Computer Science S-111
Intensive Introduction to Computer Science

Harvard Summer School 2017
David G. Sullivan, Ph.D.

Unit 1: Getting Started

Unit 2: Imperative Programming I

Unit 3: Imperative Programming II

Unit 4: Processing Collections of Data

Unit 5: Object-Oriented Programming

Unit 6: Foundations of Data Structures

Unit 8: Sequences

Unit 9: Trees and Hash Tables

Unit 1, Part 1

Intensive Introduction to Computer Science

Course Overview
Programming in Scratch

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Welcome to CS S-111!

Computer science is not so much the science of computers as it is the science of solving problems using computers.

Eric Roberts

- This course covers:
  - the process of developing *algorithms* to solve problems
  - the process of developing computer programs to express those algorithms
  - fundamental *data structures* for imposing order on a collection of information
  - the process of *comparing* data structures & algorithms for a given problem

Computer Science and Programming

- There are many different fields within CS, including:
    - software systems
    - computer architecture
    - networking
    - programming languages, compilers, etc.
    - theory
    - AI
- Experts in many of these fields don’t do much programming!
- However, learning to program will help you to develop ways of thinking and solving problems used in all fields of CS.

A *Rigorous* Introduction

- Intended for:
    - those who plan to work extensively with computers
    - future concentrators who plan to take more advanced courses
    - others who want a rigorous introduction
    - no programming background required, but can also benefit people with prior background
- Allow for **20-30 hours** of work per week
    - start work early!
    - come for help!
    - don't fall behind!
- Computer Science S-1 is a less rigorous alternative

CS 111 Requirements

- Lectures and sections
- Ten problem sets (50%)
  - part I = "written" problems
  - part II = "programming" problems
  - grad-credit students will have extra work on most assts.
- Nine unit tests (25%)
  - given at the end of lecture (see the schedule)
  - 25 possible pts. for each
  - if score lower than 18, can take a retest for a max. of 18
- Final exam (25%): Friday, August 4, 8:30-11:30 a.m.
  - comprehensive exam

Textbooks

- **Required**: *The CSCI S-111 Coursepack*
  - contains all of the lecture notes
  - will be available at Gnomon Copy on Mass Ave.
- **Optional** resource for the first half:
  Building Java Programs by Stuart Reges and Marty Stepp (Addison Wesley, 2016).
- **Optional** resource for the second half:
  Data Structures & Algorithms in Java, 2nd edition by Robert Lafore (SAMS Publishing, 2003).

Other Course Staff

- Teaching Assistants (TAs):
  Cody Doucette (head TA)
  Caitlin Fournier
  Justin Ingwersen
  Eli Saracino
- See the course website for contact info.
- **Piazza is your best bet for questions.**
- For purely administrative questions: libs111@fas.harvard.edu
  - will forward your email to the full course staff

Other Details of the Syllabus

- Schedule:
  - note the due dates and test dates
  - no lectures or sections on most Wednesdays
    - **exceptions:** July 5 (July 4 is off), July 12 (July 14 is off), August 2 (August 3 is off)
- Policies:
  - 10% penalty for submissions that are one day late
  - please don't request an extension unless it's an emergency!
  - grading
- Please read the syllabus carefully and make sure that you understand the policies and follow them carefully.
- Let us know if you have any questions.

Algorithms

- In order to solve a problem using a computer, you need to come up with one or more *algorithms*.
- An algorithm is a step-by-step description of how to accomplish a task.
- An algorithm must be:
  - *precise:* specified in a clear and unambiguous way
  - *effective:* capable of being carried out

Example of Defining an Algorithm

Programming

- Programming involves expressing an algorithm in a form that a computer can interpret.
- We will primarily be using the Java programming language.
    - one of many possible languages
- The key concepts of the course transcend this language.

What Does a Program Look Like?

