Outline

Contents

1 Welcome 2
2 Variables and methods 4
3 Our discipline 8
4 Selection 12
5 Select again 14
6 Iteration 16
7 While loops and nested loops 18
8 Array basics 22
9 Changing arrays 24
10 Uses of arrays 26
11 Sorting an array 31
12 Multidimensional arrays 34
13 Recursion 36
14 Classes and objects 36
15 Mutation 40
16 Objects as results 43
17 Interfaces 44
18 Inheritance 47
19 Lectures on user interfaces 55
20 Closing lecture: The road ahead 56
21 Hints on selected exercises 59
1 Welcome

1.1 Introduction

A program, and what can happen with it

```java
public class Kilograms {
    public static void main (String [] args) {
        final 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 — sometimes 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. Before you can complete quizzes or submit work to D2L, you must be able to log in to D2L. Here is how you do this:

• Open a browser on your computer, and go to www.uwlax.edu/d2l/
• Select your login method: NETID
• Enter your User ID and Password.

If you do not know your User ID or Password, or if you have trouble with logging in, go to the ITS Help Desk for assistance. The Help Desk is located in Room 103 of the Wing Technology Center (between Graff Main Hall and Morris Hall). Their hours and other information are on their web page. Do not wait until the last minute to straighten out any problems with your D2L account — last-minute login problems are not grounds for extensions on deadlines.

Exercise 1.2. In Eclipse, create a new project called Hello World, and within it create a class called HelloWorld (note that there is a space in the project name, but there are no spaces in the class name). 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) {
        System.out.println("Hello world! Welcome to Java!");
    }
}
```

Use the how-to links at http://cs.uwlax.edu/~jmaraist/lab-guide/ for step-by-step instructions for each step of this exercise.

2 Variables and methods

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

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:

  Scanner scnr = new Scanner(System.in);
  int quantity = scnr.nextInt();

• In today’s reading:
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

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 functions are provided as methods in a standard Java library
  - Its full name is java.lang.Math
  - Documentation is online:
    * Front page for all Java standard classes: [https://docs.oracle.com/javase/8/docs/api/](https://docs.oracle.com/javase/8/docs/api/)
    * The Math class page [https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)

Exercise 2.1. Answer these questions by writing short Java programs

- 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 2.2. 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?

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
Exercise 2.3. Write a Java class TempConverter with a static method toCelsius, which takes a Fahrenheit temperature as a double value, and returns the equivalent Celsius temperature.

Where do we find methods?

- We can define our own methods
  - Like toCelsius
- Java provides many standard methods for us
  - Like the Math class

How do we use methods?

