Outline

Contents

1 Introduction .................................................. 2

2 Simple imperative programming ............................ 6
  2.1 Names and assignments ..................................... 7
  2.2 Leap years .................................................. 10
  2.3 The for-loop ................................................ 13
  2.4 Factorials .................................................... 14
  2.5 How long is that number? .................................. 19
  2.6 Fibonacci numbers ......................................... 19
  2.7 Fix that sentence ........................................... 21
  2.8 Another kind of selection .................................. 22
  2.9 Another kind of loop ........................................ 24
  2.10 Multiple loops .............................................. 25

3 Methods ....................................................... 29
  3.1 Method basics ............................................... 29
  3.2 A method for leap years .................................... 31
  3.3 A choose method ............................................ 32
  3.4 Factorial and Fibonacci methods .......................... 33
  3.5 Factorial and Fibonacci methods ......................... 36
  3.6 Methods and your work ..................................... 37

4 Arrays .......................................................... 39
  4.1 Array basics .................................................. 39
  4.2 Combining two arrays: the dot product .................... 42
  4.3 Sorting an array .............................................. 43
  4.4 The Sieve of Eratosthenes .................................. 45
  4.5 Two-dimensional arrays ..................................... 51
  4.6 Lining up columns of information ......................... 52
  4.7 Pulling words out of a line ................................. 63
  4.8 Some random stuff .......................................... 65
  4.9 Methods and arrays ........................................ 66
1 Introduction

A program

```
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.");
        return;
    }
}
```

Inputs

Outputs

Machine code

Compiler

Run on machine

Inputs

Outputs
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.");
        return;
    }
}

A program

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.");
        return;
    }
}

Parts of the program

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.");
        return;
    }
}

Add comments to describe what the program does

/**
 * 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.");
        return;
    }
}
System.out.print(pounds);
System.out.print(" pounds is ");
System.out.print(pounds / 2.2);
System.out.println(" kilograms."); // End of this line
return;

import java.util.Scanner; // User input

/**
 * Converter from pounds to kilograms.
 */
public class Kilograms {
    public static void main (String [] args) {
        Scanner scanner = new Scanner(System.in);

        // Read a number of pounds, and convert it to kilograms
        System.out.print("How many pounds? ");
        final double pounds = scanner.nextDouble();
        final double kilograms = pounds / 2.2;

        // Print a message about the conversion
        System.out.print(pounds);
        System.out.print(" pounds is ");
        System.out.print(kilograms);
        System.out.println(" kilograms.");

        return;
    }
}

The other things that happen with a program

Compiler says we have an error

Nothing!

The other things that happen with a program
Errors are frustrating

What is the most used language in programming?

Good news! There are really only six things you need to know here!

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

The less good news is that there’s a whole bunch of detail and skill associated with each of these.
Four ways Java will help you organize your work

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

2 Simple imperative programming

The six things

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

In and out of the system

Where exactly is our program running?

• Should it matter?

• Abstract away from certain details

• There’s some source for input, and some destination for output
  – Give them a name, and describe operations on them
  – But don’t get hung up on the details of exactly what they are
  – Java calls them: System.in and System.out
  – Some operations on them are written as a suffix:
    System.out.print("Hello...");
    System.out.println("again");
  – For some operations we use a helper:
    Scanner scnr = new Scanner(System.in);
    // ... then later ...
    int quantity = scnr.nextInt();
2.1 Names and assignments

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 Tuesday:

  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

It’s an update, not an equation

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

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

    System.out.print("Enter another number: "); // 5
    num = scanner.nextInt(); // 6
    total = total + num; // 7
```
System.out.print("Just one more number: "); // 8
num = scanner.nextInt(); // 9
total = total + num; // 10
System.out.println("Their sum is " + total); // 11
return; // 12
}

• Step through with inputs 4, 7 and 2

Many types of numbers
You’ve seen already:
• int — Rounded integer values
• double — Real-number values
Java also has:
• long — Integer values from a larger range
  • int runs from -2,147,483,648 to 2,147,483,647
  • long runs from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
• 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

Clock time

public static void main (String [] args) {
    final Scanner scanner = new Scanner(System.in); // 1
    System.out.print("How many seconds? "); // 2
    int given = scanner.nextInt(); // 3
final int hours = given / (60 * 60); // 4
final int minutes = given / 60; // 6
final int seconds = given % 60; // 7
System.out.printf("%d:%02d:%02d\n", hours, minutes, seconds); // 8
return; // 9
}

• Try inputs: 3,923, 2,348 and 3,606.

Printing the clock time
What about the last statement of the clock time program?
System.out.printf("%d:%02d:%02d\n", hours, minutes, seconds);

• If you tried the program, you know it just prints the time

• Why is it different than this?

System.out.print(hours);
System.out.print(":");
System.out.print(minutes);
System.out.print(":");
System.out.println(seconds);

  – If we print the number 3, would we see 3 or 03?
  System.out.print(3);

• printf abbreviates print according to format
  – Everything in the first argument prints as normal, except for percent signs
  – The percent signs indicate how we should print things

Printing the clock time

• printf(control_string, value, value, ...);
  – Only the control string is (necessarily) printed

• %d means: take the next argument, print it as an integer (d for decimal)
  – %2d means: take two spaces for it
  – %02d means: fill up extra space with leading zeroes

• Other codes for strings, floating-point, etc.
  – We’ll come back to printf later
2.2 Leap years

Calculating a leap year

• How do we know if a year is a leap year?
  – Easy: if it’s divisible by four
• An algorithm using the Six Things
• In Java:

```java
import java.util.Scanner;

public class LeapYear {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in); // 1
        System.out.print("What year? "); // 2
        final int year = scnr.nextInt(); // 3

        if (year % 4 == 0) { // 4
            System.out.println(year + " was a leap year!"); // 5
        } else { // 6
            System.out.println(year + " was not a leap year.");
        }
    }
}
```

• Try 1986 and 1900

Really calculating a leap year

• There are exceptions to the divide-by-four rule
  – Years divisible by 100 are not leap years
  – Unless they are also divisible by 400
• So we need to make multiple decisions

```java
import java.util.Scanner;

public class LeapYear {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in); // 1
        System.out.print("What year? "); // 2
        final int year = scnr.nextInt(); // 3

        if (year % 400 == 0) { // 4
            System.out.println(year + " was a leap year!"); // 5
        } else {
            System.out.println(year + " was not a leap year."); // 6
        }

        return; // 7
    }
}
```
System.out.println(year + " was a leap year!"); // 5
} else if (year % 100 == 0) { // 6
System.out.println(year + " was not a leap year."); // 7
} else if (year % 4 == 0) { // 8
System.out.println(year + " was a leap year!"); // 9
} else { // 10
System.out.println(year + " was not a leap year.");
}
return; // 11

• Try 1986 and 1900

Calculate, then output

import java.util.Scanner;

public class LeapYear {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in); // 1
        System.out.print("What year? "); // 2
        final int year = scnr.nextInt(); // 3
        final boolean isLeapYear; // 4
        if (year % 400 == 0) { // 5
            isLeapYear = true; // 6
        } else if (year % 100 == 0) { // 7
            isLeapYear = false; // 8
        } else if (year % 4 == 0) { // 9
            isLeapYear = true; // 10
        } else { // 11
            isLeapYear = false;
        }
        if (isLeapYear) { // 12
            System.out.println(year + " was a leap year!"); // 13
        } else { // 14
            System.out.println(year + " was not a leap year.");
        }
        return; // 15
    }
}

One more tweak
import java.util.Scanner;

public class LeapYear {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("What year? ");
        final int year = scnr.nextInt();

        final boolean isLeapYear;
        if (year % 400 == 0) {
            isLeapYear = true;
        } else if (year % 100 == 0) {
            isLeapYear = false;
        } else if (year % 4 == 0) {
            isLeapYear = true;
        } else {
            isLeapYear = false;
        }

        if (isLeapYear) {
            System.out.println(year + " was a leap year!");
        } else {
            System.out.println(year + " was not a leap year.");
        }

        return;
    }
}

A debugging trick

final boolean isLeapYear;
if (year % 400 == 0) {
    System.out.println("*** branch 1 ***"); // FIXME delete
    isLeapYear = true;
} else if (year % 100 == 0) {
    System.out.println("*** branch 2 ***"); // FIXME delete
    isLeapYear = false;
} else if (year % 4 == 0) {
    System.out.println("*** branch 3 ***"); // FIXME delete
    isLeapYear = true;
} else {
    System.out.println("*** branch 4 ***"); // FIXME delete
    isLeapYear = false;
}

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

**About exceptions**

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

• For now, you should just aware of them

    – You will see them as you debug your programs
    – Exercise: use (alter if you need) one of the programs from the book or from a lab exercise to make Eclipse throw an exception.
    – What does it look like?

### 2.3 The for-loop

**A simple loop**

```java
public class SimpleLoop {
    public static void main(String[] args) {
        for(int i=0; i<10; i++) { // 1
            final int squared = i*i; // 2
            System.out.println(i + " squared is " + squared); // 3
        }
    }
}
```

**How the for-loop works**

General loop structure:

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

• Steps Java takes:
2.4 Factorials

What are factorials?
A factorial is the product of all the integers from 1 up to some number.

- The factorial of 7 is $7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$.
- Abbreviated $7!$.

How can we implement the factorial function?

```java
import java.util.Scanner;

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


How can we implement the factorial function?

```java
import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("Factorial of: ");
        final int num = scnr.nextInt();
        System.out.println("It\'s: " + product);
        return;
    }
}
```

A factorial calculator

```java
import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in); // 1
        System.out.print("Factorial of: "); // 2
        final int num = scnr.nextInt(); // 3
        long product = 1; // 4
        for(int n=2; n<=num; n++) { // 5
            product = product * n; // 6
        }
        System.out.println("It\'s: " + product); // 7
        return; // 8
    }
}
```

- Try it with 4

A different factorial calculator

```java
import java.util.Scanner;
```
public class Factorial {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("Factorial of: ");
        final int num = scnr.nextInt();
        long product = 1;

        for(int factor=num; factor>0; factor--) {
            product = product * factor;
        }

        System.out.println("It’s: " + product);

        return;
    }
}

• Try it with 4

The n-choose-m function
Factorial shows up in probability and statistics
For example:
• Let’s say we have a bag of \( n \) distinct objects
• How many ways are there to choose \( m \)?
• We write this as \( \binom{n}{m} \).
• It’s calculated as \( \binom{n}{m} = \frac{n!}{m!(n-m)!} \)

How it works
Five things: A B C D E
How can we choose 3?
• A B C
• A B D
• A B E
• A C D
• A C E
• A D E
• B C D
• B C E
There are
- Five ways to pick one thing out of five
- Four ways to pick a second one from the remaining four
- Three ways to pick a third one from the remaining three
- That’s $5 \times 4 \times 3$

But we don’t want to consider the order of these three things
- Just ABC
- Not ABC, ACB, BAC, BCA, CAB and CBA
- There are $n!$ ways to order $n$ things

So a first try would be $\binom{n}{m} = \frac{n(n-1)\ldots(n-m+1)}{m!}$
- But the whole ellipses thing is informal and unsatisfying

But note that we can multiply by $(n-m)!$ in both the numerator and denominator
- That makes the top just $n!$
- And the whole formula $\binom{n}{m} = \frac{n!}{m!(n-m)!}$

How would we implement a choose calculator?