- Here's a Java program that displays a simple message:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello, world");
    }
}
```

- Like all programming languages, Java has a precise set of rules that you must follow.
    - the *syntax* of the language
- To quickly introduce you to a number of key concepts, we will begin with a simpler language.

Scratch

- A simple graphical programming language
  - developed at the MIT Media Lab
  - easy enough for middle school kids
  - makes it easy to create animations, games, etc.

- Download **version 1.4** for free here:

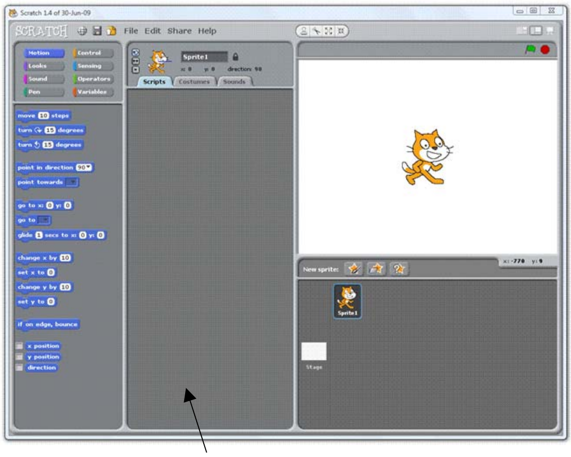
  http://scratch.mit.edu/scratch_1.4/

  - this is **not** the latest version, but it's the one we will use

Scratch Basics

- Scratch programs (*scripts*) control characters called *sprites.*

- Sprites perform actions and interact with each other on the *stage.*



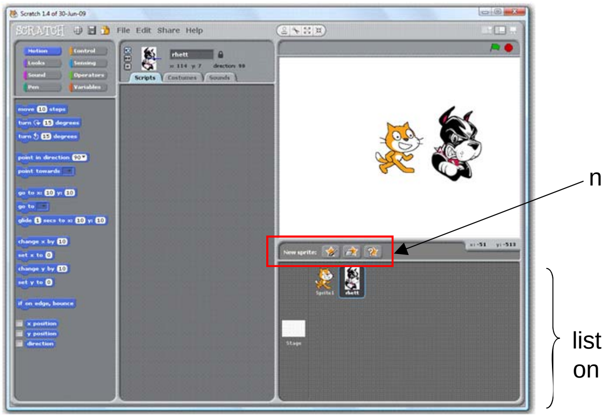
building blocks for programs/ scripts

the stage

drag building blocks here to create scripts

Sprites

- At the start, there is a single cat sprite on the stage.
- You can add and remove sprites, and change their costumes.



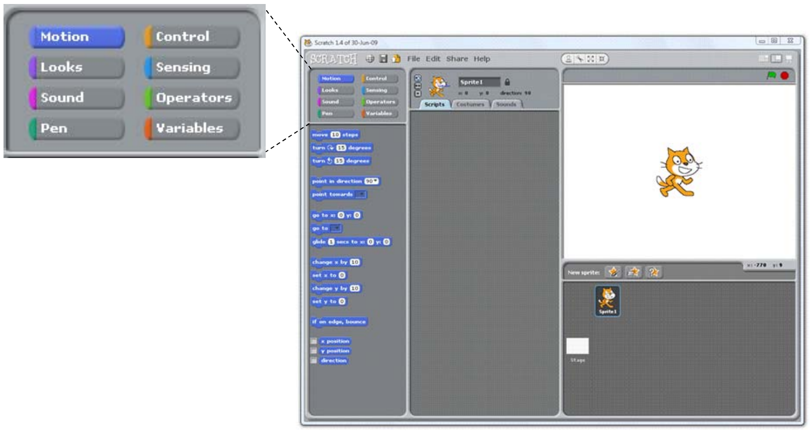
new sprite buttons

list of sprites on the stage

- Clicking on a sprite from the list below the stage shows you the scripts for that sprite (if any).

Program Building Blocks

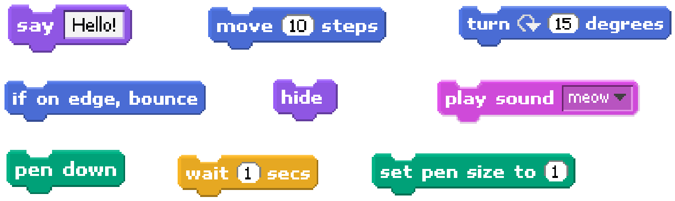
- Grouped into eight color-coded categories



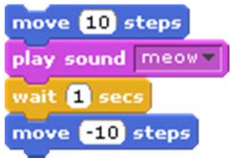
- The shape of a building block indicates where it can go.
- Right-click a building block to get help on that type of block.

Program Building Blocks: Statements

- Statement = a command or action



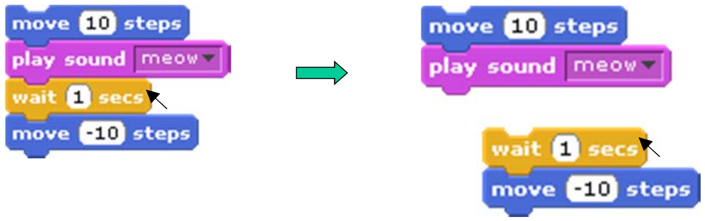
- Statements have bumps and/or notches that allow you to stack them.
  - each stack is a single script



- A statement may have:
  - an *input area* that takes a value (Hello!, 10, 15, etc.)
  - a pull-down menu with choices (meow)

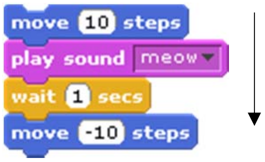
Program Building Blocks: Statements (cont.)

- Clicking on any statement in a script executes the script.
- When rearranging blocks, dragging a statement drags it *and* any other statements below it in the stack.
  - example: dragging the *wait* command below



Flow of Control

- Flow of control = the order in which statements are executed
- By default, statements in a script are executed sequentially from top to bottom when the script is clicked.

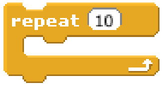


```
move 10 steps
play sound meow
wait 1 secs
move -10 steps
```

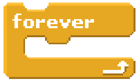
- *Control blocks* (gold in color) allow you to affect the flow of control.
  - simple example: the *wait* statement above pauses the flow of control

Flow of Control: Repetition

- Many control statements are C-shaped, which allows them to control other statements.
- Example: statements that repeat other statements.

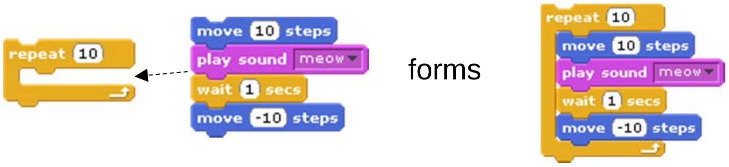


```
repeat 10
```



```
forever
```

- Drag statements inside the opening to create a repeating stack.



```
repeat 10
```

```
move 10 steps
play sound meow
wait 1 secs
move -10 steps
```

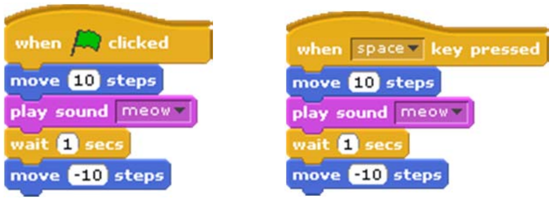
forms

```
repeat 10
    move 10 steps
    play sound meow
    wait 1 secs
    move -10 steps
```

- In programming, a group of statements that repeats is known as a *loop*.

Flow of Control: Responding to an Event

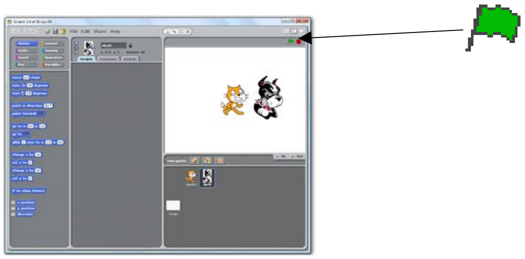
- *Hat blocks* (ones with rounded tops) can be put on top of a script.



```
when ⚑ clicked
move 10 steps
play sound meow
wait 1 secs
move -10 steps
```

```
when space key pressed
move 10 steps
play sound meow
wait 1 secs
move -10 steps
```

- They wait for an event to happen.
    - when it does, the script is executed

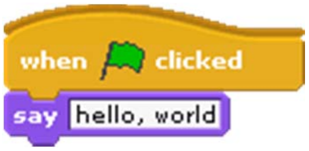


What Does a Program Look Like?

- Recall our earlier Java program:

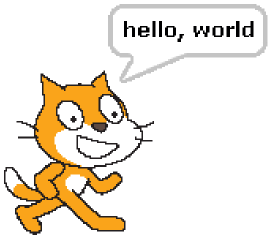
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello, world");
    }
}
```

- Here's the Scratch version



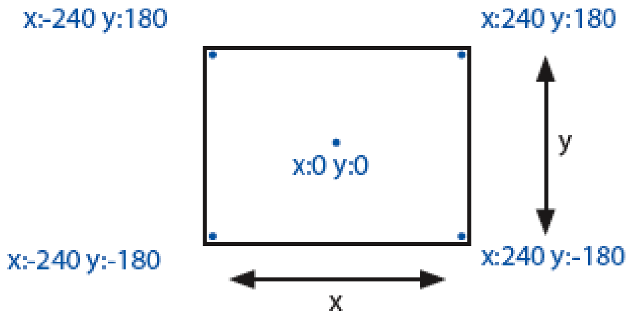
```
when ⚑ clicked
say hello, world
```

… and here's the result:

