CS120 — Software Development I

J Maraist

Study Guide — Spring 2018

Outline

Contents

Lecture 1 1
Lecture 2 4
Lecture 3 8
Lecture 4 11
Lecture 5 13
Lecture 6 14
Lecture 7 16
Lecture 8 18
Lectures 9 and 10 20
Lectures 11 and 14 24
Lecture 12 29
Lecture 13 32
Lectures 15-17 35
Lecture 18 41
Lecture 19 43
Lectures 20-22 50
Lecture 23 52
Lecture 24 56
Lecture 25 59
Hints on selected exercises 62

Guides to additional lectures will be uploaded as they are available

1 Lecture 1

1.1 Introduction
A program, and what can happen with it

```java
public class Kilograms {
    public static void main (String [] args) {
        double pounds = 20.0;
        System.out.print(pounds);
        System.out.print(" pounds is ");
        System.out.print(pounds / 2.2);
        System.out.println(" kilograms.");
    }
}
```

In general

- A *compiler* translates the program into machine code
- Computer hardware runs the program

For Java in particular

- The compiler translates the program into *Java class files*
  - Like machine code, but not specific to any machine
- The *Java virtual machine* interpreter runs class files on your computer

And we have electronic textbook

- The server runs your code, and displays the result elsewhere on the same web page

### Parts of the program

```java
public class Kilograms {

    public static void main (String [] args) {
        double pounds = 20;
        System.out.print(pounds);
        System.out.print(" pounds is ");
        System.out.print(pounds / 2.2);
        System.out.println(" kilograms.");
    }

}
```

### Add comments to describe what the program does

```java
/**
 * Converter from pounds to kilograms.
 */
public class Kilograms {

    public static void main (String [] args) {
        double pounds = 20; // Amount to convert to kilograms
        System.out.print(pounds);
        System.out.print(" pounds is ");
        System.out.print(pounds / 2.2);
        System.out.println(" kilograms."); // End of this line
    }

}
```
The other thing that happens with a program — errors!

Compiler errors

- An error that keeps the compiler from translating your program to a class file
- Nothing to run, so no inputs consumed and no outputs produced

Runtime errors

- The compiler does not notice anything wrong
  - Or maybe it gives a warning, but lets us run it anyway — some times it does work out OK
- An error that comes up while the program runs
- Some input consumed, some output produced — but then it cannot continue

The six things a program can do

1. Get input
2. Give output
3. Do arithmetic
4. Update a stored value
5. Test a condition, and select an alternative
6. Repeat a group of actions

Only six things!
• If that’s all a computer can do, maybe that’s all we have to do this semester??
• There’s a whole bunch of detail and skill associated with each of these
• We will see common patterns of combining the Six Things
• There are also questions about organizing our programs and data...

Four ways Java will help you organize your work

1. Defining sequences of operations
2. Grouping related data together
3. Associating data with operations relevant to the particular data
4. Naming these groups, sequences and associations for easy and repeated use

Exercise 1.1. In Eclipse, create a new project called Lab 1 - Hello World, and within it create a class called HelloWorld (note that there are spaces in the project name, but there are not spaces in the class name). Place the following import declaration at the beginning of the file, above the start of the class definition (public class HelloWorld...)

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

Create a main method for class HelloWorld, or fill in the main routine if it creates one for you, like this:

```java
public class HelloWorld {
    public static void main(String[] args) {
        final Scanner scan = new Scanner(System.in);
        System.out.print("What is your name? ");
        final String name = scan.next();
        System.out.println("Hello " + name + "! Welcome to Java!");
        scan.close();
    }
}
```


2 Lecture 2

Declarations

• Creates a place in the computer for a value to be stored
  – Give the place a name
  – Specify what type of item goes there
    * Java is strongly typed - once we declare a particular type, we have to be consistent
    * So once an integer, always an integer; once a Scanner, always a Scanner
  – Assign an initial value to it
• We saw in the first lecture:

```java
Scanner scnr = new Scanner(System.in);
int quantity = scnr.nextInt();
```
• In today’s reading:

```java
int litterSize = 3;
int yearlyLitters = 5;
int annualMice = 0;
```

**Pick good names**

• Use lower camel-case for variable names.
  
  – Named after the “humps” of upper-case letters in the middle of the name

• Descriptive names, but be reasonable with length

• Use letters, and maybe numbers at the very end

  • Mnemonic - assisting the memory
  
  • Consistent - ease understanding

**It’s an update, not an equation**

```java
public static void main (String [] args) {
    Scanner scanner = new Scanner(System.in); // 0
    int total = 0; // 1

    System.out.print("Enter a number: "); // 2
    int num = scanner.nextInt(); // 3
    total = total + num; // 4

    System.out.print("Enter another number: "); // 5
    num = scanner.nextInt(); // 6
    total = total + num; // 7

    System.out.print("Just one more number: "); // 8
    num = scanner.nextInt(); // 9
    total = total + num; // 10

    System.out.println("Their sum is " + total); // 11
    return; // 12
}
```

**Exercise 2.1.** Step through this program with inputs 4,7 and 2

**Many types of numbers**

You’ve seen already:

• int — Rounded integer values

• double — Real-number values

Java also has:

• long — Integer values from a larger range
  
  – int runs from -2,147,483,648 to 2,147,483,647
  
• **byte** and **short** — Integer values from shorter ranges
  – **byte** runs from -128 to 127
  – **short** runs from -32,768 to 32,767
• **float** — Less accurate real-number values
  – There are limits not just in magnitude, but also in accuracy.
  – **float** runs from about \(-10^{38}\) to \(10^{38}\) with about 7 significant digits of accuracy
  – **double** runs from about \(-10^{308}\) to \(10^{308}\) with about 16 significant digits of accuracy

Generally:
• Use **int** or **long** normally
• Never use a floating-point type when an integer will do
• Only use **byte** or **short** to really make a point about the limited range

Operations on numbers
• Usual arithmetic: +, -, *, /
  – But notice that integer division may not be what you expect!
  – Another basic operator: **modulus** \(\%\)
• Many other mathematical function are provided as methods in a standard Java library
  – Its full name is `java.lang.Math`
  * Classes in `java.lang` (unlike `java.util`) do not need an import
  * Documentation is online: [https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)

Numeric operations example
```
public class NumOpsExample {
    public static void main (String [] args) {
        final Scanner scanner = new Scanner(System.in); // 1
        System.out.print("Enter a number: "); // 2
        int given = scanner.nextInt(); // 3
        final int doubled = 2*given; // 4
        System.out.println(doubled); // 5
        return; // 6
    }
}
```

Exercise 2.2. Answer these questions by writing short Java programs (or start from the `NumOpsExample` above, and make changes):
• Does subtraction group to the **left**, or to the **right**? That is, when we ask Java to evaluate 100–50–10, will it evaluate (100–50)–10, or will it evaluate 100–(50–10)?
• Does division group to the left, or to the right?
• Does modulus group to the left, or to the right?
• Does multiplication take **precedence** over addition, as it does in school algebra? That is, when we ask Java to evaluate 100+50×10, will it evaluate (100+50)×10, or will it evaluate 100+(50×10)?
Exercise 2.3. Answer the following questions using the java.lang.Math documentation

- What methods does Java provide for logarithms?
- What methods does Java provide for trigonometry?
- What is the difference between floor and ceil?
- What is the difference between floor and round?
- What do the signum methods do? Why are there two of them?

Exercise 2.4. Write a class ClockTime whose main method reads in a number of seconds, and prints that length of time as a number of hours, minutes and seconds written with a colon between them in the way we usually write clock times.

For example, for the input 5025 the program should output 1:23:45.

What happens when you run your program on input 7260? Is the output what you would write (or expect to read) for a clock time? If not, why not? We will come back later to this program and fix this problem.

Exercise 2.5. A TwoMult sequence is a sequence of numbers where each number (after the first two) is the product of the two prior numbers. Write a Java class TwoMult whose main method reads the first two numbers of a TwoMult sequence from the user, and prints the next three numbers of the sequence.

Exercise 2.6. Write a Java class TempConverter whose main method prompts for and reads a Fahrenheit temperature as a double value, and prints the equivalent Celsius temperature.

Strings

- We’ve printed strings, but in fact they are values just like numbers and booleans
  
  ```java
  final String greeting = "hello";
  final String name = "Jim";
  ```

- The built-in operator on strings is concatenation, written with the + sign
  
  ```java
  final String greetingAndName = greeting + name;
  System.out.println(greetingAndName);
  ```

  would print out helloJim — no space! We must explicitly include a space if we want one
  
  ```java
  final String greetingAndSpaceAndName = greeting + " " + name;
  ```

Converting numbers into strings

So we have two meanings for + in Java

- It denotes adding numbers
- It also denotes concatenating strings
- We say that addition is an overloaded operator

So what if we write + between a number and a string?

```java
final String name = "Jim";
final int number = 2000;
System.out.println(name + number);
```

In this case, Java assumes that we want to convert the number into a String
• We could also write

```java
final String name = "Jim";
final int number = 2000;
final String combo = name + number;
System.out.println(combo);
```

**Example of converting numbers into strings**

We could print a message along with the doubled number

```java
public class NumOpsExample {
    public static void main (String [] args) {
        final Scanner scanner = new Scanner(System.in); // 1
        System.out.print("Enter a number: "); // 2
        int given = scanner.nextInt(); // 3
        final int doubled = 2*given; // 4
        System.out.println("Twice " + given + " is " + doubled); // 5
        return; // 6
    }
}
```

## Lecture 3

**PITFALL: About exceptions**

Some errors occur at compile time

• In these cases, Eclipse will not even let us run the program

Some error cannot be detected at compile time, and show up only when the program is running

• The program stops with an error message

• Later, we’ll look at code that generates or catches exceptions

• For now, you should keep this in mind when your program does not behave as you expect

**Exercise 3.1.** Use (alter if you need) one of the programs from the book or an exercise to make Eclipse throw an exception. What does it look like?

**REVIEW: Four ways Java will help you organize your work**

1. *Defining sequences of operations*

2. Grouping related data together

3. Associating data with operations relevant to the particular data

4. *Naming these sequences, groups and associations for easy and repeated use*
Methods
Methods are groups of operations
• They have a name, and we can refer to them by name
• We can provide values to a method, and the method’s operations can use those values
  – Called parameters
  – Each of these values has a definite, specific type
• The method can provide a return value representing the result of its work
  – Also has a definite, specific type
  – Or, it should be declared void if it does not return a value

Specifying methods
For example
public static double
toCelsius(double degreesFahrenheit) {
  return (degreesFahrenheit - 32.0) * 5.0 / 9.0;
}
• Named toCelsius
• One parameter, of type double
• Returns a double value
  – The declared type and the type of the value with the return agree

Where do we find methods?
• We can define our own methods
  – Like toCelsius
• Java provides many standard methods for us
  – Like the Math methods we saw last week

How do we use methods?
One method can be called from another method
• Methods help structure the programs we write
• For example:

  public class Temperatures {
    public static double
toCelsius(double degreesFahrenheit) {
      return (degreesFahrenheit - 32.0) * 5.0 / 9.0;
    }
    public static void main(String[] args) {
      System.out.println("75F is "+toCelsius(75.0)+"C");
      System.out.println("sin(pi/2) is "+Math.sin(Math.PI/2));
    }
  }
prints

\[ 75F \text{ is } 23.88888888888889C \]
\[ \sin(\pi/2) \text{ is } 1.0 \]

**Exercise 3.2.** Write static methods \( f_1, f_2 \) and so on implementing the following mathematical functions on real numbers (double). Do not use methods from the Math class for these.

1. \( f_1(x) = 2x + 1 \)
2. \( f_2(x, y) = x^2 + 2xy + y^2 \)
3. \( f_3(u) = u^3 + 2u^2 - 3u + 10 \)
4. \( f_4(w) = \frac{w+1}{w-1} \)
5. \( f_5(z) = f_3(z) + f_4(2 + z^2) \)

What happens when we call \( f_4(1) \) from a main method?