A simple choose calculator

```java
final Scanner scnr = new Scanner(System.in); // 0
System.out.print("How many total objects? "); // 1
final int total = scnr.nextInt(); // 2
System.out.print("Choose many? "); // 3
final int choose = scnr.nextInt(); // 4
long totalFact = 1; // 5
for(int n=2; n<=total; n++) { // 6
    totalFact *= n; // 7
}
long chooseFact = 1; // 9
for(int n=2; n<=choose; n++) { // 10
    chooseFact *= n; // 11
}
long diffFact = 1; // 13
```
for(int n=2; n<=total-choose; n++) { // 14
    diffFact *= n; // 15
} // 16

final long ways = totalFact / chooseFact / diffFact; // 17
System.out.println(ways + " ways to choose"); // 18
return; // 19

• Try it with 4, 2

Do less work!

• Note that when $n > m$ we have

$$n! = n \times (n-1) \times \cdots \times (n-m+1) \times (n-m)!$$

• So for $\binom{n}{m}$, we have

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} = \frac{n \times (n-1) \times \cdots \times (n-m+1) \times (n-m)!}{m!(n-m)!} = \frac{n \times (n-1) \times \cdots \times (n-m+1)}{m!}$$

• Much easier to calculate!

A better choose calculator

final Scanner scnr = new Scanner(System.in); // 0
System.out.print("How many total objects? "); // 1
final int total = scnr.nextInt(); // 2
System.out.print("Choose many? "); // 3
final int choose = scnr.nextInt(); // 4
long ways = 1; // 5
int totalFactor = total; // 6
for(int i=1; i<=choose; i++) { // 7
    ways = ways * totalFactor / i; // 8
    --totalFactor; // 9
}

System.out.println(ways + " ways to choose"); // 10
return; // 11

• Try it with 23, 4
2.5 How long is that number?

How to we find the length in digits of an integer?

- Use logarithms
- \(1,000 = 10^3\), so \(\log_{10} 1,000 = 3\)
- \(10,000 = 10^4\), so \(\log_{10} 10,000 = 4\)
- The number of digits is one more than the \(\log_{10}\)
- What about non-even powers of 10?
  - \(1,000 < 3,162 < 10,000\), so \(\log_{10} 1,000 < \log_{10} 3,162 < \log_{10} 10,000\)
  - We need to round the logarithm down, then add 1

The Math class

- Java has many math functions in its standard library, including \texttt{floor} for rounding down, and \(\log_{10}\)
- Its full name is \texttt{java.lang.Math}
  - Classes in \texttt{java.lang} (unlike \texttt{java.util}) do not need an import
- Calculate the number of digits in an integer:

```java
final Scanner scnr = new Scanner(System.in);
System.out.print("Enter a natural number: ");
final int num = scnr.nextInt();
final long digitCount = 1+Math.round(Math.floor(Math.log10(num)));
System.out.println(num + " has " + digitCount + " digits");
return;
```

- Are we rounding twice?
- Why long?
  - * Use the Javadoc!
- Always address warnings!

2.6 Fibonacci numbers

What are the Fibonacci numbers

- The Fibonacci sequence starts with 1, then 1, then every subsequent number is the sum of the previous two.
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots
How would we design a Fibonacci calculator?

• OUTPUT a prompt

• INPUT num, to calculate the num-th Fibonacci number

• UPDATE (INITIALIZE)
  – The current Fibonacci number to 1
  – The previous Fibonacci number to 0

• ITERATE
  – Over values of n from 1 up, while n is less than num
  – ARITHMETIC add the current and previous Fibonacci numbers
  – UPDATE the previous and current Fibonacci numbers

• OUTPUT the current Fibonacci number

A Fibonacci calculator in Java

• OUTPUT a prompt

• INPUT num, to calculate the num-th Fibonacci number

• UPDATE (INITIALIZE)
  – Current to 1
  – Previous to 0

• ITERATE n from 1 up
  – While n < num
  – Add current and previous
  – UPDATE

• OUTPUT current

import java.util.Scanner;

public class Fibonacci {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("Which Fibonacci number? ");
        final int num = scnr.nextInt(); // 3
        long thisFib = 1; // Current Fib. num. // 4
        long prevFib = 0; // Previous Fib. num. // 5

        for(int n=1; n<num; n++) { // 6
            final long newFib = thisFib + prevFib; // 7
            prevFib = thisFib; // 8
            thisFib = newFib; // 9
        }
        System.out.println("It’s: " + thisFib); // 10
        return; // 11
    }
}
2.7 Fix that sentence

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;
  ```

Characters

- Strings are made of characters

- But characters and strings have completely different types!

  ```java
  final char capitalA = 'A';
  ```

- Strings contains zero or more characters, but a character itself is exactly one character

  ```java
  final String noXes = "";
  final String oneX = "x";
  final String threeXes = "xxx";
  final char theX = 'x';
  final char noCharX = ''; // Will cause an error!
  final char twoCharsX = 'xx'; // Will cause an error!
  ```

- Strings to characters: use the charAt method

  ```java
  final String alphabet = "abcdefghijklmnopqrstuvwxyz";
  final char fifth = alphabet.charAt(4);
  ```

- Characters to strings: use the String.valueOf method

  ```java
  final char letterE = 'e';
  final String letterEagain = String.valueOf(letterE);
  ```
What does this program do?

```java
import java.util.Scanner;

public class FixCaps {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in); // 1
        System.out.print("Tell me your sentence: "); // 2
        final String sentence = scnr.nextLine(); // 3
        final char firstCharacter = sentence.charAt(0); // 4
        System.out.print(Character.toUpperCase(firstCharacter)); // 5
        for(int i=1; i<sentence.length(); i++) { // 6
            final char thisChar = sentence.charAt(i); // 7
            System.out.print(Character.toLowerCase(thisChar)); // 8
        }
        System.out.println(); // 9
        return; // 10
    }
}
```

• Try it with HELLO!

2.8 Another kind of selection

Characters and `switch` statements

```java
System.out.print("Give me some letters! ");
final Scanner scnr = new Scanner(System.in);
final String letters = scnr.nextLine();
scnr.close();

final int len = letters.length();
int vowels = 0;
for(int i=0; i<len; i++) {
    final char letter = letters.charAt(i);
    switch (letter) {
    case 'a':
      vowels += 1;
      break;
    case 'e':
      vowels += 1;
      break;
    case 'i':
      vowels += 1;
      break;
    case 'o':
```
vowels += 1;
break;
case 'u':
vowels += 1;
break;
}
}
System.out.println("There were "+vowels+" vowels there.");

Many labels

System.out.print("Give me some letters! ");
final Scanner scnr = new Scanner(System.in);
final String letters = scnr.nextLine();
scnr.close();

final int len = letters.length();
int vowels = 0;
for(int i=0; i<len; i++) {
final char letter = letters.charAt(i);
switch (letter) {
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
vowels += 1;
break;
}
}
System.out.println("There were "+vowels+" vowels there.");

The default

System.out.print("Give me some letters! ");
final Scanner scnr = new Scanner(System.in);
final String letters = scnr.nextLine();
scnr.close();

final int len = letters.length();
int vowels=0, nonvowels=0;
for(int i=0; i<len; i++) {
final char letter = letters.charAt(i);
switch (letter) {
case 'a':
case 'e':
}
case 'i':
case 'o':
case 'u':
    vowels += 1;
    break;
default:
    nonvowels += 1;
}
}

System.out.println("There were 
    + vowels + " vowels and 
    + nonvowels + " non-vowels there.");

Conditional expressions

System.out.println("There were 
    + vowels 
    + " vowel"
    + (vowels == 1 ? "" : "s") 
    + " and " 
    + nonvowels 
    + " non-vowel"
    + (nonvowels == 1 ? "" : "s")
    + " there.");

2.9 Another kind of loop

Fix that sentence again

final Scanner scnr = new Scanner(System.in); // 1
System.out.print("Tell me your sentence: "); // 2
final String sentence = scnr.nextLine(); // 3

int i=0; // 4
while (i<sentence.length()) { // 5
    final char thisChar = sentence.charAt(i); // 6
    if (i == 0) { // 7
        System.out.print(Character.toUpperCase(thisChar)); // 8
    } else { // 9
        System.out.print(Character.toLowerCase(thisChar)); // 10
    }
}

System.out.println(); // 11
return; // 12
2.10 Multiple loops

What does this program do?

```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
}
```

Pitfalls with loop variables

- In a test

```java
for (int i=1; i<=6; i++) { // 1
    for (int j=1; i<=i; j++) { // 2
        System.out.print(i); // 3
    }
    System.out.println(); // 4
}
```

- In the post-loop effect

```java
for (int i=1; i<=6; i++) { // 5
    for (int j=1; j<=i; i++) { // 6
        System.out.print(i); // 7
    }
    System.out.println(); // 8
}
```

Two inner loops

```java
for (int i=1; i<=6; i++) { // 1
    for (int j=1; j<=(6-i); j++) { // 2
        System.out.print("-"); // 3
    }
    for (int j=1; j<=i; j++) { // 4
        System.out.print(i); // 5
    }
    System.out.println(); // 6
}
```

- What if we swap the two inner loops?
Commenting on loops
Document your loop with a high-level comment on the purpose of the loop, but don’t just repeat what the code already says.

• Good

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

• Bad

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

Another triangle
For example, for 5:

0
01
012
0123
01234
012345
01234
0123
012
01
0

• How can we generate this triangle?

Another triangle
For example, for 5:
final Scanner scnr = new Scanner(System.in);
System.out.print("Triangle size: ");
final int num = scnr.nextInt();
scnr.close();

for(int i=0; i<=num; i++) {
    for(int j=0; j<=i; j++) {
        System.out.print(j);
    }
    System.out.println();
}

for(int i=num-1; i>=0; i--) {
    for(int j=0; j<=i; j++) {
        System.out.print(j);
    }
    System.out.println();
}

• Can we do this with one loop?
  :BEAMER \texttt{env}: frame

What function will give the number of columns in these rows?

• Function graph

```
|     | 5     |
|  /  | /     |
|  /  | /     |
| /   | /     |
| /   | /     |
```

• Absolute value!
  – 5-|5-i|

• In Java, it’s Math.abs
public class SideTriangle {
    public static void main(String[] arg) {
        for(int i=0; i<=10; i++) {
            for(int j=0; j<=5-Math.abs(5-i); j++) {
                System.out.print(j);
            }
            System.out.println();
        }
    }
}

Another pattern

***********
***********
**........**
**........**
**........**
**........**
**........**
**........**
***********
***********

• How can we draw this pattern (in one pair of nested loops)?
  
  – Overall size of square
  – Size of border

Another pattern

final Scanner scnr = new Scanner(System.in);
System.out.print("Overall size: ");
final int length = scnr.nextInt();
System.out.print("Border thickness: ");
final int border = scnr.nextInt();
final int upperBorder = length-border;
scnr.close();

for(int i=0; i<length; i++) {
    for(int j=0; j<length; j++) {
        if (i<border || i>=upperBorder
            || j<border || j>=upperBorder) {
            System.out.print("\*\n");
        } else {
            ...
3 Methods

So what now?

• We’ve seen all of the Six Things a Program Can Do in action

• Now we move on to a way Java helps us organize
  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

3.1 Method basics

Remember the Fibonacci calculator