One method can be called from another method

- Methods help structure the programs we write
- For example:

  ```java
  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 is 23.88888888888889C
  sin(pi/2) is 1.0

Exercise 2.4. Write static methods f1, f2 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 2.5. 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 2.6. 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 2.7. You have probably run across the factorial function in your math classes. It is defined by two rules:

\[
\begin{align*}
0! &= 0 \\
n! &= n \cdot (n-1)! & \text{ when } n > 0
\end{align*}
\]

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 the example above — there, 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.

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 2.8. Write a class NumberLengthFinder with static method lengthInDigits which takes a number as its argument, and returns the number of digits it takes to write down that number (in base-10). Your lengthInDigits method should

- Take a single long argument, and
- Return a long result.

(For a hint, see p. [59])

3 Our discipline
**Our approach to software design**

There is more to software design than just programming!

- Implementing methods in Java is a key step, but it is *not* the beginning and end of what this class is about

In this class we will follow a specific four-step *methodology* (or *discipline*) for producing quality software

0. **Stub your goal methods**

- Almost all of the problems we solve will ask for a class and methods with particular behavior
- We will always start off by writing a Java class with *stub versions* of the methods which are the goal of an exercise
  - With the right argument and result types
  - But with only a "dummy" result
  - (Or empty, if the method is `void`)
- Make it compile!
  - We start right here at the beginning making sure our code always compiles

1. **Codify your understanding of the problem as tests**

How do you know if your work is correct?

- By running them, and checking the results!
- But we need to run each method with several different arguments to check for special cases
- And we would need to run them every time we make a change to make sure that didn’t break anything
- So we will need to run several tests several times as we develop our solution
  - All this sounds very tedious!

The solution: put the tests, all of them one after the next, in the `main` method

- We set up our tests before writing the methods
  - Testing is not an afterthought
  - It drives our software development process
  - When we see tests that fail, we pick one, and fix it
  - When all the tests pass, the program is finished
- Since we have set up stub versions of the methods, we can already compile and run our tests
  - They will all fail at first — but who cares!?
  - We always keep our code able to compile and run

2. **Design the methods**

It is time to think about the methods — but we do not start by just cranking out Java.

We first write a *design* for the method body

- In plain English, not formal Java
  - And not just rephrasing Java statements in English!
- As comments in the method body
  - Use the `//`-form of comments
- Explain *why* we preform certain statements, in term of achieving method goals
3. Implement the methods

At this final step we are in a position to implement the Java statements in our goal method bodies

- Place statements immediately below the particular design comment to which they correspond
- What if your original design idea is not quite correct?
  - That’s OK — but be sure update the design to correspond to the actual implementation
  - At first, the design comments are your roadmap to the implementation
  - Later, the design comments, explain and justify your implementation

Exercise 3.1. Write a Java class DoubleIt with

- A method getDoubled which takes as parameters a single int value, and which returns the int value which is twice the value of the argument.
- A main method which makes at least three different test calls to getDoubled, and which prints for each the test argument, the expected output from the call to getDoubled with that argument, and the result returned by getDoubled.

Follow our discipline of software design, and complete this work in four steps in order:

0. Stubs of all methods.
1. Tests in the main method.
2. A design — but no implementation — for getDoubled.
3. The implementation for getDoubled.

Make sure your code compiles and runs at each step (add an empty stub for the main method at Step 0).

Exercise 3.2. A TwoMult sequence is a sequence of numbers where each number (after the first two) is the product of the two prior numbers. Set up a Java class TwoMult with

- A void method printTwoMult which takes as parameters the first two numbers of a TwoMult sequence, and which when completed will print the next three numbers of the sequence.
- A main method which makes at least five different test calls to printTwoMult, and which displays for each the test arguments, the expected output from a call to printTwoMult with those values, and then calls printTwoMult.

Follow our discipline of software design, and complete this work in four steps in order, as in Exercise 3.1.

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);
```

Exercise 3.3. Extend Exercise 3.1 to add another method printDoubled. This method should also take as parameters a single int value, but printDoubled should print a nice message with the argument and its double, and return no result. So for example, when calling printDoubled(3), your method might print

3 doubled is 6.

Follow our discipline of software design as usual:
0. First add a stub for printDoubled to class DoubleIt.
1. Add tests for printDoubled to the main method. Do not remove the old tests for getDoubled — old tests let us make sure we do not break old code when adding new code!
2. A design — but no implementation — for printDoubled
3. The implementation for printDoubled

Again, make sure your code compiles and runs at each step.

Exercise 3.4. Write a class ClockTime with a method getClockTime which
• Take a single int argument representing a number of seconds, and
• Returns a string representing the given 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, the result of calling getClockTime(5025) would be 1:23:45. Follow our discipline of software design as usual (so your class ClockTime will contain a main method in addition to getClockTime). 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.
4 Selection

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

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

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

Exercise 4.2. 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 4.3. 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 4.4.** 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.

**Exercise 4.5.** 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 4.6.** Write a class `TwoSorter` with static method `printInOrder` which

- Takes two arguments of type `int`,
- Returns nothing, and
- Prints the two numbers in ascending numeric order.

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

**Exercise 4.7.** 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 4.8.** 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 Select again

Boolean operators

• Conditions are not allowed just in if statements
• Just as there are types for numbers, there is a type for boolean values
  
  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
  || 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 TripleChecker with a static method isTriple which
  • Takes a single integer, and
  • Returns true when the integer argument is exactly divisible by 3.