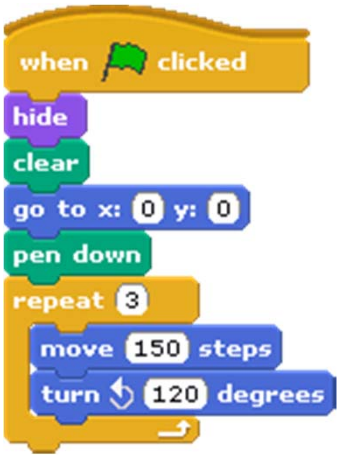


Stage Coordinates

- Dimensions: 480 units wide by 360 units tall
- Center has coordinates of 0, 0

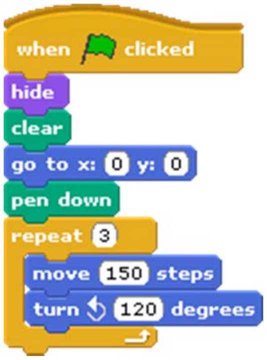


What Does This Program Draw?

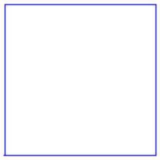


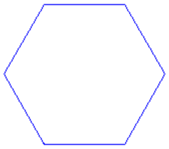
```
when [flag] clicked
hide
clear
go to x: 0 y: 0
pen down
repeat 3
    move 150 steps
    turn ↺ 120 degrees
```

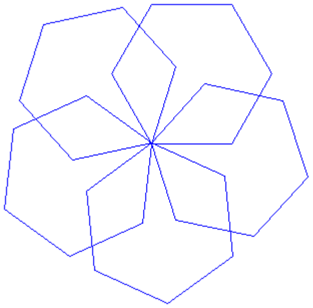
What Changes Are Needed to Draw Each of These?



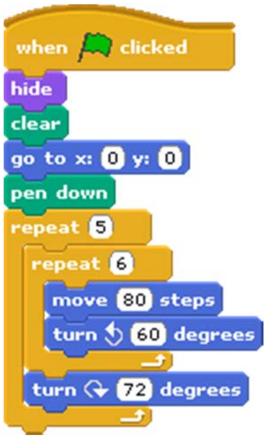
```
when [flag] clicked
hide
clear
go to x: 0 y: 0
pen down
repeat 3
    move 150 steps
    turn ↺ 120 degrees
```







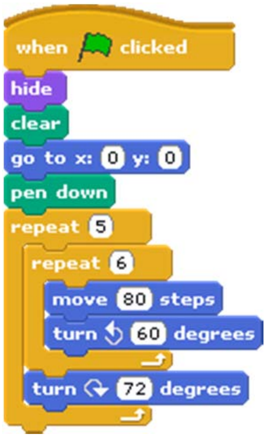
Flow of Control: Repeating a Repetition!



```
when [flag] clicked
hide
clear
go to x: 0 y: 0
pen down
repeat 5
    repeat 6
        move 80 steps
        turn ↺ 60 degrees
    turn ↻ 72 degrees
```

- One loop inside another loop!
    - known as a *nested loop*
- How many times is the *move* statement executed above?

Making Our Program Easier to Change



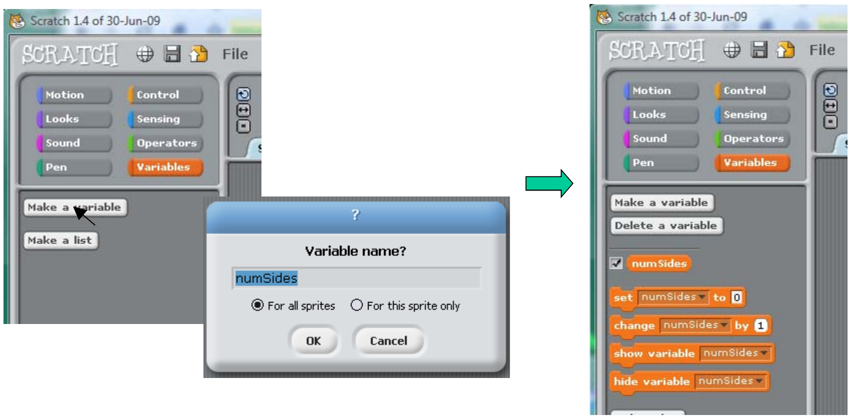
```
when [flag] clicked
hide
clear
go to x: 0 y: 0
pen down
repeat 5
    repeat 6
        move 80 steps
        turn ↺ 60 degrees
    turn ↻ 72 degrees
```

- It would be nice to avoid having to manually change *all* of the numbers.
- Take advantage of relationships between the numbers.
  - what are they?

Program Building Blocks: Variables

- A *variable* is a named location in the computer's memory that is used to store a value.
- Can picture it as a named box:

- To create a variable:

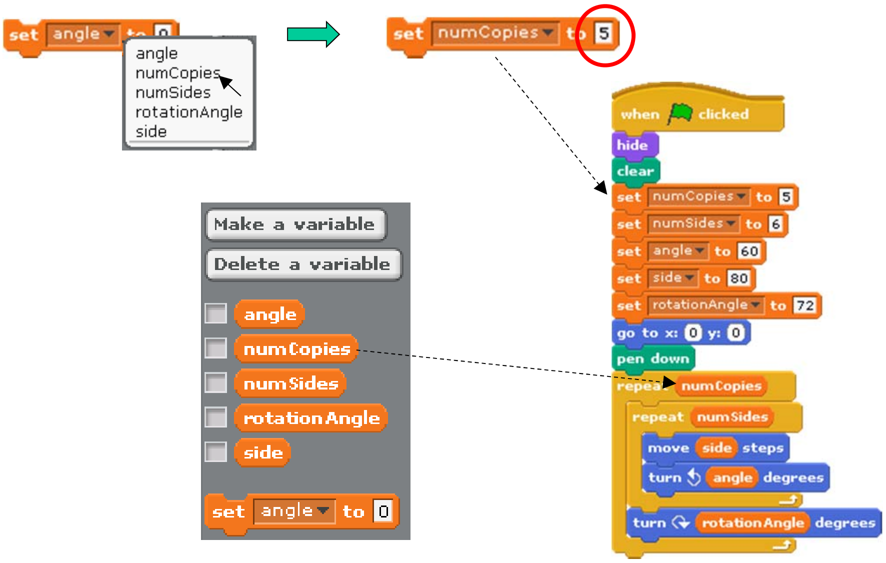


Variable name?