```java
import java.util.Scanner;

public class Fibonacci {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in); // 1
        System.out.print("Which Fibonacci number? "); // 2
        final int idx = scnr.nextInt(); // 3
        long thisFib = 1; // 4
        long prevFib = 0; // 5
        for(int n=1; n<idx; n++) { // 6
            final long newFib = thisFib + prevFib; // 7
            prevFib = thisFib; // 8
            thisFib = newFib; // 9
        }
        System.out.println("It’s: " + thisFib); // 10
        return; // 11
    }
}
```

A Fibonacci calculator method
import java.util.Scanner;

public class Fibonacci {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in); // 1
        System.out.print("Which Fibonacci number? "); // 2
        final int idx = scnr.nextInt(); // 3
        final long fib = getFibonacci(idx); // 4
        System.out.println("It’s: " + fib); // 5
        return; // 6
    }

    public static long getFibonacci(int i) { // 7
        long thisFib = 1; // 8
        long prevFib = 0; // 9
        for(int n=1; n<i; n++) { // 10
            final long newFib = thisFib + prevFib; // 11
            prevFib = thisFib; // 12
            thisFib = newFib; // 13
        }
        return thisFib; // 14
    }
}

Elements of a method

• A name — getFibonacci

• Parameters — int i
  – Any number, zero or more
  – Where we define the method, we have formal parameters
    public static long getFibonacci(int i) {
  – Where we use the method, we have actual parameters
    final long fib = getFibonacci(idx);
  – Argument is a synonym for parameter

• A return type — long

• A return value — return thisFib;
  – The return type can be void, in which case there’s no return value
Why do we create a method

- Break up the program into manageable units
- Reuse code from program to program
  - For example, the various Math methods
- Separate different aspects of the program
  - For example, separate user interaction from calculations

Names are local

- The way a name is used is *local* to methods
- If we assign something else to a name, the effect does not travel outside of the method

```java
public static void main(String[] args) {
    int y = 40;
    f(y);
    System.out.println(y);
}

public static void f(int x) {
    x = 5; // Applies only to f’s variable x
    return;
}
```

3.2 A method for leap years

Reaching way back — calculating a leap year

```java
import java.util.Scanner;

public class LeapYear {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("What year? ");
        final int year = scnr.nextInt();

        if (year % 400 == 0) {
            System.out.println(year + " was a leap year!");
        } else if (year % 100 == 0) {
            System.out.println(year + " was not a leap year.");
        } else if (year % 4 == 0) {
            System.out.println(year + " was a leap year!");
        } else {
            System.out.println(year + " was not a leap year.");
        }
    }
}
```
• Note that we need the else statements to keep from printing multiple messages

### A leap year method

```java
class LeapYear {
    public static boolean isLeapYear(int year) {
        if (year % 400 == 0) { return true; }
        if (year % 100 == 0) { return false; }
        if (year % 4 == 0) { return true; }
        return false;
    }
}
```

```java
class LeapYear {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("What year? ");
        final int year = scnr.nextInt();
        final boolean isLeap = isLeapYear(year);
        if (isLeap) {
            System.out.println(year + " was a leap year!");
        } else {
            System.out.println(year + " was not a leap year.");
        }
    }
}
```

• Since return alters the flow of control, we don’t need the else’s here

### 3.3 A choose method

#### The choose calculator

How would we write a method for the n-choose-m function?

```java
class Choose {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("How many total objects? ");
        final int total = scnr.nextInt();
        System.out.print("Choose how many? ");
        final int choose = scnr.nextInt();
        long ways = 1;
        int totalFactor = total;
        for(int i=1; i<=choose; i++) {
```
ways = ways * totalFactor / i;
--totalFactor;

System.out.println(ways + " ways to choose");
return;
}

A choose method

public class Choose {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("How many total objects? ");
        final int total = scnr.nextInt();
        System.out.print("Choose how many? ");
        final int choose = scnr.nextInt();
        System.out.println(nChooseM(total,choose) + " ways to choose");
        return;
    }

    public static long nChooseM(int total, int choose) {
        long ways = 1;
        int totalFactor = total;
        for(int i=1; i<=choose; i++) {
            ways = ways * totalFactor / i;
            --totalFactor;
        }

        return ways;
    }

}

Length methods

How would we write a method characterLength for the length of an integer in digits?

public static int characterLength(long num) {
    return 1+Math.floor(Math.log10(num));
}

public static int characterLength(String str) {
    return str.length();
}

3.4 Factorial and Fibonacci methods

Reaching back again
import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("Factorial of: ");
        final int num = scnr.nextInt();
        long product = 1;

        for(int n=2; n<=num; n++) {
            product *= n;
        }

        System.out.println("It’s: "+product);

        return;
    }
}

We make everything into a method now

public static long factorial(int num) {
    long product = 1;

    for(int n=2; n<=num; n++) {
        product *= n;
    }

    return product;

}

The recursive way

public static long factorial(int num) {
    if (num <= 1) {
        return 1;
    } else {
        return num * factorial(num-1);
    }
}

A first idea for recursive Fibonacci

public static long fib(int num) {
    if (num <= 0) {
        return 0;
    } else if (num == 1) {
        return 1;
    } else {
        return fib(num-1) + fib(num-2);
    }
}
) else {
    return fib(num-1) + fib(num-2);
}
}

Running the first idea
In the recursive implementation for computing the $n$th Fibonacci number with $n = 4$, how many recursive calls are made?

Calls we’d make
fib(5) calls first fib(3), then fib(4)
  • fib(3) calls first fib(1), then fib(2)
    - fib(1) is a base case
    - fib(2) calls first fib(0), then fib(1)
      * fib(0) is a base case
      * fib(1) is a base case
  • fib(4) calls first fib(2), then fib(3)
    - fib(2) calls first fib(0), then fib(1)
      * fib(0) is a base case
      * fib(1) is a base case
    - fib(3) calls first fib(1), then fib(2)
      * fib(1) is a base case
      * fib(2) calls first fib(0), then fib(1)
        * fib(0) is a base case
        * fib(1) is a base case
      * fib(0) is a base case
      * fib(1) is a base case

That’s 15 calls
  • Lots of repeated work!
  • Is recursion a bad approach for Fibonacci?
    – Or is this way of structuring recursion bad?

Fibonacci with a loop

public static long
    fib(final int num) {
        long fib=0;
        long next=1;

        for(int n=0; n<num; n++) {
            final long

newNext=fib+next;
fib = next;
next = newNext;
}

return thisFib;
}

• The loop runs (about) num times, so we should be able to find a way that recurs num times

• Note how we use next and fib
  – Both calculated on each pass through the loop
  – Both preserved from one pass through the loop to the next

• How can we provide both from one recursive call to the next?
  – By passing both as parameters!

Recursion with accumulating parameters

• Instead of calculating the result in a method body after the return of a recursive call,

• Calculate the result in the arguments of the call

static long fibHelper(int n, long fib, long next) {
  if (n<1) {
    return fib;
  } else {
    return fibHelper(n-1, next, fib+next);
  }
}

public static long fib(int n) {
  return fibHelper(n, 0, 1);
}

3.5 Factorial and Fibonacci methods

Recursion with accumulating parameters

• Instead of calculating the result in a method body after the return of a recursive call,

• Calculate the result in the arguments of the call

static long fibHelper(int n, long fib, long next) {
  if (n<1) {
    return fib;
  } else {
    return fibHelper(n-1, next, fib+next);
  }
public static long fib(int n) {
    return fibHelper(n, 0, 1);
}

3.6 Methods and your work

Projects going forward

• Will specify methods you should write
• Be sure that you match the method name, and the number and types of arguments
• Use the main method to show how you’ve been testing the other methods as you develop
  – Not necessarily just the required methods
  – If you decompose the problem further, test those methods too
  – I’ll look at the quality of your tests when assessing

Documenting methods

• The code in these slides do not often have comments
  – There’s usually no room for them
  – And our discussion of the code serves the same purpose
• But it’s important to document code
  – And it counts for part of your grade
• Let’s document bubbleSort as an example

    public static void bubbleSort(int[] array) {
        final int len = array.length;
        for(int a=1; a<len; a++) {
            for(int b=len-1; a<=b; --b) {
                if (array[b-1] > array[b]) {
                    final int tmp = array[b-1];
                    array[b-1] = array[b];
                    array[b] = tmp;
                }
            }
        }
    }
Inline comments for the details of the algorithm

• The beginning of a loop is almost always interesting enough to warrant a comment

  // "a" moves from the left of the array forward; we push the a^th
  // smallest element of the array to the (a-1)^st place
  for(int a=1; a<howMany; a++) {
  // We're explaining why we're looping and what a represents
  * Not necessary to point out that a loops from 1 to an upper bound; we explain instead that
    we're moving across the array
  // We're also explaining the purpose of this loop
  
  • Similarly for the inner loop

  // To push an element leftwards, we start from the right side,
  // and move leftward doing comparisons. b is the position we
  // will compare next.
  for(int b=howMany-1; a<=b; --b) {
  // We're also explaining the purpose of this loop

Inline comments for the details of the algorithm

• Decisions like if statements and switch blocks are also usually interesting enough to document

• Again, explain the significance of the decision; don’t just rephrase Java into tedious English

  // Check if two elements are out of order, and if so, swap
  // them.
  if (array[b-1] > array[b]) {
  // Here, it makes sense to explain the body of the if as well
  // Sometimes, the body of the if will be complicated enough to warrant several inline comments
    along the way

Javadoc comments for the method itself

/**
 * Sorts an array of integers.
 * @param array The array to be sorted.
 *
 * This routine works only on integer arrays, not String arrays.
 */
public static void bubbleSort(int[] array) {

• A Javadoc comment always starts /***, and the standard style is to have the vertical line of single
  asterixes

• The first lines should be a short verb clause summarizes very briefly what the method does
• Then document the parameters with @param and the return value (if there’s one) with @returns
• Next give a fuller description of the operation, limits, efficiency, etc. of the method
• Eclipse will generate the Javadoc of your code
• For examples, see the standard Java API

Testing methods
• Structuring code with methods can make testing your programs easier
• To now:
  – Every different test of a program required manually running it, changing the parameters each time
  – Hard to remember the whole suite of different values, hard to keep track of the "right" answer for each case
• But now:
  – The interesting code can be in named methods
  – We can use the main method to test the other methods

4 Arrays
4.1 Array basics
So what now?
• We’ve seen all of the Six Things a Program Can Do in action
• Now we move on to the four ways 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 seen mostly 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 arrays

```java
public class UseAnArray {
    public static void main(String[] argv) {
        int[] numbers = { 1, 2, 3, 4 }; // 1

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

- Index from 0
- 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

We can update the contents as well

```java
public class ChangeAnArray {
    public static void main(String[] argv) {
        int[] numbers = { 1, 2, 3, 4 };

        for(int i=0; i<numbers.length; i++) {
            numbers[i] *= 2;
            System.out.println(numbers[i]);
        }
    }
}
```

A notation for reading through the contents

```java
public class UseArrayElements {
    public static void main(String[] argv) {
        int[] numbers = { 1, 2, 3, 4 };
        for(final int number : numbers) {
            System.out.println(number);
        }
    }
}
```

- But this only works for reading the array elements.
- We couldn’t change them this way.
Changing to a new array

```java
int[] numbers = { 1, 2, 3, 4 };;
for(final int number : numbers) {
    System.out.println(number);
}

numbers = new int[] { 5, 6, 7 };
for(final int number : numbers) {
    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

  numbers.length = 2;
```

Arrays of any type

```java
public class UseStringArray {
    public static void main(String[] argv) {
        String[] names = { "Hello", "Goodbye" };
        for(final String name : names) {
            System.out.println(name);
        }
    }
}
```

What does final mean?
Declaring an array variable final rules out our example of resetting the array variable:

```java
public class BadUseOfArrays {
    public static void main(String[] argv) {
        final int[] numbers = { 1, 2, 3, 4 };
        for(final int number : numbers) {
            System.out.println(number);
        }

        // This line gives an error
        numbers = new int[] { 5, 6, 7 };
        for(final int number : numbers) {
            System.out.println(number);
        }
    }
}
```
But we can still change the contents

```java
public class ChangeFinalArray {
    public static void main(String[] argv) {
        final int[] numbers = { 1, 2, 3, 4 };

        for(int i=0; i<numbers.length; i++) {
            numbers[i] *= 2;
            System.out.println(numbers[i]);
        }
    }
}
```

- `final` applies only to the binding to the name itself, not to operations we might perform via the name.