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 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 5.5. 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.

Exercise 5.6. Write a class `IntervalIncludeChecker` with static method `containsPoint` which

- Takes three `double` values `x1`, `x2`, and `z`
  - The first two parameters `x1` and `x2` correspond to the start- and endpoints of one interval. We can assume that `x1` will be less than or equal to `x2`.
  - The last parameter `z` is interpreted as a point on the same axis where the interval `[x1, x2]` lies.
- Returns an integer value to describe the position of `z` relative to the interval `[x1, x2]`:
  - The result should be negative if `z` is less than `x1`, that is, if the point lies to the left of the interval.
  - The result should be positive if `z` is greater than `x2`, that is, if the point lies to the right of the interval.
  - The result should be zero if `z` is both greater than or equal to `x1` and less than or equal to `x2`, that is, if the point in the interval.

In both this and the next exercise, assume that the interval is closed, that is, that the endpoints are included in the interval. So for example

- `IntervalIncludeChecker.containsPoint(0.0, 2.0, 9.0)` should return some positive number.
- `IntervalIncludeChecker.containsPoint(0.0, 2.0, 1.0)` and `IntervalIncludeChecker.containsPoint(0.0, 2.0, 2.0)` should both return zero.
- `IntervalIncludeChecker.containsPoint(0.0, 1.0, -1.0)` should return some negative number.

Exercise 5.7. Write a class `IntervalOverlapChecker` with static method `intervalsOverlap` which

- Takes four `double` values `x1`, `x2`, `y1`, and `y2`
  - The first two parameters `x1` and `x2` correspond to the start- and endpoints of one interval. We can assume that `x1` will be less than or equal to `x2`.
  - The latter two parameters `y1` and `y2` correspond to the start- and endpoints of a second interval. Again, we can assume that `y1` will be less than or equal to `y2`.
- Returns `true` when the first and second intervals overlap.

As in the previous true, take both intervals to be closed, that is, that the endpoints are included in the intervals. So for example

- `IntervalOverlapChecker.intervalsOverlap(0.0, 2.0, 9.0, 11.0)` and `IntervalOverlapChecker.intervalsOverlap(9.0, 11.0, 0.0, 2.0)` should both return `false`. 

• `IntervalOverlapChecker.intervalsOverlap(0.0, 2.0, 1.0, 3.0)` and `IntervalOverlapChecker.intervalsOverlap(1.0, 3.0, 0.0, 2.0)` should both return true.

• `IntervalOverlapChecker.intervalsOverlap(0.0, 1.0, 1.0, 2.0)` should return true.

6 Iteration

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:
Exercise 6.1. In these exercises, do not use Math methods to calculate the square; just use multiplication.

1. Write a class SimpleLoop whose main method uses a loop to print the squares of the integers from 0 to 10.

2. Write a class SimpleLoop2 whose main method uses a loop to print the squares of the integers from 10 to 30.

3. Write a class SimpleLoop3 whose main method uses a loop to print the squares of the even integers from 10 to 30.

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 lowercase.

The standard methods toUpperCase and toUpperCase 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 \textit{choose} function from probability is defined as
\[
\binom{n}{m} = \frac{n!}{m!(n-m)!}.
\]

Given the \texttt{factorial} method above, it is certainly possible to extend the \texttt{FactorialFinder} of Exercise 6.3 class with a method to implement \texttt{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 \texttt{long}. Write a more efficient version of \texttt{nChooseM} which only performs the multiplications and divisions which are absolutely necessary. As usual, extend the \texttt{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 \texttt{FibonacciFinder} with a static method \texttt{fibonacci} which

- Accepts a single \texttt{int} parameter $n$
- Returns a \texttt{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 \texttt{main} method with examples and expected results.

