1 Introduction

2 Simple imperative programming
  2.1 Names and assignments .................................................. 6
  2.2 Leap years ..................................................................... 9
  2.3 The for-loop ................................................................... 12
  2.4 Factorials ....................................................................... 13
  2.5 How long is that number? .................................................. 18
  2.6 Fibonacci numbers ............................................................ 19
  2.7 Fix that sentence .............................................................. 20
  2.8 Another kind of selection .................................................... 21
  2.9 Another kind of loop .......................................................... 23
  2.10 Multiple loops ................................................................. 24

3 Methods
  3.1 Method basics ................................................................. 28
  3.2 A method for leap years ...................................................... 30
  3.3 A choose method ............................................................... 31
  3.4 Factorial and Fibonacci methods ........................................ 33
  3.5 Factorial and Fibonacci methods ........................................ 36
  3.6 Methods and your work ..................................................... 36

4 Arrays
  4.1 Array basics .................................................................... 38
  4.2 Combining two arrays: the dot product ................................ 41
  4.3 Sorting an array ............................................................... 42
  4.4 The Sieve of Eratosthenes ................................................. 45
  4.5 Two-dimensional arrays .................................................... 50

1 Introduction
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;
    }
}

Parts of the 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;
    }
}
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."); // End of this line
        return;
    }
}

Input as well as output

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

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

Compiler says we have an error

Nothing!

The other things that happen with a program

```java
public class Kilograms {
    public static void main (String [] args) {
        double pounds = 20;
        System.out.print(pounds);
        System.out.print(" pounds is ");
        System.out.print(pounds / 0.0);
        System.out.println(" kilograms.");
        return;
    }
}
```

Runtime system says we have an error

A little bit of I/O and then nothing!

Errors are frustrating
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:
    ```java
    System.out.print("Hello...");
    System.out.println("again");
    ```
  – For some operations we use a helper:
    ```java
    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:

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

• In today’s reading:

  ```java
  int litterSize  = 3;
  int yearlyLitters = 5;
  int annualMice   = 0;
  ```
Pick good names

- Use lower camel-case for variable names.
  - Named after the "humps" of upper-case letters in the middle of the name
- Descriptive names, but be reasonable with length
- Use letters, and maybe numbers at the very end
- *Mnemonic* - assisting the memory
  *Consistent* - ease understanding

It's an update, not an equation

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

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

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

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

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

- 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

```java
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
    given = given % (60 * 60); // 5
    final int minutes = given / 60; // 6
    final int seconds = given % 60; // 7

    System.out.printf("%d:%02d:%02d\n", // 8
                     hours, minutes, seconds);
    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?

```java
    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?

```java
    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:

    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 {
                System.out.println(year + " was not a leap year."); // 6
            }
            return; // 7
        }
    }

- Try 1986 and 1900
Really calculating a leap year

- There are exceptions to the divide-by-four rule
  - Years divisible by 100 are \textit{not} leap years
  - \textit{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 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

```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

        final boolean isLeapYear; // 4
        if (year % 400 == 0) { // 5
            isLeapYear = true; // 6
        } else if (year % 100 == 0) { // 7
            isLeapYear = false;
        }
```

10
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
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:

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

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

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) {
        final Scanner scnr = new Scanner(System.in);
        System.out.print("Factorial of: ");
        final int num = scnr.nextInt();
        int product = 1;
        for (int i = 1; i <= num; i++) {
            product = product * i;
        }
        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);
        System.out.print("Factorial of: ");
        final int num = scnr.nextInt();
        int product = 1;
        for (int i = 1; i <= num; i++) {
            product = product * i;
        }
        System.out.println("It's: "+ product);
        return;
    }
}
```
```java
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
}

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

A different factorial calculator

• 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
- B D E
- C D 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 \( \binom{n}{m} = \frac{n(n-1)\cdots(n-m+1)}{m!} \)
  - 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
}

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

Do less work!

- Try it with 4, 2

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

- 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

```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 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 floor for rounding down, and log10
  • Its full name is java.lang.Math
    – Classes in java.lang (unlike 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, · · ·

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

    final String greeting = "hello";
    final String name = "Jim";

• The built-in operator on strings is concatenation, written with the + sign

    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

    final String greetingAndSpaceAndName = greeting + " " + name;

Characters

• Strings are made of characters

• But characters and strings have completely different types!

    final char capitalA = ’A’;

• Strings contains zero or more characters, but a character itself is exactly one character
• **Strings to characters:** use the `charAt` method

```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!
```

• **Characters to strings:** use the `String.valueOf` method

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

**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();
```
```java
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.");

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;
    }
}
```

22
The default

System.out.println("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)); // 9
    }
    i++; // 10
}

System.out.println(); // 11
return; // 12

2.10 Multiple loops

What does this program do?

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

  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

  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
```
• How can we generate this triangle?

Another triangle

For example, for 5:

```java
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_env: frame

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

• Function graph

```
  |      *
  |     / \   
  |  5    *    *
  | /      / \  
  | /      /   
  | 4      *    *
  | /      /    
  | /      /     
  | 3      *     *
  | /      /      
  | /      /       
  | 2      *       *
  | /      /        
  | /      /         
```
• Absolute value!
  – 5-|5-i|
• In Java, it’s Math.abs

One loop

```java
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

```java
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("*");
        } else {
            System.out.print(".");
        }
    }
    System.out.println();
}

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

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
        }
    }
}
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

    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

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

return;
}

• Note that we need the else statements to keep from printing multiple messages

A leap year method

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;
    }
}

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.");
    }

    return;
}

• 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?

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(chooseC(total, choose));
    }
}
```java
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);
    }
}

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

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

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

```java
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

```java
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) {

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

37
Javadoc comments for the method itself

/**
 * Sorts an array of integers.
 * @param array The array to be sorted.
 * @return void
 * 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 @return

• 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
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
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
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);
}

technical
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

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] · [10,1000,100000]

How can we implement the dot product?

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

        int dotProduct = 0;
        for(int i=0; i<array1.length; i++) {
            dotProduct += array1[i] * array2[i];
        }
        System.out.println(dotProduct);
    }
}
Dot product

public 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?

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 " + i+1 + ": "); // 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

```
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;
```

• 4 numbers: 5, 20, 13, 2

What does bubble sort cost?

• How many times will we execute the comparison between elements (and possibly swap them) for an array of length \( n \)?

• The first time through the inner loop, it’s \( n - 1 \) times; then \( n - 2 \), and so on down to 1.

• So in total, it’s \( \sum_{i=1}^{n-1} i = \frac{n^2-n}{2} \).
  
  – The constant factor \( \frac{1}{2} \) isn’t an interesting detail — for input of size \( n \), the number of steps is on the order of \( n^2 \).
  
  – Even subtracting \( n \) does not have a big impact, once \( n \) starts to get big.
  
  – The \( n^2 \) growth is what’s interesting to us.
  
  – We write this as \( O(n^2) \) — on the order of \( n^2 \).

• Bubble sort is fine for smaller arrays, but for larger arrays gets too slow.

  – The best sorting algorithms run in \( O(n \log n) \) time — we’ll look at one of these later this semester.
### 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

<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

<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

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

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

<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

• And so on with the new lowest unmarked number, and so on, and so on

<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

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

- So let's code that up

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();

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];

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;
}

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.");
    }
}

The Sieve
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