**Exercise 3.3.** Write static methods \( g_1, g_2 \) and so on implementing the following mathematical functions on real numbers (double). Do use methods from the Math class for these.

1. \( g_1(x) = \sqrt{2x^2 + 1} \)
2. \( g_2(x, y) = \log_x y \)
3. \( g_3(w) = |w + 10| \)
4. \( g_4(z) = z^{200} \)

What happens when we call \( f_4(1) \) from a main method?

**Exercise 3.4.** Convert your clock-time program from Exercise 2.4 into a method. Give your method the name `getClockTime`. It should take an integer number of seconds, and return a String representing the clock time.

**Exercise 3.5.** Convert part of the main method of your TwoMult class from Exercise 2.5 into a method `printTwoMult`. It should take the first two numbers of a TwoMult sequence, and print the next three numbers of the sequence. Your main method should read in two integers as before, but should make a call to `printTwoMult` instead of calculating and printing the numbers itself.

**Exercise 3.6.** Most most cereals are made primarily of flour, sugar and high-fructose corn syrup. Write a class `CerealMaker` with a static method `announceComposition`. Your method should take three integer arguments, representing (respectively) the number of grams of flour, sugar and high-fructose corn syrup in a standard serving of some particular cereal. Your `announceComposition` method should print a well-formatted announcement of the total number of grams in a standard serving, repeat the number of grams and the name of each ingredient, and then print the total percentage of the standard serving which is sweetener. Your `announceComposition` method should not return any result.

**Exercise 3.7.** Starting with your class `ClockTime` from Exercise 3.4, discard any main method you may have kept from Exercise 2.4 so that your class contains only the `getClockTime` method. Add a new main method which validates your method’s behavior. Your method should print several lines of the form

For 5025, expected "1:23:45", got "1:23:45"

where the first string is literally written out in your `println` statement, and the second string is what your `getClockTime` method returns.
Exercise 3.8. You have probably run across the factorial function in your math classes. It is defined by two rules:

\[
0! = 0 \\
\text{when } n > 0 \\
n! = n \cdot (n-1)! 
\]

We have not yet learned enough Java to implement a factorial method. But we *can* get ready for when we implement factorial, by writing methods to test our implementation. Notice the difference with Exercise 3.7— in the previous one we checked what a method *was already doing*; now, we are setting expectations for what a method *will do*. This approach is called *test-driven development*— we write tests *first*, so that our goals are clear, and so that we can know when our method is correct.

We *stub* the factorial method by writing an implementation which we know is wrong, but which will compile and run with our tests. By making our tests compile and run (albeit with incorrect results), when we do develop the factorial method, we can do so without worrying that our test infrastructure is lacking.

```java
public class FactorialTester {
    public long factorial(int n) {
        // TODO --- later we will implement factorial correctly
        return -1;
    }
}
```

So starting from the above class, add a `main` method which tests `factorial` on several different values.

Exercise 3.9. Write a class `NumberLengthFinder` with static method `getCharacterLength` that tells us how many digits it takes to write down a number (in base-10). Your `getCharacterLength` should

- Take a single `int` argument, and
- Return an `int` result.

(For a hint, see p.62)

4 Lecture 4

REVIEW: From the book

You’ve read about how Java allows us to describe the fifth Thing that a Computer Can Do: *selection*

```java
if (CONDITION) {
    // SOME STATEMENTS
} else {
    // SOME OTHER STATEMENTS
}
```

Today we will look at some examples of using if-statements

Exercise 4.1. Write a class `Grader` with static method `getLetterGrade` which takes an integer argument representing to a percentage grade from 0 to 100, and returns a string representing the corresponding letter grade,

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>95 ≤ g</td>
<td>A</td>
</tr>
<tr>
<td>92 ≤ g &lt; 95</td>
<td>AB</td>
</tr>
<tr>
<td>86 ≤ g &lt; 92</td>
<td>B</td>
</tr>
<tr>
<td>82 ≤ g &lt; 86</td>
<td>BC</td>
</tr>
<tr>
<td>73 ≤ g &lt; 82</td>
<td>C</td>
</tr>
<tr>
<td>60 ≤ g &lt; 73</td>
<td>D</td>
</tr>
<tr>
<td>g &lt; 60</td>
<td>F</td>
</tr>
</tbody>
</table>

As a first step, write a `main` method with examples and expected grade calculations.
Exercise 4.2. WidgetCo manufactures several different kinds of widgets for re-sale by various vendors. Based on past relationships, sales targets, and other factors, certain vendors are given discount codes which entitle them to a particular discount on their purchases. Write a class `WidgetCoDiscounts` with a static method `getDiscountedPrice` which takes two arguments, a string discount code and an integer base purchase price, and returns the price which should be charged given the particular discount code.

<table>
<thead>
<tr>
<th>Code</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>10%</td>
</tr>
<tr>
<td>R</td>
<td>12%</td>
</tr>
<tr>
<td>T</td>
<td>15%</td>
</tr>
<tr>
<td>M</td>
<td>3%</td>
</tr>
<tr>
<td>E</td>
<td>8%</td>
</tr>
</tbody>
</table>

If the discount code is an empty string or does not match any in the above table, then the method should return the original base purchase price.

As a first step, write a `main` method with examples and expected price calculations, for example

```java
System.out.println("For code S and purchase $100, expect 90, got " + getDiscountedPrice("S", 100));
```

Remember that we compare strings with `.equals`, but compare numbers with `==`.

Exercise 4.3. Write a class `MonthNamer` with a static method `getMonthName` which

- Takes a single integer corresponding to a month of the year, 1 representing January through 12 representing December, and

- Returns a string for the name of the month.

As a first step, write a `main` method with examples and expected downpayment calculations.

Exercise 4.4. Square Deal Credit Union offers a program for first-time home buyers to save on the downpayment required for their loan. The downpayment is calculated according to the following table:

<table>
<thead>
<tr>
<th>Purchase price of home</th>
<th>Downpayment required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Under $50,000</td>
<td>4% of price</td>
</tr>
<tr>
<td>$50,000-$124,999</td>
<td>$2,000 plus 8% of price over $50,000</td>
</tr>
<tr>
<td>$125,000-$175,000</td>
<td>$8,000 plus 12% of price over $125,000</td>
</tr>
<tr>
<td>Over $175,000</td>
<td>Not eligible for this program</td>
</tr>
</tbody>
</table>

Write a class `SquareDeal` with static method `getProgramDownpayment` which takes one integer argument representing the home purchase price, and returns an integer representing the required downpayment under this program, or -1 if either the home is not eligible or a negative price is entered. As a first step, write a `main` method with examples and expected downpayment calculations.

Exercise 4.5. Write a class `ThreeSorter` with static method `printInOrder` which

- Takes three arguments of type `int`,

- Returns nothing, and

- Prints the three numbers in ascending numeric order.

As a first step, write a `main` method with examples and expected output.
Exercise 4.6. The Wisconsin Wants Walnuts company buys black walnuts from individuals, and shells them for sale to restaurants. Sellers’ walnuts are loaded into baskets which hold about one pound of unshelled nuts, and WWW pays $0.15 for each full basket of nuts. If the final, partially-filled basket is more than half-full, WWW pays the full $0.15 for that basket; otherwise they pay $0.05 for the partial basket. Write a class WalnutBuyer with a static method getPurchaseOffer which

- Takes a single double representing the number of baskets of black walnuts brought by a seller (so for example, 3.25 represents three full baskets and an additional basket which is one-quarter full), and
- Returns the amount that WWW will pay for those baskets.

As a first step, write a main method with examples and expected payments.

5 Lecture 5

Boolean operators

- Conditions are not allowed just in if statements
- Just as there are types for numbers, there is a type for boolean values
  ```java
  final boolean flag = x<5;
  ```
- There are two boolean constants, true and false
- Just as there are operators for integers, there are operators for boolean values
  ```java
  ||  or, disjunction
  &&  and, conjunction
  !  not
  ```

Exercise 5.1. Write a class ParityChecker and static method isOdd which

- Takes one parameter of type int, and
- Returns a result of type boolean which is true exactly when the argument is odd.

Do not use any of the methods in the Math class for your method. As a first step, write a main method with examples and expected results.

Exercise 5.2. Write a class EvenSquares and two static methods isEvenAndSquare and isEvenOrSquare where

- Both methods take one parameter of type int and have a result of type boolean,
- isEvenAndSquare returns true when the argument is both an even number and a perfect square, and
- isEvenOrSquare returns true when the argument is either an even number or a perfect square.

You are free to use any of the methods in the Math class for this exercise. As a first step, write a main method with examples and expected results for both methods.

Exercise 5.3. Consider three sticks of length two inches, three inches and six inches. We could not form a triangle with these sticks, because one stick is longer than the other two put together. But if instead the sticks had lengths two inches, three inches and four inches, we could make a triangle from those sticks. Write a class TriangleLengthsChecker and static method isTrianglePossible which

- Takes three arguments of type int,
- Returns a result of type boolean which is true exactly when sticks of the three lengths could form a triangle.

If one of the lengths is zero or negative, your method should return false. As a first step, write a main method with examples and expected results.
Exercise 5.4. Write a class LeapYearChecker with a static method isLeapYear which determines whether a year is a leap year. The rules and exceptions for determining whether a year is a leap year are:

- Most years are {not} leap years
- Unless the year is divisible by 4, in which case it is {a} leap year
- Unless the year is also divisible by 100, in which case it is {not} a leap year
- Unless the year is also divisible by 400, in which case it is {a} leap year

Your isLeapYear method should take a single argument of type int representing the year being tested, and should return its answer as a boolean, with true denoting a leap year.

6 Lecture 6

PITFALL: Keep things simple
Did your leap year method end this way?

```java
} else if (year % 4 == 0) {
    return true;
} else {
    return false;
}
```

This ending is correct — but it is more complicated than it needs to be

- When year % 4 == 0 evaluates to true, the method returns true
- When year % 4 == 0 evaluates to false, the method returns false
- Simplify by simply returning year % 4 == 0 itself!

Because simpler methods are

- Easier to understand
- Easier to debug
- Easier to maintain

REVIEW: From the book
You’ve read about how Java allows us to describe the sixth Thing that a Computer Can Do: iteration

```java
for(int VARIABLE=START; CONTINUATION_CONDITION; CHANGE) {
    STATEMENT1;
    STATEMENT2;
    ...
    STATEMENTn;
}
```

Today we will look at some examples of using for-loops
How the for-loop works

General loop structure:

```java
for(int VARIABLE=START; CONTINUATION_CONDITION; CHANGE) {
    STATEMENT1;
    STATEMENT2;
    ...
    STATEMENTn;
}
```

- Steps Java takes:
  - int VARIABLE=START
  - Check CONTINUATION_CONDITION, maybe stop running the loop
  - Run STATEMENT1 through STATEMENTn
  - Apply the CHANGE
  - Check CONTINUATION_CONDITION, maybe stop running the loop
  - Run STATEMENT1 through STATEMENTn
  - Apply the CHANGE
  - Check CONTINUATION_CONDITION, maybe stop running the loop
  - Run STATEMENT1 through STATEMENTn
  - Apply the CHANGE
  - ... and so on until the CONTINUATION_CONDITION is falsified

Exercise 6.1. Write a class SimpleLoop whose main method prints the squares of the integers from 0 to 10.

Exercise 6.2. Write a class SentenceFixer with a static method printCapitalized which

  - Accepts a String parameter assumed to be a sentence,
  - Returns nothing, and
  - Prints that sentence making sure the first character is capitalized, and that subsequent characters are lower-case.

The standard methods toUpperCase and toLowerCase in class java.lang.Character will be helpful in converting characters to the correct case. As the usual first step, write a main method with examples and expected results. Step through your method by hand for the argument string HELLO! to be sure you understand you it works.