### 4.2 Combining two arrays: the dot product

**Combining two arrays: the dot product**

- It’s a math thing, an operation on vectors

  - \( \vec{u} \cdot \vec{v} = \sum_i u_i v_i \)

    - For two vectors of equal length, multiply corresponding elements, and return the sum

- Example: \([1,2,3] \cdot [10,1000,100000]\)

**How can we implement the dot product?**

```java
public class DotProduct {
    public static void main(String[] args) {
        final int[] array1 = { 2, 4, 6 }; // 1
        final int[] array2 = { 100, 1000, 10000 }; // 2

        return; //
    }
}
```
Dot product

```java
class DotProduct {
    public static void main(String[] args) {
        final int[] array1 = { 2, 4, 6 }; // 1
        final int[] array2 = { 100, 1000, 10000 }; // 2

        int total = 0; // 3
        for(int i=0; i<array1.length; ++i) { // 4
            total += array1[i] * array2[i]; // 5
        }

        System.out.println("Dot product is: " + total); // 6
        return; // 7
    }
}
```

- Tracing execution

4.3 Sorting an array

What does this program do?

```java
final Scanner scnr = new Scanner(System.in); // 1
System.out.print("How many numbers? "); // 2
final int howMany = scnr.nextInt(); // 3
final int[] numbers = new int[howMany]; // 4
for(int i=0; i<howMany; i++) { // 5
    System.out.print("Number "+1+i+": "); // 6
    numbers[i] = scnr.nextInt(); // 7
}
for(int a=1; a<howMany; a++) { // 8
    for(int b=howMany-1; a<=b; --b) { // 9
        if (numbers[b-1] > numbers[b]) { // 10
            final int tmp = numbers[b-1]; // 11
            numbers[b-1] = numbers[b]; // 12
            numbers[b] = tmp; // 13
        }
    }
}
for(int i=0; i<howMany; i++) { System.out.print(numbers[i] + " "); } // 14
System.out.println(); // 15
return;
```
The first part is more straightforward

```java
final Scanner scnr = new Scanner(System.in); // 1
System.out.print("How many numbers? "); // 2
final int howMany = scnr.nextInt(); // 3

final int[] numbers = new int[howMany]; // 4
for(int i=0; i<howMany; i++) { // 5
    System.out.print("Number "+1+i+": "); // 6
    numbers[i] = scnr.nextInt(); // 7
}
```

And the last part just prints the array:

```java
for(int i=0; i<howMany; i++) { System.out.print(numbers[i] + " "); } // 14
System.out.println(); // 15
```

But what about these loops?

```java
for(int a=1; a<howMany; a++) { // 8
    for(int b=howMany-1; a<=b; --b) { // 9
        if (numbers[b-1] > numbers[b]) { // 10
            final int tmp = numbers[b-1]; // 11
            numbers[b-1] = numbers[b]; // 12
            numbers[b] = tmp; // 13
        }
    }
}
```

- Look at just the if-statement at Line 10
  - 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
final Scanner scnr = new Scanner(System.in); // 1
System.out.print("How many numbers? "); // 2
final int howMany = scnr.nextInt(); // 3
```
4.4 The Sieve of Eratosthenes

**How do we work out if numbers are prime?**

- Write out the numbers we’re interested in testing for primality
How do we work out if numbers are prime?

• Write out the numbers we’re interested in testing for primality

• 1 is not a prime, so scratch it out

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

How do we work out if numbers are prime?

• Write out the numbers we’re interested in testing for primality

• 1 is not a prime, so scratch it out

• Look at the lowest unmarked number — mark it as prime

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>
How do we work out if numbers are prime?

• Write out the numbers we’re interested in testing for primality

• 1 is not a prime, so scratch it out

• Look at the lowest unmarked number — mark it as prime

• But strike out its multiples — they’re definitely not prime

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

How do we work out if numbers are prime?

• Write out the numbers we’re interested in testing for primality

• 1 is not a prime, so scratch it out

• Look at the lowest unmarked number — mark it as prime

• But strike out its multiples — they’re definitely not prime

• And so on with the new lowest unmarked number

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>
• But strike out its multiples — they’re definitely not prime
• And so on with the new lowest unmarked number, and so on

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

**How do we work out if numbers are prime?**

• Write out the numbers we’re interested in testing for primality
• 1 is not a prime, so scratch it out
• Look at the lowest unmarked number — mark it as prime
• But strike out its multiples — they’re definitely not prime
• And so on with the new lowest unmarked number, and so on, and so on

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
</tr>
<tr>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
</tr>
<tr>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
<tr>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

**How do we work out if numbers are prime?**

• Write out the numbers we’re interested in testing for primality
• 1 is not a prime, so scratch it out
• Look at the lowest unmarked number — mark it as prime
• But strike out its multiples — they’re definitely not prime
• And so on with the new lowest unmarked number, and so on, and so on, and so on
So let’s code that up

The Sieve

```java
System.out.println("Find the primes up through what number? ");
final Scanner scnr = new Scanner(System.in);
final int upThrough = scnr.nextInt();
scnr.close();

final boolean[] isPrime = new boolean[1+upThrough];

isPrime[0] = false;
isPrime[1] = false;
for(int i=2; i<=upThrough; ++i) {
    isPrime[i] = true;
}
```

The Sieve

```java
System.out.println("Find the primes up through what number? ");
final Scanner scnr = new Scanner(System.in);
final int upThrough = scnr.nextInt();
scnr.close();

final boolean[] isPrime = new boolean[1+upThrough];

isPrime[0] = false;
isPrime[1] = false;
for(int i=2; i<=upThrough; ++i) {
    isPrime[i] = true;
}
```
System.out.print("Find the primes up through what number? ");
final Scanner scnr = new Scanner(System.in);
final int upThrough = scnr.nextInt();
scnr.close();

final boolean[] isPrime = new boolean[1+upThrough];

isPrime[0] = false;
isPrime[1] = false;
for(int i=2; i<=upThrough; ++i) {
    isPrime[i] = true;
}

for(int low=0; low<=upThrough; ++low) {
    if (isPrime[low]) {
        System.out.println(low + " is prime.");
    }
}

The Sieve

System.out.print("Find the primes up through what number? ");
final Scanner scnr = new Scanner(System.in);
final int upThrough = scnr.nextInt();
scnr.close();

final boolean[] isPrime = new boolean[1+upThrough];

isPrime[0] = false;
isPrime[1] = false;
for(int i=2; i<=upThrough; ++i) {
    isPrime[i] = true;
}

for(int low=0; low<=upThrough; ++low) {
    if (isPrime[low]) {
        System.out.println(low + " is prime.");
    }
}

The Sieve

System.out.print("Find the primes up through what number? ");
final Scanner scnr = new Scanner(System.in);
final int upThrough = scnr.nextInt();
scnr.close();

final boolean[] isPrime = new boolean[1+upThrough];

isPrime[0] = false;
isPrime[1] = false;
for(int i=2; i<=upThrough; ++i) {
    isPrime[i] = true;
}
for(int low=0; low<=upThrough; ++low) {
    if (isPrime[low]) {
        System.out.println(low + " is prime.");
        for(int nonPrime=low+low; nonPrime<=upThrough; nonPrime += low) {
            isPrime[nonPrime] = false;
        }
    }
}

4.5 Two-dimensional arrays

Two-dimensional arrays

• So far we’ve used arrays with a single index — called one-dimensional.

• But we can have any number of indices in a matrix:

    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
Uneven arrays

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

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

- Some true expressions:

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

- Must check the length of each inner array when traversing

4.6 Lining up columns of information

Remember the wordsWeKnow matrix

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

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

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

What do we need to think about?

What’s involved with lining up the columns?

- Every column the same width, or each column padded separately?
  
  - We’ll do both, first every column the same
• Need to look at the width of every word before we know the width of any column

• Since some rows have more columns than others, we need to look at all of the rows to know how many columns there are

• Must add the right amount of spaces to pad each string

Every column the same width

// Find the width

// For each row and column

// Add padding for this string

// Print this string

}

Every column the same width

int width=0;
for(final String[] row : wordsWeKnow) {
    for(final String word : row) {
        final int thisWidth = word.length();
        if (thisWidth>width) {
            width=thisWidth;
        }
    }
}

// For each row and column

// Add padding for this string
// Print this string

)

Every column the same width

int width=0;
for(final String[] row : wordsWeKnow) {
    for(final String word : row) {
        final int thisWidth = word.length();
        if (thisWidth>width) {
            width=thisWidth;
        }
    }
}

for(final String[] row : wordsWeKnow) {
    for(final String word : row) {
        // Add padding for this string

        // Print this string

    }
}

Every column the same width

int width=0;
for(final String[] row : wordsWeKnow) {
    for(final String word : row) {
        final int thisWidth = word.length();
        if (thisWidth>width) {
            width=thisWidth;
        }
    }
}

for(final String[] row : wordsWeKnow) {
    for(final String word : row) {
        final int pad = width - word.length();
        for(int i=0; i<pad; i++) {
            System.out.print(" ");
        }
    }
}


// Print this string


Every column the same width

int width=0;
for(final String[] row : wordsWeKnow) {
    for(final String word : row) {
        final int thisWidth = word.length();
        if (thisWidth>width) {
            width=thisWidth;
        }
    }
}

for(final String[] row : wordsWeKnow) {
    for(final String word : row) {
        final int pad = width - word.length();
        for(int i=0; i<pad; i++) {
            System.out.print(" ");
        }
        System.out.print(word);
        System.out.print(" ");
    }
    System.out.println();
}

Every column the minimum width

• What’s our algorithm here?

Every column the minimum width

// Allocate space for the number of columns
// For each row {

    // Check the width of this row, and maybe update
    // the columns count

} // Allocate space for the widths array // For each row and column entry {
// Check this column’s width, and maybe
// update the maximum

} // For each row and column entry {

// Print the padding for this column

// Print this entry

} System.out.println();

} // For each row and column entry {

Every column the minimum width

int cols = 0;
// For each row {

    // Check the width of this row, and maybe update
    // the columns count

} // For each row {

// Allocate space for the widths array
// For each row and column entry {

    // Check this column’s width, and maybe
    // update the maximum

} // For each row and column entry {
// Print the padding for this column

// Print this entry
}
System.out.println();
}

Every column the minimum width

int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    // Check the width of this row, and maybe update
    // the columns count
}

// Allocate space for the widths array
// For each row and column entry {

    // Check this column’s width, and maybe
    // update the maximum

}

// For each row and column entry {

    // Print the padding for this column

    // Print this entry
}
System.out.println();
}

Every column the minimum width

int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }
}

// Allocate space for the widths array
// For each row and column entry {

    // Check this column’s width, and maybe
    // update the maximum

} // For each row and column entry {

    // Print the padding for this column

    // Print this entry

} System.out.println();

Every column the minimum width

int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }
}

int[] widths = new int[cols];
// For each row and column entry {
/ Check this column’s width, and maybe 
/ update the maximum

// For each row and column entry {

// Print the padding for this column

// Print this entry

} System.out.println();

Every column the minimum width

int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }
}

int[] widths = new int[cols];
for (int row=0; row<wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col=0; col<rowArray.length; col++) {
        final int thisWidth = rowArray[col].length();

        // Check this column’s width, and maybe
        // update the maximum
    }
}

// For each row and column entry {

// Print the padding for this column
// Print this entry
}
System.out.println();
}

Every column the minimum width

int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }
}

int[] widths = new int[cols];
for (int row=0; row<wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col=0; col<rowArray.length; col++) {
        final int thisWidth = rowArray[col].length();
        // Check this column’s width
    }
}

// For each row and column entry {

    // Print the padding for this column

    // Print this entry

} System.out.println();
}

Every column the minimum width

int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }

}
int[] widths = new int[cols];
for (int row=0; row<wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col=0; col<rowArray.length; col++) {
        final int thisWidth = rowArray[col].length();
        if (widths[col] < thisWidth) {
            widths[col] = thisWidth;
        }
    }
}

// For each row and column entry {

    // Print the padding for this column

        // Print this entry

    }
System.out.println();
}

Every column the minimum width

int cols = 0;
for (int row=0; row<wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }
}

int[] widths = new int[cols];
for (int row=0; row<wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col=0; col<rowArray.length; col++) {
        final int thisWidth = rowArray[col].length();
        if (widths[col] < thisWidth) {
            widths[col] = thisWidth;
        }
    }
}
for (int row = 0; row < wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col = 0; col < rowArray.length; col++) {
        final String word = rowArray[col];

        // Print the padding for this column

        // Print this entry
    }
    System.out.println();
}

Every column the minimum width

int cols = 0;
for (int row = 0; row < wordsWeKnow.length; row++) {
    final int thisWidth = wordsWeKnow[row].length;
    if (thisWidth > cols) {
        cols = thisWidth;
    }
}

int[] widths = new int[cols];
for (int row = 0; row < wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col = 0; col < rowArray.length; col++) {
        final int thisWidth = rowArray[col].length();
        if (widths[col] < thisWidth) {
            widths[col] = thisWidth;
        }
    }
}

for (int row = 0; row < wordsWeKnow.length; row++) {
    final String[] rowArray = wordsWeKnow[row];
    for (int col = 0; col < rowArray.length; col++) {
        final String word = rowArray[col];
        final int pad = widths[col] - word.length();
        for (int i = 0; i < pad; i++) {
            System.out.print(" ");
        }
        System.out.print(word);
        System.out.print(" ");
    }
    System.out.println();
}
4.7 Pulling words out of a line

Pulling words out of a line

// Read in a line of text.
final Scanner scanner = new Scanner(System.in);
System.out.print("Enter a line of text: ");
final String sourceLine = scanner.nextLine();
final int sourceLength = sourceLine.length();

// Read how many words we should take from the line.
System.out.print("Extract how many words? ");
final int wordCount = scanner.nextInt();

// Prepare an array to hold the words.
final String[] words = new String[wordCount];
// We’ll scan through the line looking for spaces.
int position = 0;
// The word we’ll fill in next.
int nextWord = 0;

Pulling words out of a line

// Pull words from the line.
while (position<sourceLength && nextWord<wordCount) {

    // Skip over any spaces.
    while (position<sourceLength && sourceLine.charAt(position) == ' ') {
        ++position;
    }
    final int wordStart = position;

    // Now skip over non-spaces.
    while (position<sourceLength && sourceLine.charAt(position) != ' ') {
        ++position;
    }
    final int wordEnd = position;

    // Copy that word into the array.
    words[nextWord] = sourceLine.substring(wordStart, wordEnd);
    ++nextWord;
}

• What happens on a "normal" run?
• What happens when we ask for too many words?
Using the words