7 While loops and nested loops

LOOK BACK: Two looping patterns
We’ve seen two different patterns for how we use loops

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 \textit{before} the loop
FROM THE READING: Another kind of loop
In the book, you read about another, simpler kind of loop

```java
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

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 7.1. 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).

Exercise 7.2. The sequence $A_1$ is defined as follows:

$$A_1(n) = \begin{cases} 
    n^2 & \text{if } n \text{ is even} \\
    n + 1 & \text{if } n \text{ is odd}
\end{cases}$$

Write a class AltSequence with a static method `getSumTo` which takes an integer $n$, and returns the sum of the elements 0 through $n$ of sequence $A_1$. Use only a single loop, and test inside the loop whether a particular index is even or odd.
Exercise 7.3.  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 one, the string should use the singular `dime` instead of `dimes`.

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",

- The first and third characters `a` and `c` are self-describing letters, since `a` is the first letter in both the alphabet and in the string, and `c` is the third letter in both the alphabet and in the string.
- The second character `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.

**Nested loops**

We can *nest* one loop inside of another

- The *inner* loop runs several times each time the *outer* loop body runs

Exercise 7.5.  What do these loops 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 7.6.  Write a program to print this triangle:

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

Exercise 7.7. 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.
  ```
 ```java
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
}
```

8 Array basics

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
    ```java
    \texttt{int number = numbers[i];}
    ```
    - 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
• Pass arrays to methods, and index array elements as if the array were declared locally

    public static void arrayReceiver(double[] readings) {
        System.out.println(readings.length + " " + readings[0]);
    }

**Exercise 8.1.** 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 8.2.** 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 8.3.** What errors do you get when you:

- Try to read from 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?

Which of these errors are compile-time errors, and which are run-time errors?

**Exercise 8.4.** 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 8.5.** Update your class `MonthNamer` from Exercise 4.4 to use an array within `getMonthName`.

**Exercise 8.6.** 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 2.8

**Exercise 8.7.** 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 standard deviation: $\sigma = \sqrt{\frac{\sum(x-x_0)^2}{N-1}}$, where $\bar{x}$ is the mean and $N$ is the number of values.
Exercise 8.8. Extend method `printSummaryStats` of Exercise 8.7 to also print the median: the middlemost value contained in the array.

Exercise 8.9. Add another static method `getSummaryStats` to class `StatsFinder` from Exercise 8.7. Your `getSummaryStats` should, instead of just printing the various statistics, return 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 8.10. Modify `StatsFinder` from Exercise 8.7 (or 8.8 or 8.9) 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.

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 8.11. The dot product is a common mathematical operation on pairs of numeric vectors (arrays) of the same size. Given two vectors \((x_i)^n\) and \((y_i)^n\), their dot product \(\vec{x} \cdot \vec{y} = \sum_1^n x_i y_i\). 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.

9 Changing arrays
We can update the contents as well
The square-bracket notation can be used on the left side of an assignment

- Updates the particular location of the array, rather than the array variable itself
- Example: `names[3] = "Billy";`

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

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

Exercise 9.2. What errors do you get when you:

• Try to write to an array slot which is beyond the upper limit of an array?
• 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?

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

Exercise 9.3. Trace through the execution of this class (without running it first). What does it print?

Exercise 9.4. Consider the following class:

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 = i + 1) {
    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 9.5.** 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`

When testing your code, be sure to try both cases where `target` is found in `numbers`, and cases where `target` is not found in `numbers`.

**Exercise 9.6.** Write a class `SortedArrayIntFinder` with a static method `getIndexOf` which whose arguments and result are just as in class `ArrayIntFinder of Exercise 9.5` 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.59)

**Changing to a new array**

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

numbers = new int[] { 5, 6, 7 };
for (int i = 0; i < numbers.length; i = i + 1) {
    int number = numbers[i];
    System.out.println(number);
}
```

- The new `int[]` part is implied when we’re declaring the array variable.
- But we can’t change the length with something like

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

**10 Uses of arrays**

**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

```java
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.