Exercise 6.3. The factorial function $n!$ is defined informally as $n! = n \cdot (n - 1) \cdot \ldots \cdot 2 \cdot 1$, and is defined formally by two rules:

  - If $n = 0$, then $n! = 1$
  - If $n > 0$, then $n! = n \cdot (n - 1)!$

Write a class FactorialFinder with a static method factorial which

  - Accepts a single int parameter
  - Returns a long result representing the factorial of the argument.

Since factorial is not defined on negative numbers, it does not matter what your method does for such input. As the usual first step, write a main method with examples and expected results.
Exercise 6.4. The choose function from probability is defined as
\[
{n \choose m} = \frac{n!}{m!(n-m)!}.
\]
Given the factorial method above, it is certainly possible to extend the FactorialFinder of Exercise 6.3 class with a method to implement choose directly:

```java
public static long nChooseM(final int n, final int m) {
    return factorial(n)/factorial(m)/factorial(n-m);
}
```

But this implementation is inefficient, and may cause overflow even when the final result actually can be represented as a `long`. Write a more efficient version of `nChooseM` which only performs the multiplications and divisions which are absolutely necessary. As usual, extend the `main` method with examples and expected results as a first step.

Exercise 6.5. The Fibonacci numbers are a sequence of integers indexed from 0 up, defined by:

- Fibonacci number 0 is 0.
- Fibonacci number 1 is 1.
- For any \( n > 1 \), Fibonacci number \( n \) is the sum of the two previous Fibonacci numbers (indexed \( n-1 \) and \( n-2 \)).

Write a class `FibonacciFinder` with a static method `fibonacci` which

- Accepts a single `int` parameter \( n \)
- Returns a `long` result representing Fibonacci number \( n \).

Since the series is not defined on negative numbers, it does not matter what your method does for such input. As the usual first step, write a `main` method with examples and expected results.

Exercise 6.6. Write a class `VowelCounter` with static method `getVowelCount` whose one argument is a `String` and which returns the number of characters in the string which are vowels (a, e, i, o and u).

7 Lecture 7

LOOK BACK: Two looping patterns

We’ve seen two different patterns for combining looping with other Thing a Computer Can Do

Processing each element

- Some action for each value
- But we do not link different values together
- So far, this has been printing

Accumulating a new value

- No action for values by themselves
- But combine them (or something about them) together
- Factorial, Fibonacci
- Typified by an accumulator variable
  - Declared before the loop
  - Changed within the loop
  - Used after the loop
REVIEW: Another kind of loop
In the book, you read about another, simpler kind of loop

while (CONTINUATION_CONDITION) {
    STATEMENT1;
    STATEMENT2;
    // ... 
    STATEMENTn;
}

• Steps Java takes:
  – Check CONTINUATION_CONDITION, maybe skip running the loop body at all
  – Run STATEMENT1 through STATEMENTn
  – Check CONTINUATION_CONDITION, maybe stop running the loop
  – Run STATEMENT1 through STATEMENTn
  – Check CONTINUATION_CONDITION, maybe stop running the loop
  – Run STATEMENT1 through STATEMENTn
  – ... and so on until the CONTINUATION_CONDITION is falsified

• If there are variables to set up, or changes to make, we must implement them as separate statements before the loop or in the loop body

Exercise 7.1. Write a class NumberSequence whose main method:

• First prompts for and reads an integer. This integer indicates the number of floating-point values which the program will subsequently attempt to read.

• Next, prompts for and reads the number of double values indicated by the initially-entered integer.

• Then, after reading in the floating-point values, computes and prints the sum and the average of the double values (not the integer).

Exercise 7.2. Write a class ChangeMaker with a static method getChange which

• Takes a single int argument representing a number of cents

• Returns a string describing the way to represent that amount in the fewest number of common US coins. The string should be of the form "XX quarters, YY dimes, ZZ nickels, WW pennies". When (for example), YY is zero, the string should use the singular dime instead of dimes.

As the usual first step, write a main method with examples and expected results.

Exercise 7.3. Write a class TransactionSequence whose main method:

• First prompts for and reads a floating-point value. This value indicates the initial balance of an account. The next two steps will detail how the program will read a sequence of transactions to that account.

• Second, prompts for and reads an integer. This integer indicates the number of transactions which the program will subsequently attempt to read.

• Next, prompts for and reads the number of double values indicated by the previously-entered integer, each of which represents the value of a transaction against the account.

• Then, after reading in the floating-point values, computes and prints the balance of the account.
How to choose — while or for?

- for loops are great when the loop ranges over one variable, and there is one change to that variable only between passes of the loop
- But otherwise it may be more natural to use a while loop
  - If there are multiple loop variables
  - If there are many changes from pass to pass

**Exercise 7.4.** A character in a string is a self-describing letter if its position in the string is the same as the letter’s position in the alphabet. For example, in the string "adc",

- a and c are self-describing letters, since a is the first letter in the alphabet and in the string, and c is the third letter in the alphabet and in the string.
- d is not self-describing, since it is the fourth letter of the alphabet but the second character of the string

Write a class SelfDescribers with a static method countSelfDescribing which

- Takes a single String method
- Returns the number of self-describing letters in the string

As the usual first step, write a main method with examples and expected results.

## 8 Lecture 8

**Exercise 8.1.** What do these programs do? Try to work out what it prints without running it before checking your prediction with Java.

1. `for (int i=1; i<=6; i++) {
   for (int j=1; j<=i; j++) {
      System.out.print(i);
   }
   System.out.println();
}

2. `for (int i=1; i<=6; i++) {
   for (int j=1; j<=(6-i); j++) {
      System.out.print("-");
   }
   for (int j=1; j<=i; j++) {
      System.out.print(i);
   }
   System.out.println();
}

What if we swap the two inner loops?

**Exercise 8.2.** Write a program to print this triangle:

```
0
01
012
0123
01234
012345
```
Find a solution which uses only a single outer and a single inner loop.

Exercise 8.3. Spot the errors in these loops:

- for (int i=1; i<=6; i++) {
  for (int j=1; i<=i; j++) {
    System.out.print(i);
  }
  System.out.println();
}

- for (int i=1; i<=6; i++) {
  for (int j=1; j<=i; i++) {
    System.out.print(i);
  }
  System.out.println();
}

Exercise 8.4. Write a class FramedSquare with static method printFramed which

- Takes two arguments frameSize and innerSize
- Draws a square made of asterixes and periods, where
  - The asterixes form a frame on the outer edge of the square with thickness frameSize, and
  - The inside of the frame is filled in with a innerSize-by-innerSize square of periods.

So printFramed(2, 7) would print

```
**********
**********
**........**
**........**
**........**
**........**
**........**
**........**
**********
**********
```

Commenting on loops

Document your loop with a high-level comment on the purpose of the loop, but don’t just repeat what the code already says.

- Good

  // Print a triangle of numbers, each row
  // repeating the digit one greater than the
  // row above.
for (int i=1; i<=6; i++) { // 1
    for (int j=1; j<=i; j++) { // 2
        System.out.print(i); // 3
    }
    System.out.println(); // 4
}

• Bad

// Loop i from 1 to 6, each time loop j from
// 1 to i, each time print i, and then in the
// outer loop start a new line.
for (int i=1; i<=6; i++) { // 1
    for (int j=1; j<=i; j++) { // 2
        System.out.print(i); // 3
    }
    System.out.println(); // 4
}

9 Lectures 9 and 10

Arrays

• So far we’ve studied
  – All of the Six Things a Program Can Do
  – One of the four ways Java helps us organize our work

• Now we move on to another way Java helps us organize our work
  1. Grouping sequences of operations together
  2. Grouping related data together
  3. Associating a group of data with operations relevant to that data
  4. Naming these groups and associations for easy and repeated use

• So far we’ve worked with scalars — single numbers or characters

• Now we’re going to look at data structures
  – (Many) items of the same type: an array
  – Items of (many) different type: classes and objects

Declaring and using arrays

Must declare an array variable just as we declare a numeric or string variable

• Append [] to a type to make it an array type
  – The size is not part of the type
    – int[]
    – String[]

• To write an array of values, put the values inside curly-braces, and separate them with commas
  – { 10, 20, 30, 40 }
• Refer to one element of an array by a numeric index
  – Write the number in square-brackets after the name of an array variable
    * int number = numbers[i];
    * numbers[i] = 400;
  – Index from 0
  – Indexing out of bounds will cause an error
• Get the length of an array with .length
  – Note that we do not use parentheses after length
  – It’s a property that we look up, not a method like sin that calculates something

**Exercise 9.1.** Trace through the execution of this class (without running it first). What does it print?

```java
public class UseAnArray {
    public static void main(String[] argv) {
        int[] numbers = { 10, 20, 30, 40 }; // 1
        for(int i=0; i<numbers.length; i++) { // 2
            final int number = numbers[i]; // 3
            System.out.println(number); // 4
        }
    }
}
```

**Exercise 9.2.** What is the largest Java array you can define? Is there a difference between the largest array that the Java compiler will allow you define, and the largest array that a running program on the system you happen to be using right now can allocate to a running program?

**Exercise 9.3.** What errors do you get when you:
  • Try to read from an array slot which is beyond the upper limit of an array?
  • Try to write to an array slot which is beyond the upper limit of an array?
  • Use round parentheses instead of square brackets when you declare an array?
  • Use round parentheses instead of square brackets when you access an array element?
  • Assign a String to an array which you have declared to hold int values?
Which of these errors are compile-time errors, and which are run-time errors?

**Exercise 9.4.** Write a class UseAStringArray whose main method
  • Declares a String array containing two values, "Hello" and "Goodbye", and then
  • Loops through the array to print each of the values.

**Exercise 9.5.** Update your class MonthNamer from Exercise 4.3 to use an array within getMonthName.

**Exercise 9.6.** Write a class ColumnMaker with a static method printInColumn which
  • Takes an array of integers as its single argument, and
  • Prints the numbers right-aligned in a single column.
Use your getCharacterLength method from Exercise 3.9.
Exercise 9.7. Write a class StatsFinder with a static method printSummaryStats which takes an array of double values as its single argument, and calculates and prints messages detailing

• How many numbers there are.
• Their mean: the sum of the values divided by how many values there are.
• Their median: the value of the middlemost entry of the array.
• Their standard deviation: $\sigma = \sqrt{\frac{\sum (\bar{x} - x_i)^2}{N-1}}$, where $\bar{x}$ is the mean and $N$ is the number of values.

Exercise 9.8. Add another static method getSummaryStats to class StatsFinder from Exercise 9.7. Your getSummaryStats should, instead of just printing the various statistics, returns a new double[] array where element 0 is the mean, element 1 is the standard deviation, and so on. As the usual first step, create several tests as the main method of StatsFinder. Be sure to examine the length of the array as well as each element. Rewrite printSummaryStats to remove duplicated code, so that it just calls getSummaryStats and prints its results in a comprehensible manner.

Exercise 9.9. Modify StatsFinder from Exercise 9.7 or 9.8 to work with integers, and to additionally calculate:

• The maximum and minimum values of the array.
• The number of different values in the array, and how many times each one appears in the array.
• Their mode, the value which appears more often than any other.
• Their median, the value is less than (or equal to) half of the other values, and greater than (or equal to) the other half.