```java
final Scanner scanner = new Scanner(System.in);
System.out.print("Enter a line of text: ");
final String sourceLine = scanner.nextLine();
final int sourceLength = sourceLine.length();

System.out.print("Extract how many words? ");
final int wordCount = scanner.nextInt();

final String[] words = new String[wordCount];
int position = 0;
int nextWord = 0;

while (position<sourceLength
    && nextWord<wordCount) {
    while (position<sourceLength
        && sourceLine.charAt(position)==' ') {
        ++position;
    }
    final int wordStart = position;

    while (position<sourceLength
        && sourceLine.charAt(position)!=' ') {
        ++position;
    }
    final int wordEnd = position;

    words[nextWord] = sourceLine
        .substring(wordStart, wordEnd);
    ++nextWord;
}
```

Printing the array

First try

```java
final int found = words.length;
System.out.println("Found " + found + " words");
for(int i=0; i<found; i++) {
    System.out.println(i + ". " + words[i]);
}
```

• But what if there were fewer words in the line than the user asked us to pull?

A good change

```java
final int found = nextWord;
```
4.8 Some random stuff

Let's pick some numbers

```java
import java.util.Random;

public class RandomDistribs {
    public static void main(String[] args) {
        // Get a source of random numbers
        final Random src = new Random();
        final int MAX_PICK = 100;

        // Pick 10,001 random integers, each between 0 and
        // 99 (inclusive)
        final int[] someInts = new int[10001];
        for (int i = 0; i < someInts.length; i++) {
            someInts[i] = src.nextInt(MAX_PICK);
        }

        • So what can we do with 10,001 random numbers?

        We can find their average

        double total = 0.0;
        for (int i = 0; i < someInts.length; i++) {
            total += someInts[i];
        }
        final double average = total / someInts.length;
        System.out.println("Average of the numbers is " + average);

        We count how many there are of each

        final int[] census = new int[MAX_PICK];
        for (int i = 0; i < MAX_PICK; i++) {
            census[i] = 0;
        }
        for (int i = 0; i < someInts.length; i++) {
            census[someInts[i]] += 1;
        }

        We can find their median

        int finderTotal = 0;
        int finderIndex = 0;
        while (finderTotal < 5000) {
            finderTotal += census[finderIndex];
            finderIndex += 1;
        }
        System.out.println("Median value is " + finderIndex);
    }
}
```
We can find their mode

```java
int mode=0;
int modeFreq = census[mode];
for(int i=1; i<MAX_PICK; i++) {
    final int thisFreq = census[i];
    if (thisFreq>modeFreq) {
        mode=i;
        modeFreq=thisFreq;
    }
}
System.out.println("Mode is " + mode + " (" + modeFreq + " occurrences");
```

We can find their standard deviation

Formula: \( \sigma = \sqrt{\frac{\sum (x-\bar{x})^2}{N-1}} \)

```java
int sumOfSq=0;
for(int i=0; i<someInts.length; i++) {
    final double diff = someInts[i] - average;
    sumOfSq += diff*diff;
}
final double stdDev = Math.sqrt(sumOfSq/10000);
```

4.9 Methods and arrays

Passing arrays to methods

- Recall: the way a name is used is local to methods
  - If we assign something else to a name, the effect does not travel outside of the method
- But the internals of arrays are not duplicated for function calls

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

public static void f(int[] x) {
    if (x.length > 1) {
        x[1] = -1;
    }
    return;
}
```
• 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

**Recall bubble sort**

```java
public static void main(String[] args) {
    final Scanner scnr = new Scanner(System.in); // 1
    System.out.print("How many numbers? "); // 2
    final int howMany = scnr.nextInt(); // 3

    final int[] numbers = new int[howMany]; // 4
    for(int i=0; i<howMany; i++) { // 5
        System.out.print("Number " + 1+i + ": "); // 6
        numbers[i] = scnr.nextInt(); // 7
    }

    for(int a=1; a<howMany; a++) { // 8
        for(int b=howMany-1; a<=b; --b) { // 9
            if (numbers[b-1] > numbers[b]) { // 10
                final int tmp = numbers[b-1]; // 11
                numbers[b-1] = numbers[b]; // 12
                numbers[b] = tmp; // 13
            }
        }
    }

    for(int i=0; i<howMany; i++) { System.out.print(numbers[i] + ": "); } // 14
    System.out.println(); // 15
}
```

**Bubble sort as a method**

```java
public static void main(String[] args) {
    final Scanner scnr = new Scanner(System.in); // 1
    System.out.print("How many numbers? "); // 2
    final int howMany = scnr.nextInt(); // 3

    final int[] numbers = new int[howMany]; // 4
    for(int i=0; i<howMany; i++) { // 5
        System.out.print("Number "+1+i+": "); // 6
        numbers[i] = scnr.nextInt(); // 7
    }

    for(int a=1; a<howMany; a++) { // 8
        for(int b=howMany-1; a<=b; --b) { // 9
            if (numbers[b-1] > numbers[b]) { // 10
                final int tmp = numbers[b-1]; // 11
                numbers[b-1] = numbers[b]; // 12
                numbers[b] = tmp; // 13
            }
        }
    }

    for(int i=0; i<howMany; i++) { System.out.print(numbers[i] + ": "); } // 14
    System.out.println(); // 15
}
```
bubbleSort(numbers); // 8

for(int i=0; i<howMany; i++) { System.out.print(numbers[i] + " "); } // 9
System.out.println(); // 10

return; // 11
}

public static void bubbleSort(int[] array) {
    final int len = array.length; // 12
    for(int a=1; a<len; a++) { // 13
        for(int b=len-1; a<=b; --b) { // 14
            if (array[b-1] > array[b]) { // 15
                final int tmp = array[b-1]; // 16
                array[b-1] = array[b]; // 17
                array[b] = tmp; // 18
            }
        }
    }
    return;
}

The dot product program

public class DotProduct {
    public static void main(String[] args) {
        final int[] array1 = { 2, 4, 6};
        final int[] array2 = { 100, 1000, 10000};

        int total = 0;
        for(int i=0; i<array1.length; ++i) {
            total += array1[i] * array2[i];
        }

        System.out.println("Dot product is: " + total);
        return;
    }
}

The dot product method

public class DotProduct {
    public static void main(String[] args) {
        final int[] array1 = { 2, 4, 6};
        final int[] array2 = { 100, 1000, 10000};
final int dprod = dotProduct(array1, array2);
System.out.println("Dot product is: "+ dprod);
return;
}

public static int dotProduct(int[] array1, int[] array2) {
    int total = 0;
    for (int i = 0; i < array1.length; ++i) {
        total += array1[i] * array2[i];
    }
    return total;
}

Find an integer in a sorted array

public static int findMatch(int[] numbers, int target) {
    return findMatch(numbers, target, 0, numbers.length);
}

public static int findMatch(int[] numbers, int target, int lowIdx, int upperBnd) {
    // Midpoint of low and high values
    final int midPoint = (upperBnd + lowIdx) / 2; // 1

