CS62 • Misc Java

Things that are boring to teach and better as reference materials

Table of contents

Operators

Operator precedence

Higher on the table = evaluated earlier

Operators	Precedence
postfix	`expr++ expr--`
unary	`++expr --expr +expr -expr !expr`
multiplicative	`* / %`
additive	`+ -`
relational	`< > <= >=`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %=`

Unary Operators

- Unary operators require only one operand.

Operator	Description	Example	
+	Unary plus operator; indicates positive value (not necessary to have)	`int x = +1;`	1
–	Unary minus operator; negates an expression	`x = -x;`	-1
++	Increment operator; increments a value by 1	`++x;`	0
––	Decrement operator; decrements a value by 1	`–x;`	-1

!	Logical complement operator; inverts the value of a boolean	`boolean success = false;`
		`!success;`
			true

Pre vs post-fix operators

- The increment/decrement operators can be applied before (prefix) or after (postfix) the operand.

- The code `result++;` and `++result;` will both end in `result` being incremented by one. The only difference is that the prefix version (i.e. `++result`) evaluates to the incremented value, whereas the postfix version (i.e. `result++`) evaluates to the original value.

- If you are just performing a simple increment/decrement, it doesn't really matter which version you choose. But if you use this operator in part of a larger expression, the one that you choose may make a significant difference.

Pre vs post-fix operators example

```
int i = 3;
i++;
System.out.println(i);  // prints i (4)
++i;
System.out.println(i);  // prints i (5)
System.out.println(++i); // first increments to 6 then
prints it (6)
System.out.println(i++); // first prints i (6) then
increments i to 7
System.out.println(i); // prints i (7)
```

Conditional operators

- The && and || operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. Remember your truth tables!

exp1	exp2	exp1 && exp2	exp1 || exp2
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Even more control flow

`do-while` loop

- Variant of `while` loop that will execute the block of code in the `do` code block once, before it checks if the condition is `true`. It will then proceed as usual.

- Basic syntax:

```
do {

  // code block to be executed

} while(condition);
```

- Make sure your condition terminates otherwise you will enter an infinite loop.

do–while loop example

```
int j = 3;
do {
    System.out.println("This is the best semester ever");
    j++;
}
while(j>5);
```

- Will print

```
This is the best semester ever
```

even though the condition never got satisfied

`break`

- Exits completely out of a `for`, `while`/`do-while` loop.

break example

```
for (int l = 0; l < 10; l++) {
    if (l == 4) {
        System.out.println("I am out of here");
        break;
    }
    System.out.println(l);
}
```

- Will print

```
0
1
2
3
I am out of here
```

`continue`

- Will skip the current iteration of a `for`, `while`/`do-while` loop.

continue example

```
for (int x = 0; x < 5; x++) {

    if (x == 3) {

        System.out.println("I am skipping this step");

        continue;

    }

    System.out.println(x);

}
```

- Will print:

```
0
1
2
I am skipping this step
4
```

`switch` statement

- Use instead of writing many if-else statements.
- Evaluate expression and compare it with the values of each case
- Works with `byte`, `short`, `char`, `int`, and `String`.
- Basic syntax:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

`switch` example

```
int finger = 4;
switch (finger) {
  case 1:
    System.out.println("thumb");
    break;
  case 2:
    System.out.println("index");
    break;
  case 3:
    System.out.println("middle");
    break;
  case 4:
    System.out.println("ring");
    break;
  case 5:
    System.out.println("pinky");
    break;
  default:
    System.out.println("Not a valid number”);
}
```

`break` and `default`

- When Java reaches a `break` keyword, it breaks out of the switch block and does not execute the rest of the code.
    - You need to add a `break` statement otherwise you will go through all the remaining cases!
- The `default` keyword specifies what code to run if there is no case match.