Exercise 9.10. Trace through the execution of this class (without running it first). What does it print?

public class ChangeAnArray {
    public static void main(String[] argv) {
        int[] numbers = { 10, 20, 30, 40 };
        for(int i=0; i<numbers.length; i++) {
            numbers[i] *= 2;
            System.out.println(numbers[i]);
        }
    }
}

Allocating arrays without assigning to them

We can allocate space for the array without initializing its entries

• Use the new keyword, plus the array type
  – But to allocate space (as opposed to just giving the type), we must give a size
  – String[] names = new String[10];

• Fill in the elements one-by-one

• Arrays have initial values when we create them
  – For numeric types, zero
  – For boolean, false
  – For String, the special value null

22
Exercise 9.11. Trace through the execution of this class (without running it first). What does it print?

```java
public class UseAnArray {
    public static void main(String[] argv) {
        final int[] numbers = new int[4]; // 1
        number[0] = 1; // 2
        number[1] = 2; // 3
        number[2] = 3; // 4
        number[3] = 4; // 5

        for(int i=0; i<numbers.length; i++) { // 6
            final int number = numbers[i]; // 7
            System.out.println(number); // 8
        }
    }
}
```

Exercise 9.12. Consider the following class:

```java
public class OddsEvens {
    public static void main(String[] args) {
        char[] characters = new char[50];

        // Your code goes here

        for(int i=0; i<characters.length; i++) {
            System.out.println(i + " " + characters[i]);
        }
    }
}
```

In the indicated spot, add code which will initialize the array slots with an even-valued index (characters[0], characters[2], and so on) with 'E', and the array slots with an odd-valued index (characters[1], characters[3], and so on) with 'O'.

Exercise 9.13. The Sieve of Eratosthenes is a classical technique for finding prime numbers. The idea of the Sieve is that we can write down the numbers which we are interested in testing for primality, from 2 up to the largest number of interest. We will either circle or scratch out each number in the sequence according to the following loop:

- While there is a number which is neither circled nor scratched out
  - Circle the smallest such number
  - Scratch out every multiple (for integer factors greater than 1) of the number we just circled

After running the loop, the prime numbers are exactly those which are circled. Write a class PrimesFinder with static method printPrimes which

- Takes one argument n of type int, and
- Uses the Sieve of Eratosthenes to print the prime numbers less than or equal to n.

Multiple arrays
We often use multiple arrays at the same time
- For operations which extract a result from both
- When elements at the same index are related
Exercise 9.14. The **dot product** is a common mathematical operation on pairs of numeric vectors (arrays) of the same size. Given two vectors \(x_i\) and \(y_i\), their dot product \(\vec{x} \cdot \vec{y} = \sum_{i} x_i y_i\). Write a class **DotProduct** with static method **getDotProduct** which

- Takes two arguments of type **double[]**, and
- Returns the double dot product of the two arrays.

It does not matter what your program does if the arrays are not the same size. As the usual first step, write a **main** method with examples and expected results.

**Changing to a new array**

```java
int[] numbers = { 1, 2, 3, 4 };
for(int i=0; i<numbers.length; i++) {
    int number = numbers[i];
    System.out.println(number);
}

numbers = new int[] { 5, 6, 7 }; // The new int[] part is implied when we're declaring the array variable.
for(int i=0; i<numbers.length; i++) {
    int number = numbers[i];
    System.out.println(number);
}
```

- But we can’t change the length with something like

  ```java
  numbers.length = 2;
  ```

10 Lectures 11 and 14

**Mutating scalar parameters**

With scalar parameters, changes to the formal parameter do not escape the method

- Each method has its own workspace of storage locations for variables and local parameters
- No method call can change another method call’s storage locations

**Exercise 10.1.** What do these programs do? Trace through the programs by hand before running them in Eclipse to confirm your hypotheses.

```java
public static void main(String[] args) {
    int y = 30;
    f(y);
    System.out.println(y);
}

public static void f(int x) {
    System.out.println(x);
    x = 10;
    System.out.println(x);
}
```
• public static void main(String[] args) {
  int x=30;
  f(x);
  System.out.println(x);
}

public static void f(int x) {
  System.out.println(x);
  x=10;
  System.out.println(x);
}

Passing arrays to methods
But the internals of arrays are not duplicated when we pass them to a method
• A call to new creates space separate from the local storage of parameters and variables
  – The reference is local, but the space itself is separate
  – Even if we write Line 2 as
    int[] y = { 10, 20, 30 };
    The call to new is still implicit

Exercise 10.2. What do these programs do? Trace through each one by hand before running them in Eclipse to confirm your hypotheses.
• public static void main(String[] args) {
  int[] y = new int[] { 10, 20, 30 };
  f(y);
  for(int i=0; i<y.length; ++i) {
    System.out.println(y[i]);
  }
}

public static void f(int[] x) {
  if (x.length > 1) {
    x[1] = -1;
  }
  return;
}
• public static void main(String[] args) {
  int[] y = new int[] { 10, 20, 30 };
  f(y);
  System.out.println("="");
  for(int i=0; i<y.length; ++i) {
    System.out.println(y[i]);
  }
}

public static void f(int[] x) {
  x = new int[] { 1, 2, 3, 4, 5 };
  x[1] = -1;
  for(int i=0; i<x.length; ++i) {
    System.out.println(x[i]);
  }
}
public static void main(String[] args) {
    int[] y = new int[] { 10, 20, 30 };  
f(y); 
    System.out.println("========"); 
    for(int i=0; i<y.length; ++i) { 
        System.out.println(y[i]); 
    } 
}

public static void f(int[] x) { 
    for(int i=0; i<x.length; ++i) { 
        x[i] = i*100; 
    } 
    for(int i=0; i<x.length; ++i) { 
        System.out.println(x[i]); 
    } 
    x = new int[] { 1, 2, 3, 4, 5 }; 
    if (x.length > 1) { 
        x[1] = -1; 
    } 
    System.out.println("========"); 
    for(int i=0; i<x.length; ++i) { 
        System.out.println(x[i]); 
    } 
    return; 
}

Exercise 10.3. Write a class ScaleBy with method scaleByFactor which

• Takes two arguments:
  1. An array numbers of double
  2. Another double value called factor

• Multiplies every element of numbers by factor

• Has no explicit return value

Include a main routine with several tests of scaleByFactor

Exercise 10.4. Write a class CapitalizeAllChars with method upcaseAll which

• Takes two arguments:
  1. An array characters of char

• Capitalizes every element of characters (use the library method Character.toUpperCase)

• Has no explicit return value

Include a main routine with several tests of upcaseAll
Exercise 10.5. Write a class TwoGrouper with method sortIntoGroups which

- Takes a single argument, an array of characters, which your method should assume contains only 'R' and 'W', and
- Reorganizes the array so that all of the 'R's come before all of the 'W's.

*Important restriction:* your program may "visit" each element of the list only once. So for example, a program which simply goes through the array once to count the 'R's, and which then makes a second pass to assign 'R's and 'W's, would not satisfy this restriction!

(For a hint, see p. 62)

Exercise 10.6. (Continues from Exercise [10.5]) Write a class ThreeGrouper with method sortIntoGroups which

- Takes a single argument, an array of characters, which your method should assume contains only 'R', 'W' and 'B', and
- Reorganizes the array so that all of the 'R's come before all of the 'W's, all of the 'W's come before all of the 'B's.

Follow the same restriction as for Exercise [10.5] visit each node only once.

This problem was invented by Edsger W. Dijkstra, who discovered many important computer science algorithms, and who originated in the Netherlands. The Dutch flag consists of three stripes, one red, one white, and one blue, and Dijkstra named this problem *The Dutch National Flag problem*.

Exercise 10.7. Write a class ArrayIntFinder with a static method getIndexOf which

- Takes two arguments
  1. An array numbers of integers, and
  2. A single integer target
- Searches numbers for the index where target is found
- Returns either that index, or -1 if target was not found in numbers

As usual, begin by writing a main method which tests getIndexOf on a variety of examples. Make sure your examples include both cases where target is found in numbers, and cases where target is not found in numbers.

Exercise 10.8. Write a class SortedArrayIntFinder with a static method getIndexOf which whose arguments and result are just as in class ArrayIntFinder of Exercise [10.7] but where your method is allowed to assume that the array is sorted in order. Take advantage of the assumption of an ordered list by starting your search in the middle of the list, so that after every comparison your method can exclude half of the unsearched elements based on the target being greater or less than the middle element.

(For a hint, see p. 62)

Methods can return new arrays

- The return type of a method can be an array type

  public static int[] buildIntegerArray(...)

- The return value can be passed in from outside the array, or created within the array
public static int[] buildIntegerArray(int a, int b) {
    return new int[] { a, b };
}

public static int[] pickEvenLength(int[] a, int[] b) {
    if (a.length % 2 == 0) {
        return a;
    }
    if (b.length % 2 == 0) {
        return b;
    }
    return new int[] { 0, 1 };
}

- Return values created inside the array can be filled by code, or initialized with the declaration

Exercise 10.9. Write a class ArraySplitter with a static method getUpperHalf which
- Takes a single argument, an array of int values
- Return an array which
  - Is half the length of the argument array
  - Contains the same values as the elements of the argument from the midway point to the highest index.

As usual, begin with a main method containing tests which verify the behavior of getUpperHalf.

Exercise 10.10. Write a class ArrayInterleaver with a static method interleaveArrays which
- Takes two arguments, each an array of String values
- Returns a new array
  - Whose length is the sum of the two argument arrays
  - And which contains the elements of the arrays, drawing in alternation from one array and then the other.
  - If one argument array is longer than the other, then its additional elements should appear together at the end of the result.

So for arguments containing
"A", "B", "C"

and
"V", "W", "X", "Y", "Z"

the result should contain

As usual, begin with a main method containing tests which verify the behavior of interleaveArrays.

Exercise 10.11. Write a class BiggestElements with static method getBiggest which
- Takes two arguments
  1. An array of integers numbers
  2. An additional integer howMany
- If howMany is less than the length of numbers, then getBiggest should return a new array of length howMany containing the biggest values numbers.
  - If howMany is greater than or equal to the length of numbers, then it should return numbers as-is.

As usual, begin with a main method containing tests which verify the behavior of getBiggest.
Exercise 10.12. Write a class \texttt{WordsTaker} with a static method \texttt{getWords} which

- Takes two arguments,
  1. A \texttt{String}, expected to consist of several space-separated words
  2. An \texttt{int}, representing how many of these words are of interest
- Returns an array of strings
  - The length of the array should be the same as the integer argument
  - The first (index 0) element of the result array should be the first space-separated word of the string argument, and so on

Assume for this exercise that there will always be enough words in the string for the integer argument. As the usual first step, create several tests of \texttt{getWords} for the \texttt{main} method of \texttt{WordsTaker}. Be sure to examine the length of the array as well as each element.

11 Lecture 12

What does this method do?

```java
public static void MYSTERY(int[] numbers) {
    int howMany = numbers.length;
    for(int a=1; a<howMany; a++) {
        for(int b=howMany-1; a<=b; b--) {
            if (numbers[b-1] > numbers[b]) {
                final int tmp = numbers[b-1];
                numbers[b-1] = numbers[b];
                numbers[b] = tmp;
            }
        }
    }
}

public static void main(String[] args) {
    final int[] numbers = new int[] {
        10, 7, 3, 12, 18, 2, 8, 5, 20, 15
    };
    MYSTERY(numbers);
    for(int i=0; i<howMany; i++) { System.out.print(numbers[i] + " "); } 
    System.out.println();
}
```

The \texttt{main} method is straightforward

- We start by creating an array

  ```java
  final int[] numbers = new int[] {
      10, 7, 3, 12, 18, 2, 8, 5, 20, 15
  };
  ```

- We end by printing its contents
for(int i=0; i<howMany; i++) {
    System.out.print(numbers[i] + " ");
}
System.out.println();

• **MYSTERY** seems to change the contents, so the numbers we print may not be the same as the numbers we first put into the array

**But what about MYSTERY?**

```java
int howMany = numbers.length; // 0
for(int a=1; a<howMany; a++) { // 1
    for(int b=howMany-1; a<=b; b--) { // 2
        if (numbers[b-1] > numbers[b]) { // 3
            final int tmp = numbers[b-1]; // 4
            numbers[b-1] = numbers[b]; // 5
            numbers[b] = tmp; // 6
        }
    }
}
```

• Look at just the if-statement at Line 3
  – If two consecutive elements (b-1 and b) have a larger value first, it will swap them.

• The inner loop starts at the end of the array, and does this possible swapping from right to left
  – So at the end of the inner loop, the lowest value from position a to the end of the array will be pushed into position a.