numSides

● For all sprites ○ For this sprite only

Using Variables in Your Program

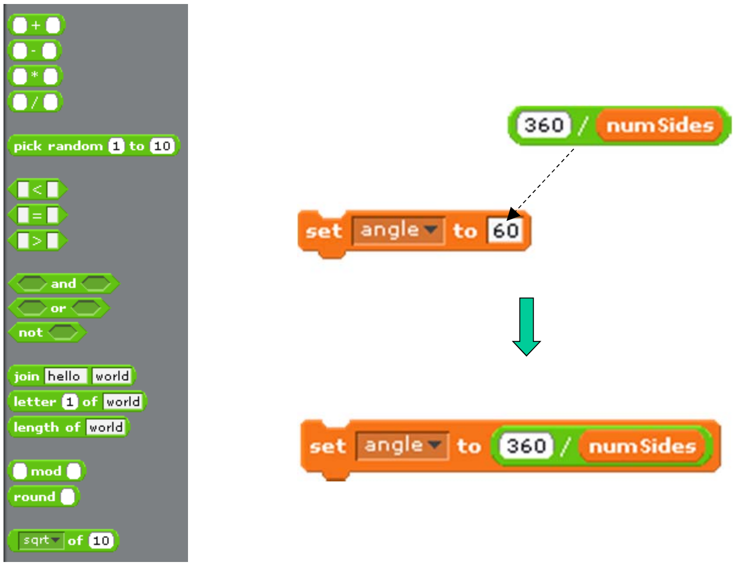


```
when [flag] clicked
hide
clear
set numCopies to 5
set numSides to 6
set angle to 60
set side to 80
set rotationAngle to 72
go to x: 0 y: 0
pen down
repeat numCopies
    repeat numSides
        move side steps
        turn ↺ angle degrees
    turn ↻ rotationAngle degrees
```

note: you must drag a variable into place, not type its name

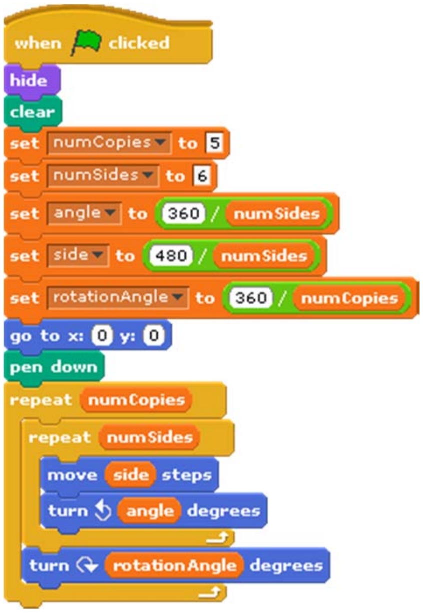
Program Building Blocks: Operators

- Operators create a new value from existing values/variables.



```
360 / numSides

set angle to 60

set angle to (360 / numSides)
```

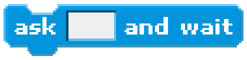
Our Program with Variables and Operators



```
when [flag] clicked
hide
clear
set numCopies to 5
set numSides to 6
set angle to (360 / numSides)
set side to (480 / numSides)
set rotationAngle to (360 / numCopies)
go to x: 0 y: 0
pen down
repeat numCopies
    repeat numSides
        move side steps
        turn ↺ angle degrees
    turn ↻ rotationAngle degrees
```

Getting User Input

- Use the *ask* command from the *sensing* category.

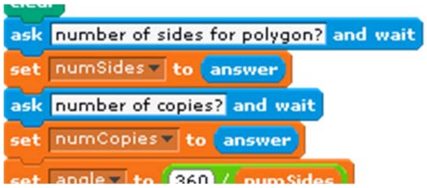


```
ask [ ] and wait
```

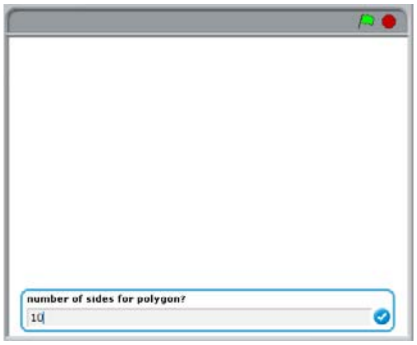
- The value entered by the user is stored in the special variable *answer*, which is also located in the sensing category.

```
answer
```

- Allowing the user to enter *numSides* and *numCopies*:



```
clear
ask number of sides for polygon? and wait
set numSides to answer
ask number of copies? and wait
set numCopies to answer
set angle to (360 / numSides)
```



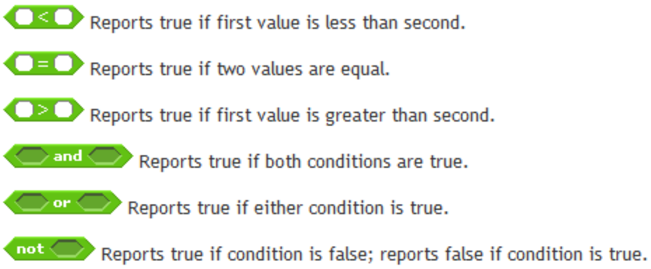
number of sides for polygon?

10

Program Building Blocks: Boolean Expressions

- Blocks with pointed edges produce *boolean* values:
  - *true* or *false*

- Boolean operators:



Reports true if first value is less than second.

Reports true if two values are equal.

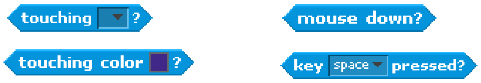
Reports true if first value is greater than second.

Reports true if both conditions are true.

Reports true if either condition is true.

Reports true if condition is false; reports false if condition is true.

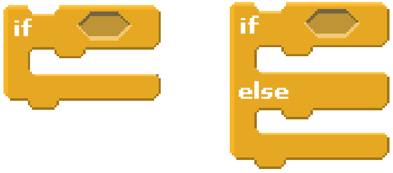
- There are also boolean expressions in the sensing palette:



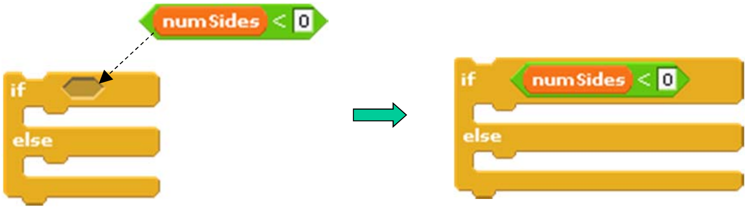
Flow of Control: Conditional Execution

- conditional execution = deciding whether to execute one or more statements on the basis of some condition

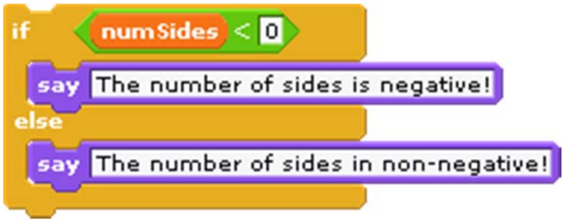
- There are C-shaped control blocks for this:



- They have an input area with pointed edges for the condition.