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

global static void f(int[] x) {
    x = new int[] { 1, 2, 3, 4, 5 };
    x[1] = -1;
    for(int i=0; i<x.length; i=i+1) {
        System.out.println(x[i]);
    }
    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=i+1) {
        System.out.println(y[i]);
    }
}

public static void f(int[] x) {
    for(int i=0; i<x.length; i=i+1) {
        x[i] = i*100;
    }
    for(int i=0; i<x.length; i=i+1) {
        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=i+1) {
        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. 59)

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 algo-
   rithms, 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. 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 getPrimesMap which
   • Takes one argument n of type int, and
   • Returns a boolean array with n + 1 elements, so that slot [i] is true exactly when i is prime.
Exercise 10.8. Extend class `PrimesFinder` of Exercise 10.7 with a static method `printPrimes` which takes a single `int` argument `n`, and uses `getPrimesMap` to print the prime numbers less than or equal to `n`.

Exercise 10.9. Extend class `PrimesFinder` of Exercise 10.7 with a static method `primesTo` which

- Takes a single `int` argument `n`, and
- Returns an array of integers containing the prime numbers less than or equal to `n`.

Methods can return new arrays

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

  ```java
  public static int[] buildIntegerArray(...)
  ```

- The return value can be passed in from outside the array, or created within the array

  ```java
  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 10.10. 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 10.11. 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 10.12.** 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 in `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 10.13.** Write a class `WordsTaker` with a static method `getWords` which

- Takes two arguments,
  1. A `String`, expected to consist of several space-separated words
  2. An `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 `getWords` for the main method of `WordsTaker`. Be sure to examine the length of the array as well as each element.

### 11 Sorting an array

**What does this method do?**

```java
public static void MYSTERY(int[] numbers) {
    int howMany = numbers.length;
    for(int a=1; a<howMany; a=a+1) {
        for(int b=howMany-1; a<=b; b=b-1) {
            if (numbers[b-1] > numbers[b]) {
                final int tmp = numbers[b-1];
                numbers[b-1] = numbers[b];
                numbers[b] = tmp;
            }
        }
    }
}
```
The `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
  
  ```java
  for(int i=0; i<howMany; i=i+1) {
      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=a+1) { // 1
    for(int b=howMany-1; a<=b; b=b-1) { // 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=a+1) { // 1
        for(int b=howMany-1; a<=b; b=b-1) { // 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 **Multidimensional 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
final String[][] wordsWeKnow = {
    { "hello", "let", "us", "eat", "goodbye", "see", "you", "later" },
    { "bonjour", "bon", "appetit", "au", "revoir" },
    { "guten", "tag", "mahlzeit", "tschau" }
};

• Some true expressions:

    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. Write a class WordMatrix with a static method printWordMatrix which
    • Takes a two-dimension array of strings, that is a String[][], as its single argument,
    • Prints the words so that each columns of words are aligned, and
    • Otherwise returns no value.

Exercise 12.2. 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 MagicSquareChecker 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 12.3. Consider the square below:

    R A D A R
    A D A R A
    D A R A D
    A R A D A
    R A D A R

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:
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.

13 Recursion

For the lab between Midterm 2 and the next lecture, we will work a few exercises on an interesting side topic.

Exercise 13.1. Write a class RecursiveFactorial with a static method factorial which implements a recursive version of the factorial function (see Exercise 6.3) which does not use a loop, but instead uses recursion.

Exercise 13.2. Write a class RecursiveFibonacci with a static method fib which implements a recursive version of Fibonacci number calculation (see Exercise 6.5) which does not use a loop, but instead uses recursion. Make sure your method does not take excessively long on large values.