    // The value we find at the midpoint
    final int foundVal = numbers[midPoint]; // 2

    if (target == foundVal) { // 3
        // Base case 1: item found at midPoint position
        return midPoint; // 4
    } else if (upperBnd - lowIdx == 1) { // 5
        // Base case 2: match not found
        return -1; // 6
    } else if (target < foundVal) { // 7
        // Recursive case: the target is less than midPoint, so it’s
        // in the lower half
        return findMatch(numbers, target, lowIdx, midPoint); // 8
    } else { // 9
        // Recursive case: the target is greater than midPoint, so it’s
        // in the upper half
        return findMatch(numbers, target, midPoint + 1, upperBnd);
    }
}

• Search for 20 in {5, 20, 53, 61, 67, 78, 85, 90}
5 Classes and objects

5.1 Class and object basics

The six things and the four ways

• 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

• Four ways Java will help you organize your work
  1. Grouping related data together
  2. Defining sequences of operations
  3. Associating data with operations relevant to the particular data
  4. Naming these groups, sequences and associations for easy and repeated use

• We’ve seen arrays and methods

• One last Java structure: our own classes

Classes and objects

A class is a sort of thing; an object is one instance of a class

• String and Scanner are two classes we’ve seen already
• "Hello" and "Goodbye" are two different String objects
  – There’s a special syntax for creating a String

• Ordinarily we create a new object of a class using new

final Scanner scn = new Scanner();

Defining a class

Name

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

Fields - storage for simple values

• The state of each object of the class
Methods - code to be executed

- We've already written static methods
  - Separate from objects of the class
- Object methods are linked to the context of one object
  - Like how `charAt` refers to the characters of a particular string

A class for student records
The university needs certain information about every student
- Their name
- Their ID
- Their current classes
- The number of credits earned

Creating fields
We can turn that information into field declarations
- Their name
  - `String firstNames, lastName;`
- Their ID
  - `int id;`
- Their current classes
  - `String[] currentClasses;`
- The number of credits earned
  - `int creditsEarned;`

Starting a student record class

```java
public class Student {
    private String firstNames, lastName;
    private int id;
    private String[] currentClasses;
    private int creditsEarned;

    // If a field should never be changed, we can tag it as final
    public class Student {
        private String firstNames, lastName;
        private final int id;
        private String[] currentClasses;
        private int creditsEarned;
    }
```
What operations might we need to perform on student records?

- Get the name, or id, or current classes, etc.
- Update the name
- Report passed classes
- Enroll in new classes

These all become different methods

**Accessor or "get" methods**

Some methods give us information

```java
public String getFirstNames() {
    return firstNames;
}

public String getLastName() {
    return lastName;
}

public int getId() {
    return id;
}
```

and so on

- Accessors often take no arguments
- Accessors return a value

**Mutator methods**

Some methods change the state of the object

The simplest mutators are *setters*, which just update a field

```java
public void setFirstNames(final String newNames) {
    firstNames = newNames;
}

public void setLastName(final String newLastName) {
    lastName = newLastName;
}
```

Other mutators can have more complicated changes

```java
public void awardCreditHours(final int credits) {
    creditsEarned = creditsEarned + credits;
}
```

- Mutators usually take arguments
- Mutators usually are *void*
Starting it all up
There is one more bit of code associated with a class — the constructor

- Constructors set up a new object
- Always run right away for a new instance of the class, and then never again
- Constructors have the same name as their class
- May also have parameters, and have a body of statements like a method
- Unlike a method, they do not have an explicit return type
- And they do not actually return anything — they just set up the space which new allocates
- Like filling in the pages of a book

Constructing a student record

```java
public Student(String fnames, String lname, int iden) {
    firstNames = fnames;
    lastName = lname;
    id = iden;
    currentClasses = new String[0];
    creditsEarned = 0;
}
```

- Fields can be set from a parameter, or to a constant
- Fields marked final must be set when the object is set up

The whole thing

```java
public class Student {
    private String firstNames, lastName;
    private final int id;
    private String[] currentClasses;
    private int creditsEarned;

    public Student(String fnames, String lname, int iden) {
        firstNames = fnames;
        lastName = lname;
        id = iden;
        currentClasses = new String[0];
        creditsEarned = 0;
    }

    public String getFirstNames() {
        return firstNames;
    }
    public String getLastName() {
return lastName;
}
public int getId() {
    return id;
}

public void setFirstNames(final String newNames) {
    firstNames = newNames;
}
public void setLastName(final String newLastName) {
    lastName = newLastName;
}

public void awardCreditHours(final int credits) {
    creditsEarned = creditsEarned + credits;
}

• Stay organized! We write
  – Fields first
  – Then the constructor
  – Then accessors
  – Then mutators
• We label fields private
  – We’ll see what that means, and what alternatives there are, later
• Two kinds of documentation:
  – Javadoc comments
    * Produce the same kind of HTML pages that we’ve looked at for String, Math, etc.
    * Start with /**, end with */
    * Describe the inputs and outputs of each method, and its overall purpose
  – 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

5.2 A class with one single field
A small class

public class SmallClass {
    private final String label;

    public SmallClass(final String label) {

this.label = label;
}

public String getLabel() {
    return label;
}

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

A main routine for SmallClass

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

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

5.3 Complex numbers

Complex number basics
    Imaginary numbers are the square roots of negative numbers

    \[ i = \sqrt{-1} \]

Complex numbers are the sum of a real component and an imaginary component

    - 1+2i, 3.4-132.65i, etc.
    - Imaginary component may have coefficient 0 — so all real numbers also complex
    - Real component may have value 0 — so all imaginary numbers also complex

Java does not have complex numbers built in

    - But we can model them as a class of our own
Designing a class of complex numbers

• What fields do we need in our complex number class?
  – One for the real component, one for the imaginary component
  – Store each as a double

• What methods on complex numbers do we need?
  – Getters for each component
  – Arithmetic: addition, subtraction, multiplication, division
    * There’s also exponentiation, trig functions, etc., although the formulas are rather complex
  – Other operations: magnitude, conjugate
  – A test for zero is a common method for this sort of class

• What sort of constructor do we need?
  – Takes real and imaginary components, stores them in the object fields

The basics

• All classes start by declaring them by name
• Then we declare the fields and their types
  – We’ll use a different object for each complex number, so the fields can be final
• The constructor just sets up the fields

public class Complex {

    private final double real, imaginary;

    Complex(final double real,
            final double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    Accessors
    The two getters are simple methods which just return a field

    public double getReal() {
        return real;
    }

    public double getImaginary() {
        return imaginary;
    }

}
Addition and subtraction

• We add complex numbers componentwise:
  – Add the real part to the real part
  – Add the complex part to the complex part

• In Java, object methods are in the context of one object
  – But addition is a binary operator
  – So the method takes a second argument
  – The result is a separate object

• Subtraction is just like addition

```java
public Complex add(final Complex that) {
    final double realResult = real + that.real;
    final double imaginaryResult = imaginary + that.imaginary;
    return new Complex(realResult, imaginaryResult);
}
```

```java
public Complex subtract(final Complex that) {
    final double realResult = real - that.real;
    final double imaginaryResult = imaginary - that.imaginary;
    return new Complex(realResult, imaginaryResult);
}
```

Multiplication

Multiplying two sums: FOIL

• \((a+bi)(c+di) = ac+adi+bci-bd\)

• Since \(i^2=-1\), instead of \(bdi^2\) we have \(-bd\)

• Simplifying, the product is \((ac-bd)+(ad+bc)i\)

```java
public Complex multiply(final Complex that) {
    final double resultReal =
        real * that.real
        - imaginary * that.imaginary;
    final double resultImaginary =
        real * that.imaginary
        + imaginary * that.real;
    return new Complex(resultReal, resultImaginary);
}
```
Conjugate
Conjugate is a simple idea that will help us when we implement division

• $a + bi = a - bi$

• The conjugate has the nice property that a number multiplied by its conjugate is always a real number

• Conjugate is a unary operation, so it does not take an additional argument

Another useful operation is simpler multiplication by a real coefficient

• $a(b+ci)=ab+aci$

Division
With conjugate and multiplication, division is straightforward

• We need to simplify $\frac{a + bi}{c + di}$ to the form $x + yi$

• First multiply the quotient top-and-bottom by the conjugate of $c + di$:

$$\frac{a + bi}{c + di} = \frac{(a + bi)(c + di)}{(c + di)(c + di)}$$

• Since $(c + di)(c + di)$ is real, we can express the quotient in terms of the operations we have already implemented

public Complex divide(final Complex c) {
    final Complex conj = c.conjugate();
    final double denom = c.multiply(conj).getReal();
    return this.multiply(conj)
      .scale(1/denom);
}
Magnitude
If we interpret complex numbers as points on a plane, magnitude represents the distance from the origin.

```java
public double magnitude() {
    return Math.sqrt(real*real + imaginary*imaginary);
}
```

5.4 Data sets

Representing a data set — first steps

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

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

Better: keep data internal to the class

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

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

- Can be accessed only by methods of the same class
- So the only way to access the object is via its methods

Constructors to set up the object

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

    public Data(double[] sourceData) {
        data = new double[sourceData.length];
    }
}
```
for (int i = 0; i < sourceData.length; i++) {
    data[i] = sourceData[i];
}

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

Adding to the data set

public class Data {
    private double[] data;

    public void addData(double[] additionalData) {
        final int l = data.length;
        double[] newData = new double[l + additionalData.length];
        for (int i = 0; i < l; ++i) {
            newData[i] = data[i];
        }
        for (int i = 0; i < additionalData.length; ++i) {
            newData[l+i] = additionalData[i];
        }
        data = newData;
    }

    // ...

Clearing the data set

public class Data {
    private double[] data;

    public void clearData() {
        data = new double[0];
    }

Then we can have multiple data sets

Data data1 = new Data(array1);
Data data2 = new Data(someOtherArray);
Data data3 = new Data(thirdArray);
Multiple constructors

public class Data {
    private String name;
    private double[] data;

    public Data(double[] sourceData) {
        this("(unnamed)", sourceData);
    } // ...

    public Data(String dataSetName, double[] sourceData) {
        name = dataSetName;
        data = new double[sourceData.length];
        for(int i=0; i<sourceData.length; i++) {
            data[i] = sourceData[i];
        }
    } // ...

    // In this situation, calling this as if it were a function means we want to use a different constructor
    // We can do this only in a constructor
    //  Moreover, we can do this only as the first step in a constructor

Creating and returning a new instance

public class Data {
    private String name;
    private double[] data;

    public Data withAdditionalData(double[] additionalData) {
        final int l = data.length;
        double[] newData = new double[l + additionalData.length];
        for(int i=0; i<l; ++i) {
            newData[i] = data[i];
        }
        for(int i=0; i<additionalData.length; ++i) {
            newData[l+i] = additionalData[i];
        }
        return new Data(name, newData);
    } // ...

   // Very useful if we consider our class data to be immutable
   //  So methods create and return new classes instead of changing existing ones.
   // Many Java library classes use this approach
An object can refer to itself
   Every Java object can refer to itself
   • The local variable this is always defined in object methods
   • It refers to the object itself
   • Its type is the class in which the method lives
   • Can be used
      – For clarity
      – When parameter names, etc. shadow field names
      – To pass the object as an argument to another method
      – We’ll see several examples of using this in the next several classes

Choosing names
   Remember how we defined the constructor

```java
public class Data {
    private String name;
    private double[] data;

    public Data(String dataSetName, double[] sourceData) {
        name = dataSetName;
        data = new double[sourceData.length];
        for(int i=0; i<sourceData.length; i++) {
           data[i] = sourceData[i];
        }
    }

    // ...