What would happen if we didn’t include break?

```
int finger = 2;
switch (finger) {
  case 1:
    System.out.println("thumb");
 case 2:
    System.out.println("index");
 case 3:
    System.out.println("middle");
 case 4:
    System.out.println("ring");
 case 5:
    System.out.println("pinky");
 default:
    System.out.println("Not a valid number”);
}
```

It will print :

index

middle

ring

pinky

Not a valid number

Ternary operator

- `? :` A conditional operator that is a shorthand for the `if-else` statement.
- Basic syntax:

```
variable = expression1 ? expression2: expression3
```

- Equivalent to:

```
if (expression1) {
    variable = expression2;
}
else {
    variable = expression3;
}
```

Ternary operator example

```
int n1 = 32;

int n2 = 47;

int max;



// Largest among n1 and n2

max = (n1 > n2) ? n1 : n2;



// Print the largest number

System.out.println("Maximum is = " + max);
```

2D arrays & for-each

Review: working with arrays

- Creating a variable to refer to an array

```
int[] numArray; // declares a variable to refer to an array of ints
int numArray[]; //also works but discouraged
```

- Creating and initializing an array

```
int[] numArray = new int[10]; // allocates an array for 10 integers
```

- Creating and initializing an array - shorthand

```
int[] numArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Multi-dimensional arrays

- An array of arrays. Each array, will have its own set of curly braces. E.g.,
  - `int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };`
- To access the elements of a multi-dimensional array, you need first to specify the array and then the element of the array. For example:
  - `System.out.println(myNumbers[1][2]); // Outputs 7`
  - We still count starting at 0!
- To change the value of an element in a multi-dimensional array, you have to index it as above. For example:
  - `myNumbers[1][2] = 9;`
  - `System.out.println(myNumbers[1][2]); // Outputs 9 instead of 7`

https://www.w3schools.com/java/java_arrays.asp

Looping through Arrays: Using a for loop and length

- Arrays have fixed length so a for loop makes sense. E.g.,

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

- Will print

```
Volvo
BMW
Ford
Mazda
```

For-each loop

- A new way of looping through arrays that doesn't need an iteration counter.
- Basic syntax:

```
for (type variableName : arrayName) {
     ...
}
```

- For example:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String car : cars) {
      System.out.println(car);
} //works same as before
```

compare to Python:

```
cars = [“Volvo”, “BMW”, …]
for car in cars:
    print(car)
```

I/O Streams

I/O streams

In Python, it was open()… read()… write()…

- Input stream: a stream from which a program reads its input data
- Output stream: a stream to which a program writes its output data
- Error stream: output stream used to output error messages or diagnostics
- Stream sources and destinations include disk files, keyboard, peripherals, memory arrays, other programs, etc.
- Data stored in variables, objects and data structures are temporary and lost when the program terminates. Streams allow us to save them in files, e.g., on disk or flash drive or even a CD (!)
- Streams can support different kinds of data: bytes, characters, objects, etc.

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Files

- Every file is placed in a directory in the file system.
- Absolute file name: the file name with its complete path and drive letter. E.g.,
  - On Windows: `C:\jli\somefile.txt`
  - On Mac/Unix: `/~/jli/somefile.txt`
- **CAUTION: DIRECTORY SEPARATOR IN WINDOWS IS \, WHICH IS A SPECIAL CHARACTER IN JAVA. SHOULD BE "\\" INSTEAD.**
- `File` class: contains methods for obtaining file properties, renaming, and deleting files. Not for reading/writing!

Writing data to a text file

- `PrintWriter output = new PrintWriter(new File("filename"));`
- If the file already exists, it will overwrite it. Otherwise, new file will be created.
- Invoking the constructor may throw an IOException so we will need to follow the catch or specify rule.
- `output.print` and `output.println` work with Strings, and primitives.
- Always close a stream!

https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html

Writing data to a text file

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteData {
    public static void main(String[] args) {

        PrintWriter output = null;
        try {
            output = new PrintWriter(new File("addresses.txt"));
            // Write formatted output to the file
            output.print("Alexandra Papoutsaki ");
            output.println(222);
            output.print(“Jingyi Li ");
            output.println(111);