14 Classes and objects

Classes and objects

A class is a description — a sort of thing

- String is one class we've seen already

An object is one instance of a class

- "Hello" and "Goodbye" are two different String objects
  - Special syntax for creating a String object
- Ordinarily we create a new object of a class using new
  - For example, from the book:
    ```java
    Scanner scn = new Scanner(System.in);
    ```

Fields

With an array, we access the elements by number

- With an object, we access the fields by name
- Declarations look like the local variables of a method
  - But they are not part of any method
public class ThreeThings {
    int myNumber;
    String myName;
    boolean myStatus;
}

Creating one object of a class

public class ThreeThings {
    int myNumber;
    String myName;
    boolean myStatus;
}

ThreeThings is like String — there can be many different String objects, and there can be many different ThreeThings objects

Here is a way to make a ThreeThings object

ThreeThings myThree = new ThreeThings();
myThree.myNumber = 20;
myThree.myName = "Threepio";
myThree.myStatus = true;
System.out.println(myThree.myName);

Classes are not just for fields

• You have already written many static methods
  – Call them in the same way as a mathematical function
  – Pass them arguments
  – Can get a value back from them

• You have also used a different kind of method
  – The length method on strings
  – Called in the context of one particular string value

• We can write our own object methods which read and write the fields of an object

Getters and setters

In Java, the typical style is to use accessor and mutator methods instead of reading or writing fields directly

• So we would prefer ThreeThings like this:

public class ThreeThings {
    private int myNumber;
    private String myName;
    private boolean myStatus;

    public int getMyNumber() { return myNumber; }
    public String getMyName() { return myName; }
    public boolean getMyStatus() { return myStatus; }

    public void setMyNumber(int n) { myNumber=n; }
    public void setMyName(String n) { myName=n; }
    public void setMyStatus(boolean s) { myStatus=s; }
}
• *private* is the opposite of *public* — it means that the field can only be directly read or written from *inside* the class
  
  – Other classes do not have access

• So our old example becomes

```java
ThreeThings myThree = new ThreeThings();
myThree.setMyNumber(20);
myThree.setMyName("Threepio");
myThree.setMyStatus(true);
System.out.println(myThree.getMyName());
```

**Constructor**

A *constructor* is a special method called when we create an object

• And at no other time!

Constructors can

• Set up an object’s fields

• Use parameters to make the initial field values

So we could add a constructor to *ThreeThings*

```java
public class ThreeThings {
    private int myNumber;
    private String myName;
    private boolean myStatus;

    public ThreeThings(int num, String name, boolean stat) {
        myNumber = num;
        myName = name;
        myStatus = stat;
    }
}
```

// ...

**Exercise 14.1.** 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.

• Store their name in two fields *firstNames* and *lastName*

• Store their ID as an *int*

• Store their current classes as an array of *strings*

• Store the number of credits earned as an *int*
**Exercise 14.2.** 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.

**Exercise 14.3.** Add a method `getRange` to class `Vehicle` of Exercise 14.2 which calculates from 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 14.4.** 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.

**Exercise 14.5.** Extend class `QuizScores` from Exercise 14.4 with two additional methods,

- 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 14.6.** Pick a board or paper game, and write a class whose objects represent the state of play of that game, and whose method provide information in a form which is convenient for applying the rules of that game. What information should your class store as fields?