```

Consistent use of this
   Many would argue that it is better style to shadow field names with constructor argument names
   • To make the purpose of the parameters clear
   • To avoid needing odd or inconsistent alternate names just to avoid shadowing

```java
public class Data {
    private String name;
    private double[] data;

    public Data(String name, double[] data) {
        this.name = name;
        this.data = new double[data.length];
        for(int i=0; i<data.length; i++) {
           this.data[i] = data[i];
        }
    }

```
This and that
Sometimes you’ll see that as a parameter name

• It’s not special like this — it’s just a local name like x or i or nameString

• Handy especially for small methods where the purpose is otherwise very clear

```java
class Data {
    private String name;
    private double data[];

    public boolean hasLargerDataSet(Data that) {
        return this.data.length > that.data.length;
    }
}
```

Caching values when we compute them

```java
private double theSavedMean;
private boolean haveMeanSaved = false;
public double getMean() {
    if (!haveMeanSaved) {
        double total=0.0;
        for(int i=0; i<data.length; ++i) {
            total += data[i];
        }
        theSavedMean = total / data.length;
        haveMeanSaved = true;
    }
    return theSavedMean;
}
```

But when we change the data, we must clear the cache

```java
public void addData(double additionalData[]) {
    final int l = data.length;
    double newData[] = new double[l + additionalData.length];
    for(int i=0; i<l; ++i) {
        newData[i] = data[i];
    }
    for(int i=0; i<additionalData.length; ++i) {
        newData[l+i] = additionalData[i];
    }
}
```
data = newData;
haveMeanSaved = false;
}

**What if we cache other things too?**

- Standard deviation, higher moments, median, etc.
- We still could compute each value separately
- Or cache all of them at the same time

```java
private boolean cacheUpdated = false;
private void ensureCached() {
    if (!cacheUpdated) {
        final int count = data.length;
        double total=0.0, sqTotal=0.0;
        for(int i=0; i<count; ++i) {
            double point = data[i];
            total += point;
            sqTotal += point*point;
        }
        theSavedMean = total/count;
        theSavedStdDev =
            Math.sqrt(count*sqTotal - total*total)/count;
        cacheUpdated = true;
    }
}
```

**Consolidate clearing**

- If we compute each value separately, then we have several different is-cached flags
- It is prudent to consolidate clearing them under one method

```java
private void clearCached() {
    haveMeanSaved = false;
    haveStdDevSaved = false;
    // ...
}
```

- Every time we mutate the values, all the flags must be reset

### 5.5 The `ArrayList` class

**Java provides built-in representations of many data structures**

- Today we’ll look at class `ArrayList`
• Based on arrays
  – So accessing an element by its index is fast
  – But inserting an element into the middle of a list can be slower

• The class keeps track of the size of the array and the size of the list
  – So the number of the elements in the list may differ from the size of the underlying array
  – When the array needs to grow, the ArrayList allocates new space without the programmer needing to know

• No direct access to the underlying array
  – Instead, ArrayList defines an application programmer interface (API) of different operations

• To use

As with an array, name the type of list elements

• When we use ArrayList as a type, we mention the type of the elements of the list
  • ArrayList<Integer>, ArrayList<Character>, ArrayList<BigInteger>, etc.

• The type of elements we try to add to a list must match the declared type
  • Elements we extract from the list will match the given type

Some ArrayList<E> constructors and methods

• ArrayList() — Constructor for a new empty list

• ArrayList(int init) — Constructor for a new empty list, where we tell Java that it should have capacity for at least init entries

• size() — Returns the number of elements now in the list

• isEmpty() — Returns true is the list contains no elements

• contains(Object o) — Returns true if o is now in the list

• clear() — Empties the list

• add(E element) — Append the element to the end of the list

• get(int index) — Returns the given element of the list

• remove(int index) — Removes the given element from the list

• set(int index, E element) — Sets the element at the given position of the list
  – It’s an error if there’s no such element of the list
A small example

```java
import java.util.ArrayList;

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

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

Changing the `ArrayList`

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

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

Iterating through the `ArrayList`

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

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

5.6 TODO Replace this with simpler examples going forward

5.7 Permuting the letters of a word

Permuting the letters of a word
For a string `tab`, the shufflings of its letters are

- `tab`
- `tba`
- `atb`
- `abt`
- `bta`
- `bat`

It’s a recursive algorithm

- We figure out a way to describe a partly-completed shuffling
- The recursive algorithm figures out the possible next steps for the partly-completed shuffling, and tries them one at a time
- The recursion ends when the shuffling is completed

How to shuffle

- A partial shuffle has
  - The prefix of the shuffle we’re working on
  - The characters which remain to be shuffled
- Given a partial shuffle, what are the next steps?
  - Each characters remaining to be shuffled could be appended to the prefix
- When there are no more characters to be shuffled, the prefix will in fact be a complete shuffle
The state of a shuffler
Each shuffler has the prefix and the characters still to be shuffled

```java
public class WordShuffler {
    private final String prefix;
    private final ArrayList<Character> unshuffled;

    public WordShuffler(String prefix,
                         ArrayList<Character> unshuffled) {
        this.prefix = prefix;
        this.unshuffled = unshuffled;
    }
```

Setting up a shuffler
When we start shuffling a string, we must set up what the constructor needs

```java
public static WordShuffler getShuffler(String string) {
    final ArrayList<Character> unshuffled = new ArrayList<Character>(string.length());
    for(final char c : string.toCharArray()) {
        unshuffled.add(c);
    }
    return new WordShuffler("", unshuffled);
}
```

Shuffling action

```java
public void shuffle() {
    if (unshuffled.isEmpty()) {
        System.out.println(prefix);
    } else {
        for(final Character c : unshuffled) {
            final String nextPrefix = prefix + String.valueOf(c);
            final ArrayList<Character> nextUnshuffled = new ArrayList<Character>(unshuffled);
            nextUnshuffled.remove(c);
            final WordShuffler nextShuffler = new WordShuffler(nextPrefix, nextUnshuffled);
            nextShuffler.shuffle();
        }
    }
}
```

A main routine

```java
public static void main(String[] args) {
    final Scanner scanner = new Scanner(System.in);
```
System.out.print("Word to scramble: ");
final String word = scanner.next();
final WordShuffler shuffler = getShuffler(word);
shuffler.shuffle();
}

**Shuffling "tab"**

private final String prefix;
private final ArrayList<Character> unshuffled;

public static WordShuffler getShuffler(String string) {
    final ArrayList<Character> unshuffled = new ArrayList<Character>(string.length());
    for(final char c : string.toCharArray()) {
        unshuffled.add(c);
    }
    return new WordShuffler("", unshuffled);
}

public void shuffle() {
    if (unshuffled.isEmpty()) {
        System.out.println(prefix);
    } else {
        for(final Character c : unshuffled) {
            final String nextPrefix = prefix + String.valueOf(c);
            final ArrayList<Character> nextUnshuffled = new ArrayList<Character>(unshuffled);
            nextUnshuffled.remove(c);
            final WordShuffler nextShuffler = new WordShuffler(nextPrefix, nextUnshuffled);
            nextShuffler.shuffle();
        }
    }
}

public static void main(String[] args) {
    final Scanner scanner = new Scanner(System.in);
    System.out.print("Word to scramble: ");
    final String word = scanner.next();
    final WordShuffler shuffler = getShuffler(word);
    shuffler.shuffle();
}

- What happens when we shuffle tab?

**Looking back on the shuffler**
public void shuffle() {
    if (unshuffled.isEmpty()) {
        System.out.println(prefix);
    } else {
        for(final Character c : unshuffled) {
            final String
                nextPrefix = prefix + String.valueOf(c);
            final ArrayList<Character> nextUnshuffled =
                new ArrayList<Character>(unshuffled);
            nextUnshuffled.remove(c);
            final WordShuffler nextShuffler =
                new WordShuffler(nextPrefix, nextUnshuffled);
            nextShuffler.shuffle();
        }
    }
}

Two things happening at the same time in the shuffler

1. Build a tree of WordShuffler nodes
2. Traversing the tree to find strings at the nodes

It makes sense to do these at the same time when all we’re doing is searching for permutations

• But the two steps are interesting separately

• For many problems we’d need to do just one or the other

• So let’s look at doing each step separately

Node state and creation

public class WordShufflerNode {
    private final String prefix;
    private final ArrayList<WordShufflerNode> children = new ArrayList<WordShufflerNode>();

    public WordShufflerNode(String prefix,
                              ArrayList<Character> unshuffled) {
        this.prefix = prefix;
        for(final Character c : unshuffled) {
            final String nextPrefix = prefix + String.valueOf(c);
            final ArrayList<Character> nextUnshuffled =
                new ArrayList<Character>(unshuffled);
            nextUnshuffled.remove(c);
            final WordShuffler nextShuffler =
                new WordShuffler(nextPrefix, nextUnshuffled);
            children.add(nextShuffler);
        }
    }
}
Traversing the tree of nodes

```java
public void traverse() {
    if (children.isEmpty()) {
        System.out.println(prefix);
    } else {
        for(final WordShufflerNode node : children) {
            node.traverse();
        }
    }
}
```

Static method and main routine mostly unchanged

```java
public static WordShufflerNode getShuffler(String string) {
    final ArrayList<Character> unshuffled = new ArrayList<Character>(string.length());
    for(final char c : string.toCharArray()) {
        unshuffled.add(c);
    }
    return new WordShufflerNode("", unshuffled);
}
```

```java
public static void main(String[] args) {
    final Scanner scanner = new Scanner(System.in);
    System.out.print("Word to scramble: ");
    final String word = scanner.next();
    final WordShufflerNode shuffler = getShuffler(word);
    shuffler.traverse();
}
```

Building the tree
Search method for WordShufflerNode

- Start with `traverse`

```java
public void traverse() {
    if (children.isEmpty()) {
        System.out.println(prefix);
    } else {
        for (final WordShufflerNode node : children) {
            node.traverse();
        }
    }
}
```

- Add parameter, and stop when we find that parameter in a leaf

```java
public boolean search(String s) {
    if (children.isEmpty()) {
        return prefix.equals(s);
    } else {
        for (final WordShufflerNode node : children) {
            if (node.search(s)) {
                return true;
            }
        }
    }
    return false;
}
```
5.8 TODO Replace this with simpler examples going forward

5.9 The TreeSet class

Trees are another sort of structure that we can use to store information

- Good for maintaining an order
- Insertion in order is easy
- Search is reasonably quick
- Java TreeSet\langle E\rangle
  - add(E e), remove(Object o)
  - size(), clear()
  - contains(Object o)
  - first(), last()
  - floor(E e), ceiling(E e)

A TreeSet example