• The outer loop performs this pushing first to position 0, then to position 1, and so on up to the next-to-last position in the array.
  – So the smallest value ends up in position 0, the next smallest in position 1, and so on.
  – These loops sort the array.

**Stepping through bubble sort**

```java
public static void bubbleSort(int[] numbers) {
    int howMany = numbers.length;
    for(int a=1; a<howMany; a++) { // 1
        for(int b=howMany-1; a<=b; b--) { // 2
            if (numbers[b-1] > numbers[b]) { // 3
                final int tmp = numbers[b-1]; // 4
                numbers[b-1] = numbers[b]; // 5
                numbers[b] = tmp; // 6
            }
        }
    }
}
```

• What happens on a call

```java
int[] fourNums = new int[] { 5, 20, 13, 2 };
bubbleSort(fourNums);
```
Exercise 11.1. What would happen in the case of applying bubble sort to this algorithm:

```java
int[] manyNums = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                           11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
                           21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 0};
bubbleSort(manyNums);
```

What happens the second time we run the body of the outer loop? The third? The thirtieth? How can we improve `bubbleSort` based on this observation?

What does bubble sort cost?

- How many times will we execute the comparison between elements (and possibly swap them) for an array of length \( n \)?
  - The first time through the inner loop, it’s \( n - 1 \) times; then \( n - 2 \), and so on down to 1.
  - So in total, it’s \( \sum_{i=1}^{n-1} i = \frac{n^2-n}{2} \).
    - The constant factor \( \frac{1}{2} \) isn’t an interesting detail — for input of size \( n \), the number of steps is on the order of \( n^2 \).
    - Even subtracting \( n \) does not have a big impact, once \( n \) starts to get big.
    - The \( n^2 \) growth is what’s interesting to us — the growth is proportionate to \( n^2 \).
- Bubble sort is fine for smaller arrays, but for larger arrays gets too slow.
  - The best sorting algorithms run in \( O(n \log n) \) time — we’ll look at one of these later this semester if time allows.

Exercise 11.2. Bubble sort is not the only possible sorting algorithm. Selection sort implements the following idea:

- First, go through every element of the list, and find the index of the smallest value. If that value is not at slot 0, swap it with the value at slot 0.
- Next, go through every element of the list except the first, and find the index of the smallest value among them. If that value is not at slot 1, swap it with the value at slot 1.
- And so on, for each slot in the list.

Write a class `SelectionSorter` with a method `sort` which

- Takes a list of integers as its input,
- Performs a selection sort on the list, and
- Returns nothing.

As the usual first step, create several tests of `sort` for the `main` method of `SelectionSorter`.

Exercise 11.3. For each of the following arrays, how many swaps will selection sort make for each? How many will bubble sort make?

- \( \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \)
- \( \{10, 9, 8, 7, 6, 5, 4, 3, 2, 1\} \)
- \( \{20, 40, 60, 80, 100, 120, 140, 160, 0\} \)
- \( \{160, 0, 20, 40, 60, 80, 100, 120, 140\} \)
- \( \{120, 140, 160, 180, 200, 20, 40, 60, 80, 100\} \)
- \( \{200, 180, 160, 140, 120, 100, 80, 60, 40, 20\} \)
12 Lecture 13

12.1 Two-dimensional arrays

Two-dimensional arrays

- So far we’ve used arrays with a single index — called one-dimensional.
- But we can have any number of indices in a matrix:

```java
final String[][] phrases = {
    { "Hello", "Let’s eat", "See you later" },
    { "Bonjour", "Bon appetit", "Au revoir" },
    { "Guten Tag", "Mahlzeit", "Tschau" }
};
```

- When we traverse this matrix, we can use the known length of each inner array

Uneven arrays

- Two dimensional arrays do not have to be rectangular
- Each row can span a different number of columns

```java
final String[][] wordsWeKnow = {
    { "hello", "let", "us", "eat",
      "goodbye", "see", "you", "later" },
    { "bonjour", "bon", "appetit", "au", "revoir" },
    { "guten", "tag", "mahlzeit", "tschau" }
};
```

- Some true expressions:

```java
wordsWeKnow.length == 3
wordsWeKnow[0].length == 8
wordsWeKnow[1].length == 5
wordsWeKnow[2].length == 4
```

- Must check the length of each inner array when traversing

Exercise 12.1. A magic square is an \(n\)-by-\(n\) array of integers where

- The values 1 through \(n^2\) each occur exactly once in the array
- The values of each row sum to \(n\)
- The values of each column sum to \(n\)
- The values of each diagonal sum to \(n\)

Write a class MagicSquareChacker with a method `isMagicSquare` which

- Takes a single argument, an array of arrays of integers
- Returns true exactly when
  - The array is a square — the top-level array-of-arrays and each contained array all have the same length, and
  - The array forms a magic square

As usual, first write a main method with ample tests of `isMagicSquare`. 
Exercise 12.2. Consider the square below:

\[
\begin{array}{c}
\text{R} & \text{A} & \text{D} & \text{A} & \text{R} \\
\text{A} & \text{D} & \text{A} & \text{R} \\
\text{D} & \text{A} & \text{R} & \text{A} \\
\text{A} & \text{R} & \text{A} & \text{D} \\
\text{R} & \text{A} & \text{D} & \text{A}
\end{array}
\]

There are many different paths we could follow starting from a square in this array, where the letters in the path spell RADAR. Some examples:

\[
\begin{array}{c}
\text{R} & \text{R} & \text{A} & \text{D} & \text{A} \\
\text{A} & \text{R} \\
\text{D} & \text{A} & \text{R} & \text{A} \\
\text{R} & \text{A} & \text{D} & \text{A} & \text{R}
\end{array}
\]

Write a class RadarFinder with method getRadarCount which

- Takes an array of arrays of char, and
- Returns the number of paths through neighboring squares which spell RADAR

Decide for yourself (and document) how you will treat non-square arrays, and diagonal paths. As usual, first write a main method with ample tests of getRadarCount.

12.2 Lining up columns of information

Printing the wordsWeKnow matrix

```
final String[][] wordsWeKnow = {
    { "hello", "let", "us", "eat",
      "goodbye", "see", "you", "later" },
    { "bonjour", "bon", "appetit", "au", "revoir" },
    { "guten", "tag", "mahlzeit", "tschau" }
};
```

- We discussed, but did not implement, having the columns line up
- A starting point, for this implementation

```
for(int i=0; i<wordsWeKnow.length; i++) {
    for(int j=0; j<wordsWeKnow[i].length; j++) {
        System.out.print(wordsWeKnow[i][j] + " ");
    }
    System.out.println();
}
```

What do we need to think about?

What’s involved with lining up the columns?

- Every column the same width, or each column padded separately?
  - We’ll do both, first every column the same
- Need to look at the width of every word before we know the width of any column
- Since some rows have more columns than others, we need to look at all of the rows to know how many columns there are
- Must add the right amount of spaces to pad each string
Every column the same width

```java
int width=0;
for(int j=0; j<wordsWeKnow.length; j++) {
    String[] row = wordsWeKnow[j];
    for(int k=0; k<row.length; k++) {
        String word = row[k];
        final int thisWidth = word.length();
        if (thisWidth>width) {
            width=thisWidth;
        }
    }
}
for(int j=0; j<wordsWeKnow.length; j++) {
    String[] row = wordsWeKnow[j];
    for(int k=0; k<row.length; k++) {
        String word = row[k];
        final int pad = width - word.length();
        for(int i=0; i<pad; i++) {
            System.out.print(" ");
        }
        System.out.print(word);
        System.out.print(" ");
    }
    System.out.println();
}
```

Every column the minimum width

• What’s our algorithm here?

Every column the minimum width

```java
int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }
}
int[] widths = new int[cols];
for (int row=0; row<wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col=0; col<rowArray.length; col++) {
        final int thisWidth = rowArray[col].length();
        if (widths[col] < thisWidth) {
            widths[col] = thisWidth;
        }
    }
}
for (int row=0; row<wordsWeKnow.length; row++) {
```
13 Lectures 15-17

Classes and objects
A class is a description — a type of thing. An object is one instance of a class
• String and Scanner are two classes we’ve seen already
• "Hello" and "Goodbye" are two different String objects
  – There’s a special syntax for creating a String
• Ordinarily we create a new object of a class using new

    Scanner scn = new Scanner(System.in);

Defining a class
Name
• Normally the name of a class is the same as the name of the file where it is defined

Fields - storage for simple values
• The state of each object of the class

Methods - code to be executed
• We’ve already written static methods
  – Separate from objects of the class
• Object methods are linked to the context of one object
  – Like how charAt refers to the characters of a particular string
A small class

```java
public class SmallClass {
    private final String label;

    public SmallClass(final String label) {
        this.label = label;
    }

    public String getLabel() {
        return label;
    }

    public boolean longerLabelThan(final SmallClass thatLabel) {
        return this.label.length() > thatLabel.label.length();
    }

    public static void main(String[] args) {
        final SmallClass
            sc1 = new SmallClass("alpha"),
            sc2 = new SmallClass("beta"),
            sc3 = new SmallClass("gamma");

        System.out.println("sc1 label: "+ sc1.label);
        System.out.println("sc3 > sc2: "+ sc3.longerLabelThan(sc2));
    }
}
```

- **Fields** store information in one instance
  - Fields which never change can receive the keyword `final`
  - We can (and should) use `final` with method local variables as well

- **Constructor**
  - Looks a little like a method
  - Called once for each instance upon `new`, when setting it up

- **Methods** — without the `static` keyword!
  - Associated with one particular instance
  - Values in the field may be particular to each instance
  - **Accessor methods** just return a field
    - Usually named with `get` before the field name
    - `private` fields and methods can referenced from `within` the class body only — not from within other classes
  - **Mutator methods** change fields
  - (Other methods, which are neither accessors nor mutators, are just "methods".)

• **Fields** store information in one instance
  - Fields which never change can receive the keyword `final`
  - We can (and should) use `final` with method local variables as well

• **Constructor**
  - Looks a little like a method
  - Called once for each instance upon `new`, when setting it up

• **Methods** — without the `static` keyword!
  - Associated with one particular instance
  - Values in the field may be particular to each instance
  - **Accessor methods** just return a field
    - Usually named with `get` before the field name
    - `private` fields and methods can referenced from `within` the class body only — not from within other classes
  - **Mutator methods** change fields
  - (Other methods, which are neither accessors nor mutators, are just "methods".)
• One class can have both static and object methods
  – main is just another static method
  – But only classes with a main method can be run

Exercise 13.1. Create a class Student to store the following information about a student:
  • Their name
  • Their ID
  • Their current classes
  • The number of credits earned

Include a constructor which receives these values, and accessor methods for each.

Exercise 13.2. Extend class Student of Exercise 13.1 with the assumption that a student ID will never change.

Exercise 13.3. Extend class Student of Exercise 13.2 with methods to update a student record:
  • Update the name
  • Report passed classes
  • Enroll in new classes

Exercise 13.4. Create a class Vehicle with the following private field:
  • A single String field makeModel for the make/model
  • An int field fuelCap for the capacity of the fuel tank
  • An double field mpg for the miles-per-gallon efficiency

Define a constructor which receives and assigns the field values in the above order, and accessor methods for each. None of the fields should be mutable.

Exercise 13.5. Add a method getRange to class Vehicle of Exercise 13.4 which calculates from the the fuelCap and mpg fields the maximum distance which the vehicle could drive on a single tank. As the usual first step, add tests of getRange to a main method.

Exercise 13.6. Revise class Vehicle of Exercise 13.5 with an additional private fields:
  • An int field mileage for the number of miles recorded by the vehicle

Modify the constructor to initialize this field as well, and add both an accessor and a mutator method to set the field. As the usual first step, add tests of using the setter to a main method.
Exercise 13.7. Write a class QuizScores to hold information about a student’s quiz scores in a class (obviously not this class, since we do not have quizzes). Your class should have the following constructor and methods:

- The constructor should take an array of integers, holding the quiz scores.
- Methods getHighest and getLowest should return the highest and lowest scores received. They take no arguments, and return an int.
- Method getAverage should return a double value, the average score received.
- Method countGreaterThan takes an integer and returns the number of quiz scores which are greater than the argument.
- Method getAverageDroppingLowest should return a double value, the average score excluding the lowest.