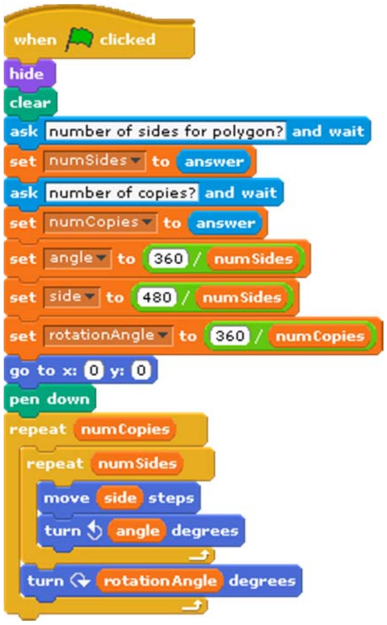
Flow of Control: Conditional Execution (cont.)



```
if numSides < 0
    say The number of sides is negative!
else
    say The number of sides in non-negative!
```

- If the condition is true:
  - the statements under the *if* are executed
  - the statements under the *else* are not executed
- If the condition is false:
  - the statements under the *if* are not executed
  - the statements under the *else* are executed

Dealing With Invalid User Inputs



```
when clicked
hide
clear
ask number of sides for polygon? and wait
set numSides to answer
ask number of copies? and wait
set numCopies to answer
set angle to 360 / numSides
set side to 480 / numSides
set rotationAngle to 360 / numCopies
go to x: 0 y: 0
pen down
repeat numCopies
    repeat numSides
        move side steps
        turn ↻ angle degrees
    turn ↺ rotationAngle degrees
```

More Info on Scratch

- Documentation is available online:
  - getting started guide for version 1.4: http://download.scratch.mit.edu/ScratchGettingStartedv14.pdf
  - reference guide for version 1.4: http://download.scratch.mit.edu/ScratchReferenceGuide14.pdf

- Creating a Scratch account is not required for this course.

Unit 1, Part 2

Programming in Java

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Programs and Classes

- In Java, all programs consist of one of more *classes*.
- For now:
  - we'll limit ourselves to writing a single class
  - you can just think of a class as a container for your program
- Example: our earlier program:

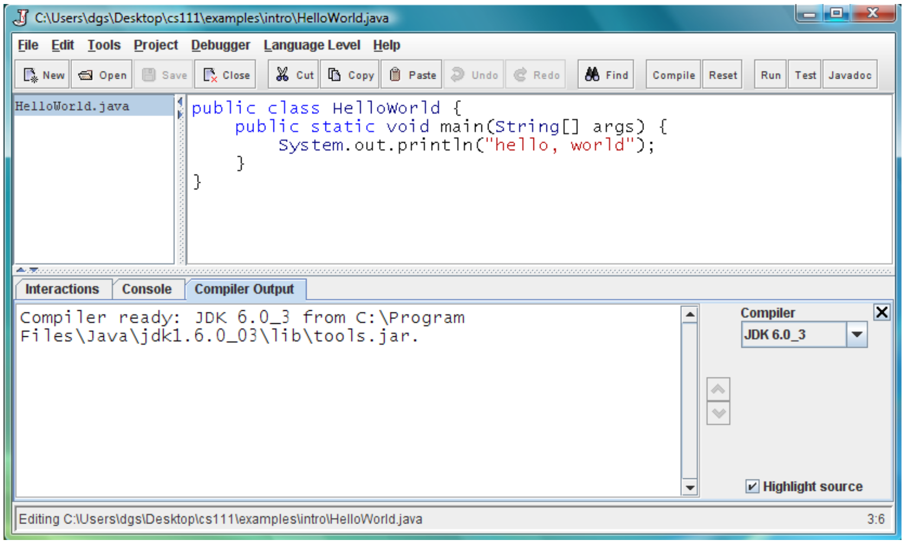
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello, world");
    }
}
```

- A class must be defined in a file with a name of the form <classname>`.java`
  - for the class above, the name would be `HelloWorld.java`

Using an IDE

- An *integrated development environment (IDE)* is an application that helps you develop programs.
- We'll use the Dr. Java IDE.
  - PS 0 told you how to obtain and install it.
- With an IDE, you do the following:
  - use its built-in text editor to write your code
  - instruct the IDE to compile the code
    - turns it into lower-level instructions that can be run
    - checks for violations of the syntax of the language
  - instruct the IDE to run the program
  - debug as needed, using the IDE's debugging tools

Using Dr. Java



```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello, world");
    }
}
```

```
Compiler ready: JDK 6.0_3 from C:\Program
Files\Java\jdk1.6.0_03\lib\tools.jar.
```

Compiler: JDK 6.0_3

☑ Highlight source

Editing C:\Users\dgs\Desktop\cs111\examples\intro\HelloWorld.java 3:6

Format of a Java Class

- General syntax:

```
public class <name> {

        code goes here...

}
```

  where <name> is replaced by the name of the class.

- Notes:
  - the class begins with a *header*:

    `public class <name>`
  - the code inside the class is enclosed in curly braces (`{` and `}`)

Methods

- A method is a collection of instructions that perform some action or computation.
- Every Java program must include a method called `main`.
  - contains the instructions that will be executed first when the program is run
- Our example program includes a `main` method with a single instruction:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello, world");
    }
}
```

Methods (cont.)

- General syntax for the `main` method:

```
public static void main(String[] args) {
    <statement>;
    <statement>;
    …
    <statement>;
}
```

  where each <statement> is replaced by a single instruction.

- Notes:
  - the main method always begins with the same *header*:
    `public static void main(String[] args)`
  - the code inside the method is enclosed in curly braces
  - each statement typically ends with a semi-colon
  - the statements are executed sequentially

Identifiers

- Used to name the components of a Java program like classes and methods.