```java
public static void main(String[] args) {
    final TreeSet<Integer> set = new TreeSet<Integer>();
    set.add(58);
    set.add(35);
    set.add(75);
    set.add(14);
    set.add(83);
    set.add(63);
    set.add(46);
    set.add(97);
    set.add(36);
    set.remove(83);
    System.out.println("size() " + set.size());
    System.out.println("first() " + set.first());
    System.out.println("last() " + set.last());
    System.out.println("floor(40) " + set.floor(40));
    System.out.println("ceiling(50) " + set.ceiling(50));
    System.out.println("contains(14) " + set.contains(14));
    System.out.println("contains(20) " + set.contains(20));
}
```

6 Subclasses and abstract classes

6.1 A class for representing people

A class for representing people
public class Person {
    private final String firstNames;
    private final String lastName;
    public Person(final String firstNames,
                  final String lastName) {
        this.firstNames = firstNames;
        this.lastName = lastName;
    }
    public String getFirstNames() { return firstNames; }
    public String getLastName() { return lastName; }
    public String fullName() {
        return firstNames + " " + lastName;
    }
}

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

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

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

Instantiating classes and subclasses

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

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

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

then we can pass person to it
final Student student = new Student(
    "Jean-Luc", "Godard",
    new int[] {
        1250678, 1250678, 1250678
    });

- Has methods `getFirstNames`, `getLastName` and `getCourseIds`
- We can also pass `student` to `getInLine`
  - Every `Student` is also a `Person`
  - But not every `Person` is a `Student`

What about the fields

```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

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

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

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

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

Using these methods
Assume we have a method in a third class
public class Driver {
    // ...
    public void printFancyEmail(Person p) {
        System.out.print(p.getFirstNames());
        System.out.print(" ");
        System.out.print(p.getLastName());
        System.out.print(" <");
        System.out.print(p.getEmailAddr());
        System.out.println(" >");
    }
}

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

Subclasses can also change parent methods

• The method

    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

```java
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"
  – In fact two things in Java are called polymorphic; this one is *method polymorphism*

### 6.2 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 over-ridden

 * In String, for example, equals is overridden to check the length and then the corresponding characters

 Are two Card instances the same?

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

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

 • But there’s one problem

   – isEqualTo takes a Card as its argument

   – But 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:

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

 From Object to Card

 To make equals work, we must explicitly check the type, and make a downcast

 @Override public boolean equals(final Object o) {
     if (o instanceof Card) {
         final Card that = (Card)o;
         return this.isEqualTo(that);
     } else {
         return false; // Not equal if not the same type
     }
 }
6.3 Abstract classes

Classes extending classes

- So far we’ve seen classes extending classes
  - We can create a `Student` object, or a `Parent` object, or even an `Object` object
  - Each superclass has been a fully realized class in its own right

- *But it doesn’t have to be that way!*
  - We can define a sort of class in which some method are promised but not actually defined
  - We can’t actually create objects of such "incomplete" classes, but:
    * We can use them for the types of method parameters
    * Other classes can extend them

- It can be useful as we structure our program to stage partially-completed descriptions of classes
  - Bundle common implementation aspects together
  - Then further subclasses can differ

**Remember: our class of data sets**

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

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

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

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

**Separate the operations on the data store from the actual representation**

```java
public abstract class AbstractDataSet {
    public abstract int getDataSetSize();
    public abstract double getDataSetItem(int index);
```
public abstract void addDataSetItem(int index, double d);

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

We cannot create instances of AbstractDataSet

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

• When we compile:

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

One implementation might use an ArrayList

public class ArrayListDataSet extends AbstractDataSet {
    private final ArrayList<Double> theList = new ArrayList<Double>();

    public int getDataSetSize() {
        return theList.size();
    }
    public double getDataSetItem(int index) {
        return theList.get(index);
    }
    public void addDataSetItem(int index, double d) {
        theList.add(index, d);
    }
}

• We do not need to define mean — we inherit from AbstractDataSet
An implementation that cares about order might use a **TreeSet**

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

    public int getDataSetSize() {
        return theSet.size();
    }

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

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

7 Interfaces

**Interfaces**

- Like an abstract class, but limited in its contents
- In many Java systems:
  - Only declares methods, but gives no method bodies
  - No fields
  - No constructors
- More recent Java systems:
  - Methods can have default implementations
  - Fields must be **public, static and final**
  - Still no constructors
- But classes can implement more than one interface even though they can extend only a single class
7.1 Complex numbers revisited

Complex numbers revisited
Earlier we used complex numbers as an example of a class.

But there are two common representations of complex numbers

- \( z = x + iy \), with a real and an imaginary component
  - What we used in class Complex
- But we can also use polar coordinates
  - Imagine the complex numbers on a plane, real numbers along the x-axis, imaginary numbers along the y-axis
  - Instead of storing an x- and y-coordinate, store the distance from 0, and the angle from the x-axis
  - \( z = re^{i\theta} \)

- How can we have both representations in Java at the same time?

Use an interface to describe supported operations
Complex will be an interface instead of a class

```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);  
}

Our old class is one implementation of the interface

Different class name

• And implementing the Complex interface

public class ComplexComponents
    implements Complex {
    private final double real, imaginary;
    public ComplexComponents
        (final double real,  
         final double imaginary) {
        this.real = real;  
        this.imaginary = imaginary;
    }
    public double getReal() { return real; }  
    public double getImaginary() {  
        return imaginary;
    }
    public boolean isZero() {  
        return real == 0.0 && imaginary == 0.0;
    }
    public double getMagnitude() {  
        return Math.sqrt(  
            real*real + imaginary*imaginary);
    }
    public double getAngle() {  
        return Math.atan2(real, imaginary);
    }
}

Our old class is one implementation of the interface

We can no longer assume that every Complex has the same implementation ComplexComponents

• We can use this.real and this.imaginary

• But when we have a Complex argument, we must use methods since we cannot make assumptions about the fields

    public Complex add(Complex that) {  
        return new ComplexComponents(  
            real+that.getReal(),  
            imaginary+that.getImaginary());
    }
Another implementation

We can now define another class to implement the same interface, but with different internals

```java
public class ComplexPolar implements Complex {
    private final double magnitude, angle;
    public ComplexPolar(final double magnitude,
                         final double angle) {
        this.magnitude = magnitude;
        this.angle = angle;
    }
    public double getReal() { return magnitude * Math.cos(angle); }
    public double getImaginary() { return magnitude * Math.sin(angle); }
    public double getMagnitude() { return magnitude; }
    public double getAngle() { return angle; }
    public boolean isZero() { return magnitude == 0.0; }
}

Some things are easier this way

```java
public Complex multiply(Complex that) {
    return new ComplexPolar(this.magnitude*that.getMagnitude(),
                              this.angle+that.getAngle());
}
public Complex dividedBy(Complex that) {
    return new ComplexPolar(this.magnitude/that.getMagnitude(),
                              this.angle-that.getAngle());
}
```

• Addition, on the other hand, is very complicated

What about equality?

We are OK with different representations being judged equal if they represent the same number

```java
public class ComplexComponents
    implements Complex {
    // ...

    @Override public boolean equals(Object o) {
        if (o instanceof Complex) {
            final Complex that = (Complex)o;
            final double thatRe = that.getReal(),
            thatIm = that.getImaginary();
            return real == thatRe
```
public class ComplexPolar
    implements Complex {
    // ...

    @Override public boolean equals(Object o) {
        if (o instanceof Complex) {
            final Complex that = (Complex)o;
            final double thatMa = that.getMagnitude(),
                thatAn = that.getAngle();
            return magnitude == thatMa
                && (magnitude == 0.0
                    || angle == thatAn);
        } else {
            return false;
        }
        return false;
    }

    7.2 Iterators

    Iterators
    We've seen the ArrayList class from Java's collections library

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

    • An iterator for elements of type E has two essential methods
      
        – boolean hasNext()
        – E next()

    Iterators example

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

• Prints

Moe
Larry
Curly

**Iterators example — the HashSet**

HashSet is another standard Java library class

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

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

• HashSet is a set, so no duplicates

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

**Iterators example — the TreeSet**

 TreeSet is a third type of collection provided in the standard libraries

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

    final Iterator<String> namesIterator = names.iterator();
    while (namesIterator.hasNext()) {
        System.out.println(namesIterator.next());
    }
}
```
The iterator method is a member of an interface
All of these classes implement the class Iterable

- Iterable = things which can be iterated
- Iterator = the thing which does the iteration on behalf of something else
- Iterable drives the Java for loop

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

Requires that names implement Iterable<String>

Interfaces in the collections hierarchy
If we look at the ArrayList<E> documentation, we see that it implements several interfaces:

- Collection<E>
  - Specified methods like clear(), add(E item) and contains(Object o) that are about one thing containing another
- Iterable<E>
  - Specifies the iterator() method
- List<E>
  - Specifies methods like add(int index, E element) that rely on an order among elements
- The various abilities are specified in several different places

Recall again our Data class

```java
public class Data {
    private String name;
    private double[] data;

    public Data(String name, double[] data) {
        this.name = name;
        this.data = new double[data.length];
        for(int i=0; i<data.length; ++i) {
            this.data[i] = data[i];
        }
    }
}
```
• What if we wanted to be able to iterate through our data?

Steps

• Create a class of iterators over our array
• Make sure it implements Iterator
• Add a method iterator() to Data, returning an object of the new class
• Declare Data to implement Iterable

Iterators for the array

class DataIterator implements Iterator<Double> {
    private double[] data;
    private int nextToSend=0;

    public DataIterator(double[] data) {
        this.data = new double[data.length];
        for(int i=0; i<data.length; ++i) {
            this.data[i] = data[i];
        }
    }

    • The constructor makes a copy of the array, to make sure the underlying data doesn’t change

Iterators for the array

class DataIterator implements Iterator<Double> {
    private double[] data;
    private int nextToSend=0;
    public DataIterator(double[] data) {
        this.data = new double[data.length];
        for(int i=0; i<data.length; ++i) {
            this.data[i] = data[i];
        }
    }
}
```java
this.data[i] = data[i];
}
}

public boolean hasNext() {
    return nextToSend<data.length;
}

public Double next() {
    return data[nextToSend++];
}
}

• We use the index and the length of the array to judge if there is more to iterate
• Note the use of ++ — we place it after the variable
  – So we fetch the value of nextToSend, and then increment it
  – We get the value before the increment, not after
  – If the ++ came first, then we would get the incremented value

Generating the iterator

public class Data implements Iterable<Double> {
    private final double[] data;

    public Iterator<Double> iterator() {
        return new DataIterator(data);
    }

    // ...

7.3 More interface concepts

An interface for doing something
    A small interface example:

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

And a class which implements that interface:

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();
}

Extending the interface

Just as one class can extend another class, one interface can extend another interface

public interface DoItPlus extends DoIt {
    public boolean didItWork(int i, double x, String s);
}

A class which implements DoItPlus must define not only didItWork but also all of DoIt’s methods

public class PrintAndMeasureBetter implements DoItPlus {
    public void doSomething(int i, double x) {
        System.out.println(i*i+x);
    }
    public int doSomethingElse(String s) {
        return s.length();
    }
    public boolean didItWork(int i, double x, String s) {
        return i<x && s.length()>2;
    }
}

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);
    }
}
8 GUIs

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 in 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
– An ActionListener, which receives ActionEvents

• Use fields in the window’s class to hold objects particular to the domain
  – So the Eclipse window would know about 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

A first GUI example
File SalaryCalcButtonFrame.java

• What is the superclass? What interfaces does our class implement?

• Where do we set up the window?

• Why are some components declared as fields, but other components declared as local variables of the
  constructor?

• Why do we need to both
  – Include the actionPerformed method, and
  – Declare the class to implement ActionListener?

• What is the purpose of the addActionListener call?

Tweaking the GridBagLayout

• What is the anchor field of the GridBagConstraints used for?

• How do we indicate the different directions to which a component can be anchored?

• Why do we use pack() again after changing a label?

Multiple buttons

• How can one listener tell the difference between two different buttons?

• How and why do we separate the main window from the components’ listener?

• How can different buttons have different listeners?