As the usual first step, add tests of these methods to a main method.

Exercise 13.8. Pick a board or paper game, and write a class whose object represent the state of play of that game. Add methods which update the state in response to a player move, and write tests for the effects which are not simple (such as displacing an opponent's piece).

Exercise 13.9. Write a class CharChecker whose constructor takes a single string (which we will refer to as base), and with the following methods:

- An accessor getBase for base
- A method countMatches which take a single string str, and which returns the number of characters of str which are also in base.

For example, the statements

CharChecker cc = new CharChecker("abc");
System.out.println(cc.countMatches("abracadabra"));

should print 8.

As the usual first step, add tests of these methods to a main method. Do not use any methods of String except charAt and length to solve this problem.

Exercises 13.10 through 13.17 will remind you of Exercises 9.7 through 9.9. Review your StateFinder implementations before starting these exercises, and expect to deploy some of the same algorithms here.

Exercise 13.10. Write a class DataSet to perform statistical analysis of a set of numbers.

- It should have a constructor which takes a single argument, an array of double values.
- Method mean should take no arguments, and should return the mean of the values from the array.
- Its field for storing data should be private.

As the usual first step, add tests of its method to a main method.
Exercise 13.11. Consider your class `DataSet` from Exercise 13.10. What happens if we run

```java
int[] numbers = new int[] { 5.0, 6.0, 7.0 };
DataSet ds = new DataSet(numbers);
numbers[1] = 12.0;
System.out.println(ds.mean());
```

With the usual setup for fields

```java
private double[] numbers;
public DataSet(double[] numbers) {
  this.numbers = numbers;
}
```

The `mean` would return 8.0, not 6.0 — the array itself is shared — we have allocated only one of them, and passed around references to it. Update the `DataSet` constructor to make sure that the storage we use for the numbers in a `DataSet` instance is separated from the array passed to the constructor. (See p. 20 for a hint if you need.)

Exercise 13.12. Extend the class `DataSet` of Exercise 13.10 with two methods:

- `addData`, which takes an array `additionalData` of double values. These values should be added to the values already in the `DataSet`, and should cause `mean` to return an updated, new value based on all of the data.
- `clearData`, which resets the data storage to contain no numbers.

Add tests of the new methods and the changes they make to subsequent calls in the `main` method.

Exercise 13.13. Extend class `DataSet` of the previous exercises to give the data sets a name.

- Add an additional constructor that takes both the `String` name and a `double` array.
- Provide accessor and mutator methods `getName` and `setName`.
- But keep the original constructor as well! For a `DataSet` instance created without a name, the `getName` accessor should return the string "(unnamed)".

Exercise 13.14. Add a method `stdDev` to `DataSet` from Exercise 13.12 which takes no arguments and returns the standard deviation of the data. (See Exercise 9.7 for a formula for standard deviation.) Since we use the mean for calculating the standard deviation, `stdDev` will need to call `mean`.

Exercise 13.15. Consider your class `DataSet` from Exercise 13.12 (without the `stdDev` method of Exercise 13.14). We have made `mean` a method, but when the data do not change, the mean will not change either. Update the class and methods so that we recalculate the mean value only when we need to, and just save the value when the data is unchanged. (See p. 20 for a hint if you need.)

Exercise 13.16. There are two ways to add caching of derived values to a version of `DataSet` with more than one derived value. Consider the version of `DataSet` from Exercise 13.14, with both `mean` and `stdDev`: extend this class to add two pairs of fields, one for `mean` and one for `stdDev`, in the manner of Exercise 13.15.

Exercise 13.17. Another way to add caching of derived values is to use a `single boolean` flag to indicate new data, but still multiple `double` fields for the cached values, plus a `private` helper method `updateCached` which recomputes all of the cached values. Extend `DataSet` of Exercise 13.14 to implement this idea, making sure that `mean` and `stdDev` call it only when there is new data.
Methods can create and return objects

Just as methods can create and return arrays, methods can create and return objects

```java
public class SmallClass {
    private final String label;

    // ... constructors, accessors ...

    public SmallClass addSublabel(String sublabel) {
        return new SmallClass(label + "." + sublabel);
    }
}
```

Exercise 13.18. Write a class `Complex` to model complex numbers

- The constructor should take two `double` values, for the real and imaginary parts
- Numbers are constants, so the fields should be `final`
- Provide accessors `getReal` and `getImaginary`
- Provide method `magnitude`, returning a `double`
- Provide method `conjugate`, which needs no arguments, and returns another `Complex`
- Provide methods `add`, `subtract`, `multiply` and `divide`, each taking a single `Complex` argument and returning a `Complex` result

The formulas for these operations are on p. 62

Exercise 13.19. Update Exercise 13.8 so that the board representations are immutable, and the methods in response to a player move return a new object representing the new state.

Exercise 13.20. In Exercise 13.12 we added a method `addData` which changed the underlying data, so that `mean` would normally return a different value. For this exercise we will again extend the `DataSet` Exercise 13.10 to add data, but in a way that does not change the underlying data set:

- Add a method `addData` which takes an array of `double` values, and returns a new `DataSet`. The resulting instance should contain all of the values of the original instance, plus all values from the argument array. The original instance should not be changed in any way.
- Add a method `removeData` which takes an array of `double` values, and which also returns a new `DataSet`. The resulting instance should contain all of the values of the original instance except the values from the argument array. Again, the original instance should not be changed in any way.

As usual, begin by adding tests of the new methods to `main`.

Notes on writing classes

- Stay organized! Write:
  - Fields first
  - Then the constructor
  - Then accessors
  - Then mutators
  - Then static methods
- Label fields `private`
– Except for global constants like pi, this is a nearly-universal style in Java
– Label fields and variables `final` whenever they do not change

• Two kinds of documentation:

  – Javadoc comments
    * Produce the same kind of HTML pages that we’ve looked at for `String`, `Math`, etc.
    * Start with `/**`, end with `*/`
    * Describe the inputs and outputs of each method, and its overall purpose
    * Use the Javadoc tool to generate reference pages
  – Algorithm descriptions
    * Part of the code only, not the Javadoc pages
    * Start with `//`
    * Describe how the statements of the method achieve the method’s overall goals

### 14 Lecture 18

**Interfaces**
Interfaces describe methods, but do not implement them

• Traditional use of methods:
  – Only declare methods, but give no method bodies
    * Exceptions in more recent Java systems, but we do not consider these now
  – No fields
  – No constructors

• Classes can declare themselves to implement an interface (or multiple interfaces) but must provide full implementations of the methods

**An interface for doing something**
A small interface example:

```java
public interface DoIt {
    public void doSomething(int i, double x);
    public int doSomethingElse(String s);
}
```

And a class which implements that interface:

```java
public class PrintingNumbersMeasuringStrings implements DoIt {
    public void doSomething(int i, double x) {
        System.out.println(i*i+x);
    }
    public int doSomethingElse(String s) {
        return s.length();
    }
}
```


Implementing two interfaces

Another simple interface:

```java
public interface DoMoreover {
    public long doAThirdThing(double x);
}
```

A class can declare that it implements both interfaces

```java
public class VeryAble implements DoIt, DoMoreover {
    public void doSomething(int i, double x) {
        System.out.println(i*i+x);
    }
    public int doSomethingElse(String s) {
        return s.length();
    }
    public long doAThirdThing(double x) {
        return Math.round(x*x);
    }
}
```

Why use interfaces?

- More than one class can provide the same interface

```java
public class AlsoAble implements DoIt, DoMoreover {
    public void doSomething(int i, double x) {
        System.out.println("i is " + i);
        System.out.println("x is " + x);
    }
    public int doSomethingElse(String s) { return 3; }
    public long doAThirdThing(double x) {
        return Math.round(x)+2;
    }
}
```

- Method can give an interface as an argument type
  - Any class implementing the interface is accepted
  - Only interface methods may be used on that argument

```java
public void workWith(DoIt d) {
    d.doSomething(1, 3.4);
    d.doSomething(20, 0.004);
    d.doSomethingElse("Hello");
}
```

Exercise 14.1. There is another common way to represent complex numbers. We can imagine the complex numbers as points on a plane, with the real component for the x-axis, and the imaginary component as the y-axis. But we can also represent points on the plane by two other numbers: their distance from the origin, and the measure of the angle formed by a line from the point to the origin, and the positive part of the x-axis.

So for example, the point (1,1) is at a distance of $\sqrt{2}$ from the origin. It forms a $45^\circ$-angle with the positive part of the x-axis, but we use radian measure instead of degrees, so we would measure the angle as $\frac{\pi}{4}$. In terms of complex numbers, we write: $1 + i = \sqrt{2}e^{\frac{\pi}{4}i}$.

Refactor the class Complex of Exercise 13.18 to have the name CartesianComplex, and instead define Complex to be this interface:
public interface Complex {
    public double getReal();
    public double getImaginary();
    public boolean isZero();
    public double getMagnitude();
    public double getAngle();
    public Complex add(Complex that);
    public Complex subtract(Complex that);
    public Complex multiply(Complex that);
    public Complex dividedBy(Complex that);
}

Then create a class PolarCartesian which also implements Complex. Be sure you use accessor methods to access the components of a number passed as an argument to the various operations — this way it makes sense to add two numbers which happen to have different representations.

As the usual first step, write tests making sure that both CartesianComplex and PolarComplex are correct.

15 Lecture 19

15.1 A class for representing people

Revisiting how we represent people

public class Person {
    private final String firstNames;
    private final String lastName;
    public Person(final String firstNames,
                   final String lastName) {
        this.firstNames = firstNames;
        this.lastName = lastName;
    }
    public String getFirstNames() { return firstNames; }
    public String geLastName() { return lastName; }
    public String fullName() {
        return firstNames + " " + lastName;
    }
}

What if some of the people we are representing are students?

• We’d also want to store the classes they’re taking
• But this would be true only of students, so not everyone should have this information
• One way of modeling this is through a subclass

public class Student extends Person {
    private final int[] courseIds;
    public Student(final String firstNames, 
                   final String lastName, 
                   final int[] courseIds) {
        super(firstNames, lastName);
        this.courseIds = courseIds;
    }
    public int[] getCourseIds() { return courseIds; }
}
Instantiating classes and subclasses

final Person person
    = new Person("Jean-Luc", "Godard");

• Has methods getFirstNames and getLastName

• If we have some method

    public void
        getInLine(Person p) {
            // ...
        }

    then we can pass person to it

final Student student
    = new Student(
            "Jean-Luc", "Godard",
            new int[] {1250678, 1250678, 1250678});

• Has methods getFirstNames, getLastName and getCourseIds

• We can also pass student to getInLine
  – Every Student is also a Person
  – But not every Person is a Student

What about the fields

public class Person {
    private final String firstNames;
    public String getFirstNames() {
        return firstNames;
    }
    // etc.