- Rules:
  - must begin with a letter (a-z, A-Z), $, or _
  - can be followed by any number of letters, numbers, $, or _
  - spaces are not allowed
  - cannot be the same as a *keyword* – a word like `class` that is part of the language itself (see the book)

- Which of these are *not* valid identifiers?

```
n1              num_values      2n
avgSalary       course name
```

- Java is *case-sensitive* (for both identifiers and keywords).
  - example: `HelloWorld` is not the same as `helloWorld`

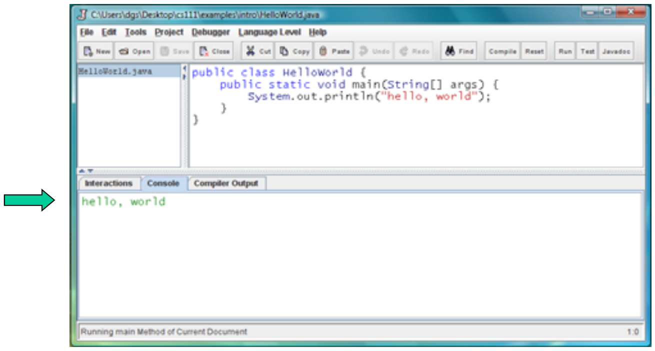
Conventions for Identifiers

- Capitalize class names.
  - example: `HelloWorld`
- Do not capitalize method names.
  - example: `main`
- Capitalize internal words within the name.
  - example: `HelloWorld`

Printing Text

```
 public class HelloWorld {
     public static void main(String[] args) {
         System.out.println("hello, world");
     }
}
```

- Our program contains a single statement that prints some text.
- The printed text appears in a window known as the *console*.


C:\Users\dgs\Desktop\cs111\examples\intro\HelloWorld.java
File Edit Tools Project Debugger Language Level Help
New Open Save Close Cut Copy Paste Undo Redo Find Compile Reset Run Test Javadoc
HelloWorld.java
Interactions Console Compiler Output
Running main Method of Current Document
1:0

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello, world");
    }
}
```

```
hello, world
```

Printing Text (cont.)

- The general format of such statements is:

  ```
  System.out.println("<text>");
  ```

  where <text> is replaced by the text you want to print.

- A piece of text like `"Hello, world"` is referred to as a *string literal.*
  - string: a collection of characters
  - literal: specified explicitly in the program ("hard-coded")

- A string literal must be enclosed in double quotes.

- You can print a blank line by omitting the string literal:

  ```
  System.out.println();
  ```

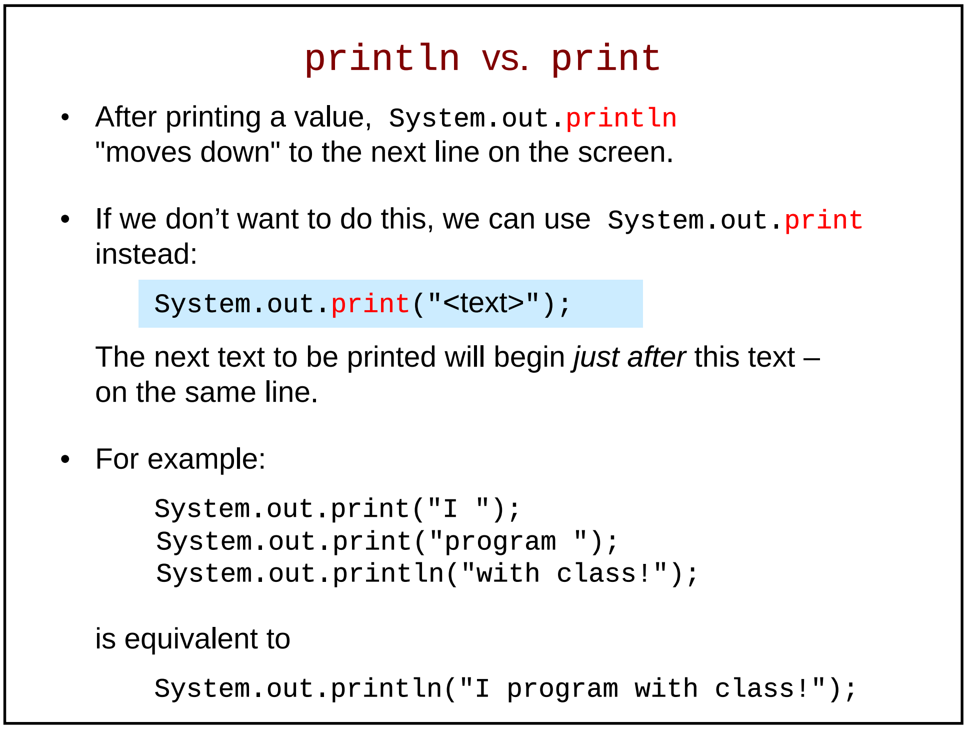
Printing Text (cont.)

- A string literal cannot span multiple lines.
  - example: this is *not* allowed:

    ```
    System.out.println("I want to print a string
      on two lines.");
    ```

- Instead, we can use two different statements:

  ```
  System.out.println("I want to print a string");
  System.out.println("on two lines.");
  ```



`println` vs. `print`

- After printing a value, `System.out.println` "moves down" to the next line on the screen.
- If we don't want to do this, we can use `System.out.print` instead:

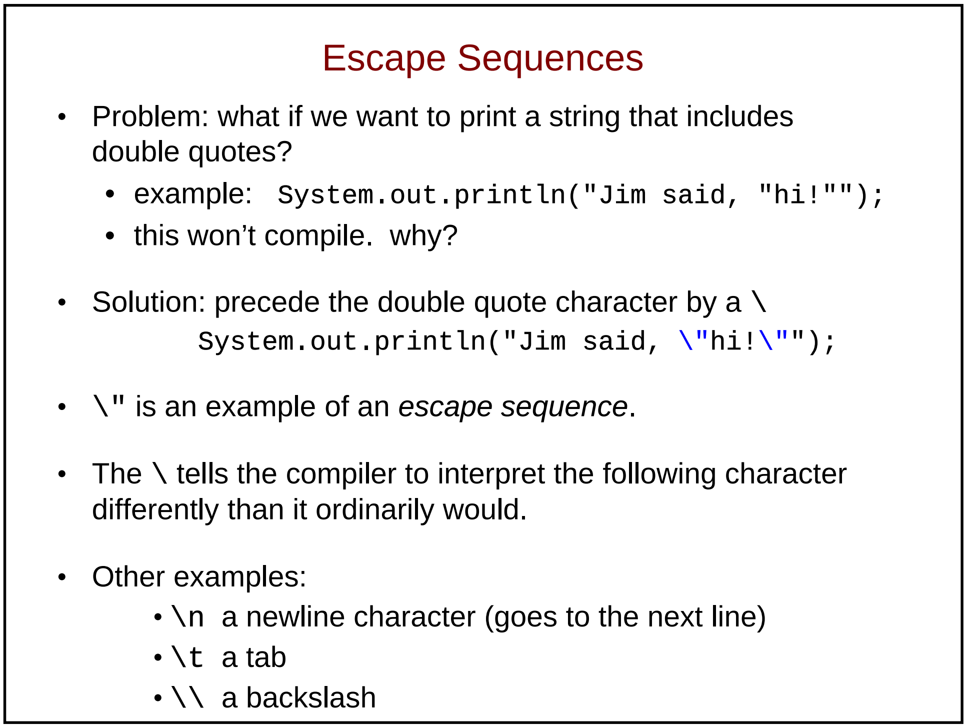
  ```
  System.out.print("<text>");
  ```

  The next text to be printed will begin *just after* this text – on the same line.
- For example:

