Java releases this chunk of memory. Those 1s and 0s that used to hold your variable can now be used for other purposes: for other variables, or even by other programs.

Another way to think about it is in terms of the name of the variable. After we write `int x`, we have a variable `x`, and we can do things like set `x = 0`. In other words, the name "x" is only to be used for this variable. If I was to declare another variable named `x`, I would get an error:

```java
int x;
double x; //Error!
int x; //Also error!
```

When the lifetime of a variable is over, we are free to create a new variable with the same name.

Methods A variable declared in a method is completely local to the method: the lifetime of the variable cannot extend beyond the method.

```
 1 public void doSomethingInteresting() {
 2     int x = 0;
 3     if(x < 5) {
 4         System.out.println("x is " + x);  //OK!
 5     }
 6     //this is the last line where x is available
 7 }
 8 public void doSomethingElse() {
 9     doSomethingInteresting();  //let's call the other method
10     System.out.println("x is " + x); //Error! Lifetime of x ended
11 }
```

If we have a variable that contains useful information, we need to move that information somewhere else. One easy way is to `return` the variable; then the information is passed back to the calling method. We can also store the information in an instance variable—remember that instance variables are available to *all* methods.

Code Blocks: Loops and Conditionals. The lifetime of a variable declared in a loop or conditional ends at the end of the loop or conditional. This can be counterintuitive!

```
 1 public void printSize(int y) {
 2     if(y < 100) {
 3         String response = "small y";
 4         //reached the end of the if
 5     } else {
 6         String response = "large y";
 7         //reached the end of the else
 8     }
 9     System.out.println(response); //Error!
10 }
```

In the above code, each copy of `response` only lasts until the end of their code block. There is no response variable declared by the time we reach line 9.

To fix this code, we must declare the variable outside of the if statement, like in the following.

```
1 public printSize(int y) {
2     String response;
3     if(y < 100) {
4         response = "small y";
5     } else {
6         response = "large y";
7     }
8     System.out.println(response); //works!
9 }
```

The difference between these two code snippets is that in the former, `response` was declared *inside* the `if` statement. In the latter, it was not—it lasts until the end of the surrounding method.

Loops have similar rules. In fact, you may have already taken advantage of this without noticing.

```java
for(int x = 0; x < 10; x++) {
    System.out.println(x);
}
for(int x = 0; x > -10; x--) { //OK!
    System.out.println(x);
}
```

In the above code, we declared `int x` twice. This is OK, because the lifetime of `x` is limited to the `for` loop.

This can be annoying in some cases, however: we cannot use (say) `for` loop indices after the loop finishes. Consider the following code. (From the example towards the beginning of this document: we want to print elements of an array, and stop after we see a -1.)

```java
for(int x = 0; x < arr.length; x++) {
    System.out.print(arr[x] + " ");
    if(arr[x] == -1) {
        break;
    }
}
System.out.println("-1 was found at " + x); //Error!
```

As before, we can fix this by moving the variable declaration out of the loop. (Yes, you can leave segments of a `for` loop empty—though often a `while` loop is clearer in those contexts)

```java
int x = 0;
for( ; x < arr.length; x++) {
    System.out.print(arr[x] + " ");
    if(arr[x] == -1) {
        break;
    }
}
System.out.println("-1 was found at " + x); //OK!
```

Code Blocks: The Real Rule for Java Scope. The rule for Java scope is actually quite simple: the lifetime of a variable is limited by the curly braces that enclose it. This explains all of the above examples: methods, loops, and conditionals all have curly braces, and the scope of the variable is limited to those curly braces.