• Person and Student each have private fields

• Could we add something like this method to Student?

    public String statusLine() {
        return firstNames + " " + lastName + " in " + courseIds.length + " classes";
    }

public class Student
    extends Person {
    private final int[] courseIds;
    public int[] getCourseIds() {
        return courseIds;
    }
    // etc.
• We cannot!
  – When a field is parked private, it is accessible only in the class where it is defined
  – But getLastName() is public, so the methods are accessible even if the field isn’t
  – We can also label protected instead of public or private — accessible in subclasses

So subclasses can add functionality…

```java
public class Student extends Person {
    private final int[] courseIds;
    public Student(final String firstNames,
                   final String lastName,
                   final int[] courseIds) {
        super(firstNames, lastName);
        this.courseIds = courseIds;
    }
    public int[] getCourseIds() { return courseIds; }
}

... and subclasses can also change parent methods

public class Person {
    // ...
    public String getEmailAddr() {
        return firstNames
            .substring(0,1)
            + lastName
            + "@gmail.com";
    }
}

• Every person has a gmail address, just assume it’s a boring one made from the name
• But students have a school address

public class Student
    extends Person {
    // ...
    @Override public String
        getEmailAddr() {
        return lastName
            .substring(0,4)
            + firstNames
            .substring(0,3)
            + "597@uwlax.edu";
    }
}

• The definition of getEmailAddress in Student overrides the definition in Person

45
Using these methods
  Assume we have a method in a third class

```java
public class Driver {
    // ...
    public void printFancyEmail(Person p) {
        System.out.print(p.getFirstNames());
        System.out.print(" ");
        System.out.print(p.getLastName());
        System.out.print(" <");
        System.out.print(p.getEmailAddr());
        System.out.println(">");
    }
}
```

- What will this method print for an instance of `Person`? Of `Student`?

Subclasses can also change parent methods

- The method

```java
public void printFancyEmail(Person p) {
    System.out.print(p.getFirstNames());
    System.out.print(" ");
    System.out.print(p.getLastName());
    System.out.print(" <");
    System.out.print(p.getEmailAddr());
    System.out.println(">");
}
```

- Called with `Person` instance

```java
final Person bill = new Person("William", "Shatner");
printFancyEmail(bill);
```

  - Prints
    
    William Shatner <WShatner@gmail.com>

Subclasses can also change parent methods

- The method

```java
public void printFancyEmail(Person p) {
    System.out.print(p.getFirstNames());
    System.out.print(" ");
    System.out.print(p.getLastName());
    System.out.print(" <");
    System.out.print(p.getEmailAddr());
    System.out.println(">");
}
```

- Called with `Student` instance
final Person noamy = new Student("Noam", "Chomsky", new int[] { 1250678, 1261800, 1350786 });
printFancyEmail(noamy);

– Prints
  Noam Chomsky <chomnoa597@uwlax.edu>
– Because the version of getEmailAddr depends on the type of the object itself, not the type of the variable’s declaration

Relevant methods are set by object creation

final Person bill = new Person("William", "Shatner");
final Person
  noamy = new Student("Noam", "Chomsky", new int[] { 1250678, 1261800, 1350786 });

• It doesn’t matter that the objects are assigned after creation to variable declared to be Person
• What matters is that they are created as a Person and Student
  – These types, at creation time, determine the methods which will run
• The determination stays even if we put the actual instances in a context which "forgets" the specific type
• When we pass noamy to printFancyEmail:
  – Inside printFancyEmail we can’t make assumptions that the argument might be a Student
  – Java can only assume it is a Person
  – But it is the object itself that knows how to run getEmailAddr

• This effect is known as polymorphism
  – "Many forms"
  – We have already seen polymorphism in action, with interfaces
  – In fact two things in Java are called polymorphic; this one is method polymorphism, also called dynamic dispatch
    * We will explore parametric polymorphism in Week 14

Exercises 15.1 through 15.3 refer to class Rectangle:

public class Rectangle {
  private final double width, height, leftX, upperY;

  /**
   * Construct the representation of a rectangle on the plane.
   * @param width The width along the x-axis of this rectangle
   * @param height The width along the y-axis of this rectangle
   * @param leftX The x-coordinate of the left edge of the rectangle
   * @param upperY The y-coordinate of the upper edge of the rectangle
   */
  public Rectangle(final double width, final double height, final double leftX, final double upperY) {
    this.width = width;
  }
this.height = height;
this.leftX = leftX;
this.upperY = upperY;
}

public double getWidth() { return width; }
public double getHeight() { return height; }
public double getLeftX() { return leftX; }
public double getUpperY() { return upperY; }
}

Exercise 15.1. Add a method overlaps to class Rectangle which
• Takes one argument, which is also a Rectangle, and
• Return true when the two rectangles overlap in the plane with an area greater than 0, or false otherwise.
As usual, write a main method for Rectangle which verifies that overlaps is implemented correctly.

Exercise 15.2. Add a static method centeredAt to class Rectangle which
• Takes four double arguments,
  1. A width
  2. A height
  3. An x-coordinate
  4. A y-coordinate, and
• Returns a Rectangle whose center is at the given point.
Extend the main method for Rectangle to verify that centeredAt is implemented correctly.

Exercise 15.3. Write a class Square which extends Rectangle. Square should have a single constructor
which takes three double arguments: the first for the length of a side of the triangle, and the other for the x- and y-coordinates of the upper-left corner of the square.
Extend the main method for Rectangle to verify that your constructor for Square is implemented correctly.

15.2 Class Object
Everything is an Object
• Even if you don’t declare it, every class we define extends Object
• It is the ultimate Java superclass
• Methods declared on Object are available for every object of every class. These methods include:
  toString() — produce a representation of the object as a String
     – By default, include the class name and the address of the object
     – Overrides of this method should include the @Override annotation
  equals(Object o) — test if this is equal to another object
     – By default, tests whether the two are exactly the same instance
       * So it looks at the pointers and the location in memory, not the contents
     – For most classes, this is not what we want "equals" to mean, so this method is commonly overridden
       * In String, for example, equals is overridden to check the length and then the corresponding characters
Exercise 15.4. The class Card models playing cards

```java
public class Card {
    private final String suit;
    private final int rank;

    public Card(final String suit, final int rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public String getSuit() { return suit; }
    public int getRank() { return rank; }
}
```

Add an equals method to this class which overrides Object.equals, and which returns true when the two Card instances have both the same rank and the same suit. Remember to think about the appropriate comparison techniques for each field, and add tests of the correct operation of equals to your main method.

Exercise 15.5. Add a toString method to class Card which overrides Object.toString, and which

- Returns an output string of the form RANK of SUIT. So for example the statements
  
  Card c = new Card("hearts", 5);
  System.out.println(c);

  should print 5 of hearts

- Prints rank 1 as ace, rank 11 as jack, rank 12 as queen, and rank 13 as king. So for example the statements

  Card c = new Card("spades", 12);
  System.out.println(c);

  should print queen of spades. Add tests to ensure that toString returns correct results to your main method.

Exercise 15.6. Add a method equals to class Complex of Exercise 13.18 which overrides Object.equals, and which equates two instances of Complex exactly when they have both the same real component, and the same complex component. Add tests to your main method to ensure that equals returns correct results.

Exercise 15.7. Add a toString method to class Complex of Exercise 13.18 which overrides Object.toString, and which

- Returns the single-character 0 for any instance in which both components are zero
- Returns the string representation of the real component only when the complex component is zero
- Returns the string representation of the imaginary component followed by i when the real component is zero
- Returns a string of the form X+Yi otherwise

Add tests to your main method to ensure that toString returns correct results.

Exercise 15.8. Add toString methods to classes CartesianComplex and PolarComplex of Exercise 14.1 which override Object.toString, and which both return a string in the same format as described in Exercise 15.7. Add tests to your main methods to ensure that toString returns correct results.
Exercise 15.9. Add equals methods to classes CartesianComplex and PolarComplex of Exercise 14.1 which override Object.equals, and which test for standard equality of complex numbers for any representation implementing interface Complex. So for example, the statements

```
Complex cm2i = new CartesianComplex(-2,0),
    pm2i = new PolarComplex(2,Math.pi);
System.out.println(cm2i.equals(pm2i));
```

should print true. Add tests to your main methods to ensure that equals returns correct results.

Object is not BigInt
   In BigInt we use isEqualTo
   • Should we instead override equals?
   • We weren’t talking about overriding at that point, but it is the better approach.
   • But there’s one problem
     – isEqualTo takes a BigInt as its argument
     – But equals can take any Object
     – To test semantic equality of BigInt objects, we need the digits field
     – But that field doesn’t exist on Objects, so Java rejects this code:
       ```
       @Override public boolean equals(Object that) {
         if (this.digits.length != that.digits.length) {
             return false;
         } // ... otherwise compare digits ...
       }
       ```

From Object to BigInt
   To make equals work, we must explicitly check the type, and make a downcast
   ```
   @Override public boolean equals(Object o) {
     if (o instanceof BigInt) {
         final BigInt that = (BigInt)o;
         return this.isEqualTo(that);
     } else {
         return false; // Not equal if not the same type
     } }
   ```

16 Lectures 20-22

Listening to the outside world
   This semester we focused on information moving around within a program
   • From method to method
   • Into and out of arrays and objects
   We have also printed information — but otherwise our classes have mostly not interacted with the outside world
   • Early, we looked at a few examples of reading information from the keyboard
   We now consider building graphic user interfaces (GUIs) which allow us to both receive information from, and present information to, a user
Graphic user interfaces

• Different from a text-based program
  – Design and layout of a graphic window
  – Buttons can usually be pressed in any order
  – Different criteria for exiting

• The central structure of GUI programming is the event loop
  – The main activity is to loop "forever," each time waiting for a new event
    * One button click
    * One keypress
    * One system notification
    * One timer that we'd set going off
    * One background task that we'd started completing
  – Then dispatch the event, and loop again
    * Exiting is something we do explicitly, as a reaction to a particular sort of event
  – More programming if you do it from scratch
    * But Java libraries take care of much of the details

• Even more important to separate different parts of the program
  – In main for text programs, good to separate user interaction from the guts of the program
  – In GUI programs, we’ll see separation of
    * Domain-specific classes — the back end
    * GUI elements (which sometimes need to know about program specific stuff) — the front end
    * Deciding how to react to mouse and keyboard events

Anatomy of a program with a GUI

• The window itself, and the window components that go into it
  – A JFrame
  – A helper for arranging the components within the window
  – An object of some subclass of Layout, often a GridBagLayout, plus helper objects describing different constraints and choices
  – A separate object for each component
  – The different kinds of component have different classes: JLabel, JTextField, JTextArea, JButton, JTable etc., etc.
  – Add scroll bars via JScrollPane, or menus

• Additional objects implement interfaces to react to user actions
  – Today, we will skip these details — we will assemble windows, but they will not yet react to the mouse and keyboard
  – An ActionListener, which receives ActionEvents

• Use fields in the window’s class to hold objects particular to the domain
  – Example: Eclipse is a GUI which works with your files, a compiler, and a JVM
    * So the Eclipse window would know about project options, where the compiler lives on your system, information about the program, etc.

• The main routine is short, may do nothing more than activate the primary window
17 Lecture 23

Promising and delivering methods

- So far we’ve seen
  - Classes, which always provide full implementations of methods
  - And interfaces, only describing methods implemented elsewhere
- *But it doesn’t have to be one extreme or the other!*
  - We can define a sort of class in which some method are promised but not actually defined
  - We can’t actually create objects of such "incomplete" classes, but:
    * We can use them for the types of method parameters
    * Other classes can extend them
- It can be useful as we structure our program to stage partially-completed descriptions of classes
  - Bundle common implementation aspects together
  - Then further subclasses can differ

Remember: our class of data sets

```java
public class Data {
    public double data[];

    public double mean() {
        double total = 0.0;
        for(int i=0; i<data.length; i++) {
            total += data[i];
        }
        return total/data.length;
    }

    // We added various other methods too
}
```

- What if an array isn’t right for us?
  - If we often need to add data points, maybe in the middle
  - If we’re concerned about the order of the data
  - If there’s some other structure to the data that that arrays cannot capture

Separate the operations on the data store from the actual representation