  ```
  System.out.print("I ");
  System.out.print("program ");
  System.out.println("with class!");
  ```

  is equivalent to

  ```
  System.out.println("I program with class!");
  ```



Escape Sequences

- Problem: what if we want to print a string that includes double quotes?
  - example: `System.out.println("Jim said, "hi!"");`
  - this won't compile. why?
- Solution: precede the double quote character by a \

  ```
  System.out.println("Jim said, \"hi!\"");
  ```
- `\"` is an example of an *escape sequence*.
- The \ tells the compiler to interpret the following character differently than it ordinarily would.
- Other examples:
  - `\n` a newline character (goes to the next line)
  - `\t` a tab
  - `\\` a backslash

Unit 1, Part 3

Procedural Decomposition

(How to Use Methods to Write Better Programs)

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Example Program: Writing Block Letters

- Here's a program that writes the name "DEE" in block letters:

```
public class BlockLetters {
    public static void main(String[] args) {
        System.out.println("    -----");
        System.out.println("     |   \\");
        System.out.println("     |    |");
        System.out.println("     |   /");
        System.out.println("    -----");
        System.out.println();
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
        System.out.println();
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
    }
}
```

Example Program: Writing Block Letters

- The output looks like this:

```
-----
 |   \
 |    |
 |   /
-----

+-----
|
+----
|
+-----

+-----
|
+----
|
+-----
```

Code Duplication

```
public class BlockLetters {
    public static void main(String[] args) {
        System.out.println("    -----");
        System.out.println("     |   \\");
        System.out.println("     |    |");
        System.out.println("     |   /");
        System.out.println("    -----");
        System.out.println();
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
        System.out.println();
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
    }
}
```

- The code that writes an E appears twice – it is duplicated.

Code Duplication (cont.)

- Code duplication is undesirable. Why?
- Also, what if we wanted to create another word containing the letters D or E? What would we need to do?
- A better approach: create a command for writing each letter, and execute that command as needed.
- To create our own command in Java, we define a method.

Defining a Simple Static Method

- We've already seen how to define a `main` method:

```
public static void main(String[] args) {
    <statement>;
    <statement>;
    …
    <statement>;
}
```

- The simple methods that we'll define have a similar syntax:

```
public static void <name>() {
    <statement>;
    <statement>;
    …
    <statement>;
}
```

- This type of method is known as *static method.*

Defining a Simple Static Method (cont.)

- Here's a static method for writing a block letter E:

```
public static void writeE() {
    System.out.println("    +-----");
    System.out.println("    |");
    System.out.println("    +----");
    System.out.println("    |");
    System.out.println("    +-----");
}
```

- It contains the same statements that we used to write an E in our earlier program.
- This method gives us a command for writing an E.
- To use it, we simply include the following statement:

```
writeE();
```

Calling a Method

- The statement

```
writeE();
```

  is known as a *method call.*

- General syntax for a static method call:

  <method-name>`();`

- Calling a method causes the statements inside the method to be executed.

Using Methods to Eliminate Duplication

- Here's a revised version of our program:

```
public class BlockLetters2 {
    public static void writeE() {
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
    }

    public static void main(String[] args) {
        System.out.println("    -----");
        System.out.println("     |   \\");
        System.out.println("     |    |");
        System.out.println("     |   /");
        System.out.println("    -----");
        System.out.println();
        writeE();
        System.out.println();
        writeE();
    }
}
```

Methods Can Be Defined In Any Order

- Here's a version in which we put the `main` method first:

```
public class BlockLetters2 {
    public static void main(String[] args) {
        System.out.println("    -----");
        System.out.println("     |   \\");
        System.out.println("     |    |");
        System.out.println("     |   /");
        System.out.println("    -----");
        System.out.println();
        writeE();
        System.out.println();
        writeE();
    }

    public static void writeE() {
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
    }
}
```

- By convention, the `main` method should appear first or last.

Flow of Control

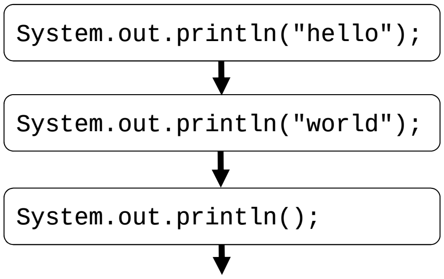
- A program's *flow of control* is the order in which its statements are executed.
- By default, the flow of control:
  - is sequential
  - begins with the first statement in the `main` method

Flow of Control (cont.)

- Example: consider the following program:

```
public class HelloWorldAgain {
    public static void main(String[] args) {
        System.out.println("hello");
        System.out.println("world");
        System.out.println();
        ...
    }
}
```

- We can represent the flow of control using a flow chart:



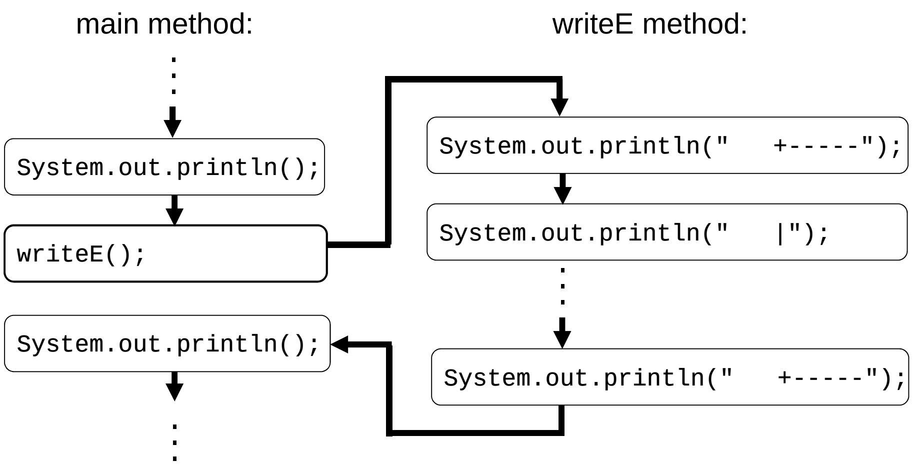
Method Calls and Flow of Control

- When we call a method, the flow of control jumps to the method.
- After the method completes, the flow of control jumps back to the point where the method call was made.