        } catch (IOException e) {
            System.err.println(e.getMessage());
        } finally {
            if (output != null)
                output.close();
        }
    }
}
```

need to import relevant classes

call .print or .println to write to file

catch IOException for any errors

.close() the I/O stream

https://liveexample.pearsoncmg.com/html/WriteData.html

Reading data

- `java.util.Scanner` reads Strings and primitives and breaks input into tokens, denoted by whitespaces.
- To read from keyboard: `Scanner inputStream = new Scanner(System.in);`
  - `String input = inputStream.nextLine();`
  - `input` is a String. If you want to convert it into a number, you will need to use the wrapper class of the primitive you want, e.g., `Integer.parseInt(input);`
- To read from file: `Scanner inputStream = new Scanner(new File("filename"));`
- Need to close stream as before.
- `inputStream.hasNext()` tells us if there are more tokens in the stream. `inputStream.next()` returns one token at a time.
  - Variations of next are `nextLine()`, `nextByte()`, `nextShort()`, etc.

Reading data from a text file

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ReadData {
   public static void main(String[] args) {

      Scanner input = null;
      // Create a Scanner for the file
      try {
         input = new Scanner(new File("addresses.txt"));

         // Read data from a file
         while (input.hasNext()) {
            String firstName = input.next();
            String lastName = input.next();
            int room = input.nextInt();
            System.out.println(firstName + " " + lastName + " " + room);
         }
      } catch (IOException e) {
         System.err.println(e.getMessage());
      } finally {
         if (input != null)
            input.close();
      }
   }
}
```

same try...catch...finally structure

use Scanner class

use a while loop to check if file still has lines

.next() is space separated (if you want the whole line, call .nextLine())

close the file

Full example/reference:

https://github.com/pomonacs622025sp/code/blob/main/Lecture3/FileIOExample.java

https://liveexample.pearsoncmg.com/html/ReadData.html

Reading data with a buffered reader

- `import java.io.FileReader;`
- `import java.io.BufferedReader;`

```
FileReader fr = new FileReader("fileToRead.txt");

BufferedReader br = new BufferedReader(fr);

String line = br.readLine();

while ((line!= null) {

    //do something

    line = br.readLine();

}
```

a BufferedReader object takes a FileReader object as input.

the .readLine() method will return null when the file has no more lines to read, so we can write a while loop

You'll see this in HW3: Darwin

Exceptions

Exceptions are exceptional or unwanted events

- They are operations that disrupt the normal flow of the program. E.g.,
  - wrong input, divide a number by zero, run out out of memory, ask for a file that does not exist, etc. E.g.,

```
int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]); // error!
```

  - Will print something like

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
```

and terminate the program.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exception terminology

- When an error occurs within a method, the method throws an exception object that contains its name, type, and state of program.
- The runtime system looks for something to handle the exception among the call stack, the list of methods called (in reverse order) by `main` to reach the error.
- The exception handler catches the exception. If no appropriate handler, the program terminates.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Three major types of exception classes

- Checked Exceptions: Should follow the *Catch or Specify* requirement.
  - errors caused by program and external circumstances and caught during compile time. E.g.,
    - `java.io.FileReader`
- Unchecked Exceptions: Do NOT follow the *Catch or Specify* requirement and are caught during runtime.
  - `Error`: the application cannot recover from. E.g.,
    - `java.lang.StackOverflowError` (for stack)
    - `java.lang.OutOfMemoryError` (for heap)
- `RuntimeException`: internal programming errors that can occur in any Java method and are unexpected. E.g.,
    - `java.lang.IndexOutOfBoundsException`
    - `java.lang.NullPointerException`
    - `java.lang.ArithmeticException`

https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

Useful exceptions to know

- Checked - you have to catch or specify they throw an exception
    - `IOException`: when using file I/O stream operations.
- Unchecked - you don't have to catch/specify them, but it can still be a good idea to do so.
    - `ArrayIndexOutOfBoundsException`: when you try to access an array with an invalid index value
    - `ArithmeticException`: when you perform an incorrect arithmetic operation. For example, if you divide any number by zero.
    - `IllegalArgumentException`: when an inappropriate or incorrect argument is passed to a method.
    - `NullPointerException`: when you try to access an object with the help of a reference variable whose current value is `null`.
    - `NumberFormatException`: when you pass a string to a method that cannot convert it to a number. e.g., Integer.parseInt("hello")

https://stackify.com/types-of-exceptions-java/

The Catch or Specify requirement

- Code that might throw checked exceptions must be enclosed either by
  - a try-catch statement that catches the exception,

```
  try {
         //one or more legal lines of code that could throw an
exception
  } catch (TypeOfException e) {
         System.err.println(e.getMessage());
  }
```

  - or have the method specify that it can throw the exception. The method must provide a throws clause that lists the exception.