In fact, you can add *extra* curly braces to your code in Java, and it still works.

```java
public void exampleMethod() {
    //first scope
    {
        int x = 0;
        int y = x + 10;
        System.out.println(y);
    }
    //second scope
    {
        int x = 0; //OK!
        if(y > 100) { //Error!  y is not in this scope
            System.out.println("y is large");
        }
    }
}
```

Instance Variables, Naming and Scope. The following code is *legal* (though not advised!) in Java. You may have parameters with the same name as an instance variable, and you may declare variables with the same name as an instance variable.

```java
/* This class has many variables with the same name as
 * one of its instance variables.
 * Legal but not advised!
 */
public class Student{
    int graduationYear;

    public void exampleParameter(int graduationYear) {
        System.out.println(graduationYear);  //refers to the parameter
    }

    public void exampleLocal() {
        graduationYear = -1;
        System.out.println(graduationYear);  //refers to the variable defined above
    }

    public Student(int newGraduationYear) {
        graduationYear = newGraduationYear;
    }

    public static void main(String[] args) {
        Student s1 = new Student(2026);

        s1.exampleParameter(0); //prints 0
        s1.exampleLocal(); //prints -1
    }
}
```

If you do have instance variables with the same name as parameters or local method variables, you can use the `this` keyword to specify that we're talking about the instance variable. This is most often used in constructors. Some people really like this notation; others think it risks mistakes.

```java
/*This class is legal Java code.
 * using a constructor like this is valid but controversial
 * code style
 */
public class Student{
    String name;
    int graduationYear;

    public Student(String name, int graduationYear) {
        //this.name is the instance variable; name is the parameter
        this.name = name;
        this.graduationYear = graduationYear;
    }
}
```

Classes. One interesting question is what classes are aware of each other. In Data Structures, we will not be answering this question in detail—we will only be working with 2–3 classes at a time, and always in a way that they are all aware of each other.

For our purposes, the relevant fact is that any `classes` stored in files in the same directory on your computer can access each other.

For example, in Lab 2, we had `Move.java` in the same folder as `CoinStrip.java`. In `CoinStrip.java`, we were free to write:

```java
Move m = new Move();
```

We did not need to import `Move.java`—in fact, we needed no setup for this line whatsoever. Java automatically allows files in the same directory to refer to each other.

Classes and Compilation. You may have also noticed in Lab 2 that we only had to compile `CoinStrip.java`, and that Java compiled `Move.java` and `OptimalAI.java` automatically—even if we only ran

```
> javac CoinStrip.java
```

The Java compiler does some legwork for you in this case. When you compile a file, Java looks to see if it references any other files. If it does, Java compiles those as well. So Java sees that `CoinStrip.java` references `Move.java` and `OptimalAI.java`; it compiles all three files every time.

Some Important Considerations for Objects

The `this` keyword. In Java, the `this` keyword is used to refer to the current object. Let's look at an example of why this may be useful.

Let's say we have two classes, `Student` and `Course`. The `Student` class is responsible for handling data about a specific student, while the `Course` class is responsible for handling data about students in general.

Let's say that the `Student` class has the following method to find a study partner for the class. Once the student finds a compatible study partner, they need to call the `createPartners()` method of the `Course` class with the other student and *itself* as an argument; then these two students can be added to the lists of pairs of students in the course.

```java
public void findPartner(Student[] partners, Course theCourse) {
    for(int index = 0; index < partners.length; index++) {
        if(isCompatible(partners[index])) {
            theCourse.createPartners(this, partners[index]);
        }
    }
}
```

The `this` keyword is used so that the `Student` can pass *itself* as an argument.

Lab 2 has another example. In the `onePlayerGame()` method, the `CoinStrip` class needs to pass *itself* to the `findBestMove(CoinStrip game)` method. The `this` keyword is used to accomplish this.

Testing Equality You should never use == to test equality for objects. Instead, you should use the object's `.equals()` method.

Next week, we'll see how to write our own `.equals()` methods. For now, this point is most important for comparing `Strings`.

```java
String s1 = "hello";
String s2 = "hello";
//if(s1 == s2) { //don't do this!!!  Could give unexpected results
if(s1.equals(s2)) {
    System.out.println("These strings are equal:");
    System.out.println(s1);
    System.out.println(s2);
} else {
    System.out.println("These strings are not equal:");
    System.out.println(s1);
    System.out.println(s2);
}
```