**Exercise 14.7.** 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

```java
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 14.8 through 14.11 will remind you of Exercises 8.7 through 8.10. Review your `StateFinder` implementations before starting these exercises, and expect to deploy some of the same algorithms here.
Exercise 14.8. 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 14.9. Consider your class `DataSet` from Exercise 14.8. 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. 59 for a hint if you need.)

Exercise 14.10. 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 14.11. Add a method `stdDev` to `DataSet` from Exercise 15.5 which takes no arguments and returns the standard deviation of the data. (See Exercise 8.7 for a formula for standard deviation.) Since we use the mean for calculating the standard deviation, `stdDev` will need to call `mean`.

15 Mutation

Objects can change over time

- Fields can be updated, just like local variables
- `Mutator` methods change one field
  - Mutator and accessor method are companion simple methods

Distinguishing immutable and mutable fields

- Fields which never change can receive the keyword `final`
- We can (and should) use `final` with local method variables as well
Exercise 15.1. Extend class Student of Exercise 14.1 with the assumption that a student ID will never change.

Exercise 15.2. Extend class Student of Exercise 15.1 with methods to update a student record:

- Update the name
- Report passed classes
- Enroll in new classes

Exercise 15.3. Revise class Vehicle of Exercise 14.3 with an additional private field:

- 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 15.4. Revisit your game board design from Exercise 14.6. 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 15.5. Extend the class DataSet of Exercise 14.8 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.

Exercise 15.6. Consider your class DataSet from Exercise 14.5 (without the stdDev method of Exercise 14.11). 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. 21 for a hint if you need.)

Exercise 15.7. 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 14.11 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 15.6.

Exercise 15.8. 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 14.11 to implement this idea, making sure that mean and stdDev call it only when there is new data.

Exercise 15.9. In DataSet, we considered data where the individual points have no particular identity — they were just numbers. Sometimes, individual data points have an identity beyond just their value, and each comes with its own label. For examples, the times of racers may be labeled with the racer’s name, or rainfall totals may be labelled with the location of measurement.

Write a class TopThree to keep track of the three highest labelled data values. There should be six simple accessor methods for the stored data:

- Methods getValue1 and getLabel1 return the highest data value, and its label
- Methods getValue2 and getLabel2 return the second-highest data value, and its label
• Methods `getValue3` and `getLabel3` return the third-highest data value, and its label

There should be one mutator method `reviewDatum` which
• Takes two arguments, a `double` value and a `String` label
• Returns no value
• Might update some of the top three values, if the value is higher than the ones reviewed previously

The constructor for `TopThree` should take the initial top three values and labels

```java
public TopThree(double bestValue, String bestLabel,
                double secondValue, String secondLabel,
                double thirdValue, String thirdLabel)
```

**Exercise 15.10.** Modify class `TopThree` of Exercise 15.9 so that the constructor does not assume that the three value/label pairs are already in order, but instead works out for itself which label has the highest value.

**Be mindful about mutability**
Pay attention to whether fields are mutable
• When understanding others’ software
  – To understand how the class is used
  – To understand what you can do with it
• Which Java classes have you seen…
  – That are completely immutable
  – That can change over time
• Often, immutable classes are easier to understand
  – And less error-prone

**Notes on writing classes**
• Stay organized! Write your class contents in order:
  – Fields first
  – Then the constructor
  – Then accessors
  – Then mutators
  – Then any other object methods
  – Then static methods
  – Use the navigator windows in Eclipse
• 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:
  – Use the 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

## 16 Objects as results

### 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 16.1.** 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. [60](#).

**Exercise 16.2.** Update your board game idea from Exercises [14.6](#) and [15.4](#) so that the board representations are immutable, and the methods in response to a player move return a new object representing a board in the new state.

**Exercise 16.3.** In Exercise [15.5](#) 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 [14.8](#) 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.

Exercises 16.4 and 16.5 refer to class Rectangle:

```java
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 16.4. 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 16.5. 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.

17 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();
    }
}
```

Two ways to do something

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

One way

```java
public class PrintMeasure1 implements DoIt {
    public void doSomething(int i, double x) {
        System.out.println(i*i+x);
    }
    public int doSomethingElse(String s) {
        return s.length();
    }
}
```

Another way
public class PrintMeasure2 implements DoIt {
    public void doSomething(int i, double x) {
        System.out.println(2*i*i + x);
    }
    public int doSomethingElse(String s) {
        return s.length()/2;
    }
}

Implementing two interfaces

Another simple interface:

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

A class can declare that it implements both interfaces

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

    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+x);
        }
    }

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

Exercise 17.1. 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°-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: $1 + i = \sqrt{2}e^{\pi i / 4}$.

Refactor the class Complex of Exercise 16.1 to have the name CartesianComplex, and instead define Complex to be this interface:

```java
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.