```java
public abstract class AbstractDataSet {

    public abstract int getDataSetSize();
    public abstract double getDataSetItem(int index);
    public abstract void addDataSetItem(double d);

    public double mean() {
        double total = 0.0;
        for(int i=0; i<getDataSetSize(); i++) {
            total += getDataSetItem(i);
        }
        return total/getDataSetSize();
    }
```
We cannot create instances of AbstractDataSet

public class AbstractCreator {
    public static void main(String[] args) {
        final AbstractDataSet ads = new AbstractDataSet();
        System.out.println(ads);
    }
}

• When we compile:

    > javac AbstractCreator.java
    AbstractCreator.java:4: error: AbstractDataSet is abstract;
    cannot be instantiated
       final AbstractDataSet ads = new AbstractDataSet();
              ^
    1 error

We can add back our arrays through a concrete subclass

public class DataSet extends AbstractDataSet {
    private final double[] dataStorage;
    private int inUse=0;

    public DataSet(int capacity) {
        this.data = new int[capacity];
    }

    public int getDataSetSize() {
        return inUse;
    }
    public double getDataSetItem(int index) {
        return dataStorage[index];
    }
    public void addDataSetItem(double d) {
        dataStorage[inUse] = d;
        inUse = inUse+1;
    }
}

• We do not need to define mean — we inherit from AbstractDataSet

An implementation that cares about order might use a TreeSet

public class TreeDataSet extends AbstractDataSet {
    private final TreeSet<Double> theSet = new TreeSet<Double>();

    public int getDataSetSize() {
        return theSet.size();
    }
}
public double getDataSetItem(int index) {
    final Iterator<Double> iter = theSet.iterator();
    int i=0;
    while (iter.hasNext()) {
        final double d = iter.next();
        if (i == index) {
            return d;
        } else {
            i += 1;
        }
    }
    throw new IndexOutOfBoundsException();
}

public void addDataSetItem(double d) {
    theSet.add(d);
}

Also in between interfaces and classes: default methods
In the current version of Java, we can give default implementations to methods

• Will be inherited by classes which implement the method
• But can also be overridden
• Can refer to other methods in the interface
  – Even if those other methods do not have a default implementation

Behavior of a data set
As an abstract class
public abstract class AbstractDataSet {

    public abstract int getDataSetSize();
    public abstract double getDataSetItem(int index);
    public abstract void addDataSetItem(double d);

    public double mean() {
        double total = 0.0;
        for(int i=0;
            i<getDataSetSize();
            i++) {
            total += getDataSetItem(i);
        }
        return total/getDataSetSize();
    }
}

public class DataSet
extends AbstractDataSet {
private final double[] dataStorage;
private int inUse=0;

// ...
}

As an interface with defaults

public interface DataSetInterface {
    public int getDataSetSize();
    public double getDataSetItem(int index);
    public void addDataSetItem(double d);

    public default double mean() {
        double total = 0.0;
        for(int i=0; i<getDataSetSize(); i++) {
            total += getDataSetItem(i);
        }
        return total/getDataSetSize();
    }
}

public class DataSet implements DataSetInterface {
    private final double[] dataStorage;
    private int inUse=0;

    // ...
}

Class or interface?

- Can we instantiate it?
- Can it spell out method bodies?
- Can it promise method signatures?
- From how many can a class derive?
- Can it have fields?
- Can it have constructors?
Java provides built-in representations of many data structures

• Today we’ll look at class ArrayList
• Based on arrays
  – So accessing an element by its index is fast
  – But inserting an element into the middle of a list can be slower
• The class keeps track of the size of the array and the size of the list
  – So the number of the elements in the list may differ from the size of the underlying array
  – When the array needs to grow, the ArrayList allocates new space without the programmer needing to know
• No direct access to the underlying array
  – Instead, ArrayList defines an application programmer interface (API) of different operations

As with an array, name the type of list elements

• When we use ArrayList as a type, we mention the type of the elements of the list
• ArrayList<Integer>, ArrayList<Character>, ArrayList<BigInteger>, etc.
  – We’ll discuss this < > notation on Wednesday
• The type of elements we try to add to a list must match the declared type
• Elements we extract from the list will match the given type

Some ArrayList<E> constructors and methods

• ArrayList() — Constructor for a new empty list
• ArrayList(int init) — Constructor for a new empty list, where we tell Java that it should have capacity for at least init entries
• size() — Returns the number of elements now in the list
• isEmpty() — Returns true if the list contains no elements
• add(E element) — Append the element to the end of the list
• set(int index, E element) — Sets the element at the given position of the list
  – It’s an error if there’s no such element of the list
• contains(Object o) — Returns true if o is now in the list
• get(int index) — Returns the given element of the list
• remove(int index) — Removes the given element from the list
• clear() — Empties the list
A small example

```java
import java.util.ArrayList;

public class UseArrayList {
    public static void main(String[] argv) {
        final ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);

        for(int i=0; i<numbers.size(); i++) {
            final int number = numbers.get(i);
            System.out.println(number);
        }
    }
}
```

Changing the ArrayList

```java
public class ChangeAnArrayList {
    public static void main(String[] argv) {
        final ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);

        for(int i=0; i<numbers.size(); i++) {
            numbers.set(i, 2*numbers.get(i));
            System.out.println(numbers.get(i));
        }
    }
}
```

ArrayList provides an Iterator

ArrayList (like many standard Java library classes) can give us an *iterator* — a way of accessing contents one by one

- An iterator for elements of type `E` has two essential methods
  - `boolean hasNext()`
  - `E next()`
- Used to generalize the idea of a *loop* beyond just a for-loop

```java
public static void main(String[] args) {
    final ArrayList<String> names = new ArrayList<String>();
    names.add("Moe");
    names.add("Larry");
    names.add("Curly");
}
```
final Iterator<String> namesIterator = names.iterator();
while (namesIterator.hasNext()) {
    System.out.println(namesIterator.next());
}

- Prints
  Moe
  Larry
  Curly

The **iterator** method is a member of an interface
ArrayList (and many other classes!) implement the interface Iterable
- **Iterable** = things which can be iterated
- **Iterator** = the thing which does the iteration on behalf of something else
- **Iterable** drives Java's generalized for loop

```
for(final String name : names) {
    System.out.println(name);
}
```

Requires that names conform to Iterable<String>

**Interfaces in the collections hierarchy**
If we look at the ArrayList<E> documentation, we see that it implements several interfaces:
- **Iterable<E>**
  - Specifies the iterator() method
- **Collection<E>**
  - Specified methods like clear(), add(E item) and contains(Object o) that are about one thing containing another
- **List<E>**
  - Specifies methods like add(int index, E element) that rely on an order among elements
- Several different classes implement these interfaces in different ways

**HashSet**
HashSet is another standard Java library class

```
public static void main(String[] args) {
    final HashSet<String> names = new HashSet<String>();
    names.add("Moe");
    names.add("Larry");
    names.add("Curly")
    names.add("Curly");

    final Iterator<String> namesIterator = names.iterator();
    while (namesIterator.hasNext()) {
        System.out.println(namesIterator.next());
    }
}
```
HashSet is a set, so no duplicates

• We can’t make assumptions about the order that things will print
  – HashSet makes no guarantees about the order of its elements — as we’d expect for a mathematical set!
  – But it’s a set, so Curly will not be printed twice

**TreeSet**

```java
class TreeSetExample {
    public static void main(String[] args) {
        final TreeSet<String> names = new TreeSet<String>();
        names.add("Moe");
        names.add("Larry");
        names.add("Curly");

        final Iterator<String> namesIterator = names.iterator();
        while (namesIterator.hasNext()) {
            System.out.println(namesIterator.next());
        }
    }
}
```

• No duplicates, as we would expect from a set
• But it *does* keep an order among its elements
• Prints
  
  Curly
  Larry
  Moe

### 19 Lecture 25

**Type parameters**

Consider how we write and use a method:

• We specify formal parameters that represent inputs to the method
• We write the method so that it works regardless of what those inputs actually are
• When we call (invoke, use) the method, we pass in actual values (arguments) for it to process.

We would like to be able to do something similar for a class:

• Specify parameters that represent type "inputs" to the class
• Write the class so that it works regardless of what those type inputs actually are
• Provide actual types (arguments) for the type "inputs" when we use the class (at variable declarations and object instantiations)
Java generics

Generic types provide a way to do this by using the concept of a type variable

- Added to Java 5 in 2004
- Known from more experimental languages since the 80s
- Allows for classes and methods to be written for any complex type

Specifically, generics allow type parameters when defining classes, interfaces, and methods

A generic class is a class that is defined with one or more type parameters (type variables). (A class that takes “inputs”.)

```java
public class MyGenericClass<T1, T2, ..., Tn> { /* .. */ }
```

More info at [https://docs.oracle.com/javase/tutorial/java/generics/types.html](https://docs.oracle.com/javase/tutorial/java/generics/types.html)

An analogy with methods

- A method specifies input values via formal parameters

```java
public static void someMethod(int var1, String var2) { /* .. */ }
```

- `var1` and `var2` are formal parameters for `someMethod`
- Can be used anywhere in the method itself where an `int` or a `String` would be used

- A generic class specifies type values via type parameters

```java
public class MyGenericClass<TypeVar1, TypeVar2> { /* .. */ }
```

- `TypeVar1` and `TypeVar2` are type parameters for the class
- Can be used anywhere in the class itself where a `type` would be used

A simple example

A regular class

```java
public class StringBox {
    private String data;
    public StringBox(String d) {
        data = d;
    }
    public void set(String d) {
        data = d;
    }
    public String get() {
        return data;
    }
}
```

- Defines a type called `StringBox`
A generic class

```java
public class Box<SomeType> {
    private SomeType data;
    public Box(SomeType d) {
        data = d;
    }
    public void set(SomeType d) {
        data = d;
    }
    public SomeType get() {
        return data;
    }
}
```

• Defines a generic type which requires an input in order to be used

Generic type invocation

• Method invocation passes arguments (values) to a method

```java
public static void someMethod(int var1, String var2) { /* ... */ }
public static void main(String[] args) {
    someMethod(42, "Hello, World!");
}
```

– Method is run using 42 and "Hello, World!" for var1 and var2

• Generic type invocation passes type arguments to a generic class

```java
// Create parameterized type
final Box<String> wordContainer;
// Instantiate
wordContainer = new Box<String>("Hello, World!");

// Another type and instance
final Box<Integer> carBox = new Box<Integer>(57);

// Creates the parameterized types Box<String> and Box<Integer>
// Instantiates objects of these parameterized types
```

Interfaces with type parameters

Interfaces can also take type parameters

• Like for iterator in Iterable, for next in Iterator

• Gives an object which lets us see the elements of an array or list one at a time

```java
final String[] myStrings;
// Setup for myStrings omitted

final Iterator<String> iter = myStrings.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```
An interface of things we can compare

Comparable is a generic interface for comparisons

- Parameter is the type to which it is valid to compare

```java
public interface Comparable<T> {
    public int compareTo(T o);
}
```

- So we can compare an Integer to an Integer, and a String to a String, but not an Integer to a String

## 20 Hints on selected exercises

**Exercise 3.9** Use the logarithm function for base 10.

**Exercise 10.5** Use two different int variables as two different pointers into the list — one from the start, looking for Ws; the other from the end, looking for Rs. Use a while loop instead of a for loop to decide whether there are more swaps to be made.

**Exercise 10.8**

- Identify lo and hi indices
- Structure your searching loop to continue as long as lo<hi.
- When target is less than what you find at the midpoint between lo and hi, update hi to exclude the indices where you now now that target could not possibly be; and similarly for when target is greater than the middle element.

**Exercise 13.11** Allocate a new array for the DataSet object, and then copy the numbers into it.

**Exercise 13.15** Add a pair of private fields:

- A boolean field newData which set to true whenever there is new data.
- A double field mean which is sometimes set to the mean value.

The method mean should check whether there is new data: if not, then it can used the stored value; if so, it should first update both of these two fields.

**Exercise 13.18**

- **Magnitude:** \( |x + iy| = \sqrt{x^2 + y^2} \)
- \((x + iy) + (z + iw) = (x + z) + i(y + w)\)
- \((x + iy) - (z + iw) = (x - z) + i(y - w)\)
- \((a + bi) \cdot (c + di) = ac + adi + bci - bdii = (ac - bd) + i(ad + bc)\)
- **Conjugate:** \(\overline{a + ib} = a - ib\)
- \(\frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)}.\) Note that the product of a number with its conjugate is a real number.