```
method() throws Exception{
    if(some error) {
         throw new Exception();
    }
}
```

https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

Catching exceptions

```
method(){
    try {
         statements; //statements that could throw exception
    } catch (Exception1 e1) {
            //handle e1;
    }
    catch (Exception2 e2) {
            //handle e2;
    }
}
```

- If no exception is thrown, then the catch blocks are skipped.
- If an exception is thrown, the execution of the try block ends at the responsible statement.
- The order of catch blocks is important. A compile error will result if a catch block for a more general type of error appears before a more specific one, e.g., **Exception should be after ArithmeticException.**

https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.6

finally block

- Used when you want to execute some code regardless of whether an exception occurs or is caught

```
method(){
    try {
         statements; //statements that could thrown exception
    } catch (Exception1 e) {
            //handle e; catch is optional.
    }
    finally {
            //statements that are executed no matter what;
    }
}
```

- **The `finally` block will execute no matter what. Even after a `return`.**

https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html

Misc review

The simple assignment operator

- One of the most common operators that we’ve already encountered is the simple assignment operator “="; it assigns the value on its right to the operand on its left. For example:
  - int age = 19;
  - int year = 2024;

Arithmetic operators

- Java arithmetic operators support addition, subtraction, multiplication, division, and remainder/modulo.

Operator	Description
+	Additive operator (also used for String concatenation)
–	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

Other assignment operators

- The assignment operators +=, -=, *=, /=, and %= are a compound of arithmetic and assignment operators.

- They operate by adding/subtracting/multiplying/dividing/taking the remainder of the current value of the variable on the left to the value on the right and then assigning the result to the operand on the left. E.g.,

- `num1 += num2;` means `num1 = num1 + num2;`

Equality and relational operators

- Determine if one operand is greater than, less than, equal to, or not equal to another operand

Operator	Description
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Practice problems

Practice problems

Assume you are given the following Java code. What would be printed on your screen?

```
int result = 1 + 2;
System.out.println("1 + 2 = " + result);
int original_result = result;

result = result - 1;
System.out.println(original_result + " - 1 = " + result);
original_result = result;

result = result * 2;
System.out.println(original_result + " * 2 = " + result);
original_result = result;

result = result / 2;
System.out.println(original_result + " / 2 = " + result);
original_result = result;
```

Answer

1 + 2 = 3

3 - 1 = 2

2 * 2 = 4

4 / 2 = 2

2 + 8 = 10

10 % 7 = 3

Worksheet time!

1. Consider the following code:

```
int i = 10;
int n = i++%5;
```

   a. What are the values of i and n after the code is executed?

   b. What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i)? That is, the code was:

   ```
   int i = 10;
   int n = ++i%5;
   ```

Worksheet answers

- a. i is 11, n is 0 (since i++ evaluates first, then increments i)
- b. i is 11, n is 1 (since ++i increments i before evaluation)

Worksheet time!

- What does this print?

```
int n1 = 10;
int n2 = 47;
int n3 = 4;
System.out.println((n1%n3>n2%n3) ? (n1+n2):(n1-n2));
```

Worksheet answers

- `(n1%n3>n2%n3) ? (n1+n2):(n1-n2)`
- 10%4 = 2, 47%4 = 3. 2 > 3 is false, so we evaluate n1-n2, or 10-47, so it prints **-37**.

Worksheet time!

- Declare and initialize an array of strings with all the classes you are taking this semester.
  - Remember the word `class` is a reserved word, you cannot use it to name your variables.
- Write a for loop that loops through each class
- If a class is called “CS62” you need to print “CS62: This is the best class ever, no need to see more” and break the for loop.
  - We will use the equals method to compare equality among Strings.
  - e.g., someString.equals(someOtherString)
- Otherwise, if a class is called “CS101”, you need to print “CS101: New CS achievement unlocked” and continue to the next iteration.
- Otherwise, print the name of the class.

Worksheet answers

- You could have also used a regular for loop instead of a for-each loop.

```
String[] classes = {"PHYS32", "CS101", "ANTH51", "CS62", "IMAG2"};
for(String myClass:classes){
    if(myClass.equals(”CS62”)){
         System.out.println("CS62: This is the best class ever, no need to see more");
         break;
    }
    else if(myClass.equals(“CS101”)){
        System.out.println("CS101: New CS achievement unlocked");
        continue;
    }
    System.out.println(myClass);
}
```

do you need the continue statement?