18 Inheritance

18.1 A class for representing people

Revisiting how we represent people

```java
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*

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

**Instantiating classes and subclasses**

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

- Has methods `getFirstNames` and `getLastName`
- If we have some method

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

then we can pass `person` to it

```java
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`
The super statement

What is this magic?

```java
public Student(final String firstNames,
    final String lastName,
    final int[] courseIds) {
    super(firstNames, lastName);
    // ...
}
```

Remember: every Student is also a Person

- The constructor for Person carries the instructions for setting up each instance of Person
- So to set up a Student, we must first set up the Person aspects of it
- The call to super here refers to the constructor of the superclass
  - Right number and type of arguments
  - Must be the first step of the subclass constructor

What about the fields?

```java
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";
    }
}
```

```java
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

```java
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

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

- The definition of `getEmailAddress` in `Student` overrides the definition in `Person`

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(" <");
    }
}
```
• What will this method print for an instance of Person? Of Student?

Subclasses can also change parent methods

• The method

    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

    final Person bill = new Person("William", "Shatner");
    printFancyEmail(bill);
    – Prints
        William Shatner <WShatner@gmail.com>

Subclasses can also change parent methods

• The method

    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

```java
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

**Exercise 18.1.** Write a class `Square` which extends class `Rectangle` of Exercises 16.4-16.5. `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.

### 18.2 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 18.2. 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 18.3. 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 18.4. Add a method equals to class Complex of Exercise [16.1] 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 18.5. Add a toString method to class Complex of Exercise [16.1] 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 18.6. Add `toString` methods to classes `CartesianComplex` and `PolarComplex` of Exercise 17.1 which override `Object.toString`, and which both return a string in the same format as described in Exercise 18.5. Add tests to your main methods to ensure that `toString` returns correct results.

Exercise 18.7. Add `equals` methods to classes `CartesianComplex` and `PolarComplex` of Exercise 17.1 which override `Object.equals`, and which test for standard equality of complex numbers for any representation implementing interface `Complex`. So for example, the statements

```java
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.

Exercise 18.8. Write classes which implement a media library:

- Class `Media` includes immutable private fields and public accessors for the title, primary artist, and playing time
- Classes `Video` and `Audio` both extend `Media`
  - Class `Video` includes immutable private fields and public accessors for the director, studio and actors
  - Class `Audio` includes immutable private fields and public accessors for the music company/label
- Classes `DVD`, `FromNetflix`, `CD`, and `Cassette` extend the above classes with further information specific to each format

Exercise 18.2 — card checking

Add an `equals` method to class `Card`:

```java
public boolean isEqualTo(final Card that) {
  return this.getSuit().equals(that.getSuit())
    && this.getRank() == that.getRank();
}
```

It's easy to imagine a method on `Card` that tests whether two cards are the same

```java
public boolean isEqualTo(final Card that) {
  return this.getSuit().equals(that.getSuit())
    && this.getRank() == that.getRank();
}
```

But there's one problem
- `isEqualTo` takes a `Card` as its argument
- `equals` can take any `Object` — `equals(Object o)`
- To test semantic equality of `Card` objects, we need the `getSuit()` and `getDigits()` methods
- But that field doesn’t exist on `Objects`, so Java would reject this code:

  ```java
  @Override public boolean equals(final Object that) {
      return this.getSuit() == that.getSuit() && this.getRank() == that.getRank();
  }
  ```

---

19 Lectures on user interfaces

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 *user interfaces*
- *Keyboard input* is a simple way to get plain text input from the user
- *Graphic user interfaces* (GUIs) which allow us to both receive and present information in more sophisticated ways

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

20 Closing lecture: The road ahead
20.1 The roadmap in depth

On the roadmap: programming skills, from basic to solid

On the roadmap: software engineering

On the roadmap: hardware and systems
On the roadmap: theory and numerical techniques

On the roadmap: large applications
On the roadmap: web technologies

21 Hints on selected exercises

Exercise 2.8 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 9.6 • 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 know that target could not possibly be; and similarly for when target is greater than the middle element.

Exercise 14.9 Allocate a new array for the DataSet object, and then copy the numbers into it.

Exercise 15.6 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 use the stored value; if so, it should first update both of these two fields.

**Exercise [16.1]**

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