```
public class BlockLetters2 {
    public static void writeE() {
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
    }

    public static void main(String[] args) {
        System.out.println("    -----");
        System.out.println("     |   \\");
        System.out.println("     |    |");
        System.out.println("     |   /");
        System.out.println("    -----");
        System.out.println();
        writeE();
        System.out.println();
        ...
```

Method Calls and Flow of Control (cont.)

- Here's a portion of the flowchart for our program:


main method:
writeE method:
System.out.println();
writeE();
System.out.println();
System.out.println("   +-----");
System.out.println(" |");
System.out.println("   +-----");

Another Use of a Static Method

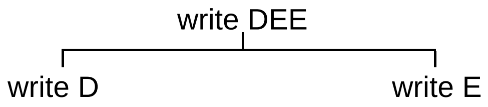
```
public class BlockLetters3 {
    public static void writeD() {
        System.out.println("    -----");
        System.out.println("     |   \\");
        System.out.println("     |    |");
        System.out.println("     |   /");
        System.out.println("    -----");
    }

    public static void writeE() {
        System.out.println("    +-----");
        System.out.println("    |");
        System.out.println("    +----");
        System.out.println("    |");
        System.out.println("    +-----");
    }

    public static void main(String[] args) {
        writeD();
        System.out.println();
        writeE();
        System.out.println();
        writeE();
    }
}
```

Another Use of a Static Method (cont.)

- The code in the `writeD` method is only used once, so it doesn't eliminate code duplication.
- However, using a separate static method still makes the overall program more readable.
- It helps to reveal the *structure* of the program.

Procedural Decomposition

- In general, methods allow us to *decompose* a problem into smaller subproblems that are easier to solve.
  - the resulting code is also easier to understand and maintain
- In our program, we've decomposed the task "write DEE" into two subtasks:
  - write D
  - write E (which we perform twice).
- We can use a *structure diagram* to show the decomposition:



Procedural Decomposition (cont.)

- How could we use procedural decomposition in printing the following lyrics?

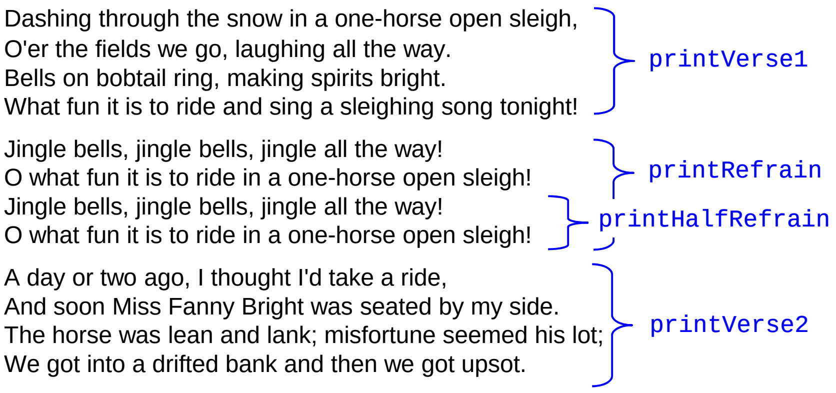
Dashing through the snow in a one-horse open sleigh,
O'er the fields we go, laughing all the way.
Bells on bobtail ring, making spirits bright.
What fun it is to ride and sing a sleighing song tonight!

Jingle bells, jingle bells, jingle all the way!
O what fun it is to ride in a one-horse open sleigh!
Jingle bells, jingle bells, jingle all the way!
O what fun it is to ride in a one-horse open sleigh!

A day or two ago, I thought I'd take a ride,
And soon Miss Fanny Bright was seated by my side.
The horse was lean and lank; misfortune seemed his lot;
We got into a drifted bank and then we got upsot.

Jingle bells, jingle bells, jingle all the way!
O what fun it is to ride in a one-horse open sleigh!
Jingle bells, jingle bells, jingle all the way!
O what fun it is to ride in a one-horse open sleigh!

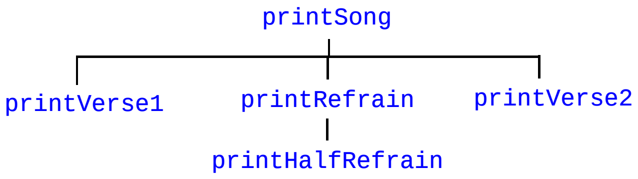
Procedural Decomposition (cont.)



Dashing through the snow in a one-horse open sleigh,
O'er the fields we go, laughing all the way.
Bells on bobtail ring, making spirits bright.
What fun it is to ride and sing a sleighing song tonight!

Jingle bells, jingle bells, jingle all the way!
O what fun it is to ride in a one-horse open sleigh!
Jingle bells, jingle bells, jingle all the way!
O what fun it is to ride in a one-horse open sleigh!

A day or two ago, I thought I'd take a ride,
And soon Miss Fanny Bright was seated by my side.
The horse was lean and lank; misfortune seemed his lot;
We got into a drifted bank and then we got upsot.



Code Reuse

- Once we have a set of methods, we can use them to solve other problems.
- Here's a program that writes the name "ED":

```
public class BlockLetters4 {
    // these methods are the same as before
    public static void writeD() {
        ...
    }
    public static void writeE() {
        ...
    }
    public static void main(String[] args) {
        writeE();
        System.out.println();
        writeD();
    }
}
```

Tracing the Flow of Control

- What is the output of the following program?

```
public class FlowControlTest {
    public static void methodA() {
         System.out.println("starting method A");
    }

    public static void methodB() {
         System.out.println("starting method B");
    }

    public static void methodC() {
         System.out.println("starting method C");
    }

    public static void main(String[] args) {
         methodC();
         methodA();
    }
}
```

Methods Calling Methods

- The definition of one method can include calls to other methods.
- We've seen this already in the `main` method:

```
public static void main(String[] args) {
     writeE();
     System.out.println();
     writeD();
}
```

- We can also do this in other methods:

```
public static void foo() {
     System.out.println("This is method foo.");
     bar();
}

public static void bar() {
     System.out.println("This is method bar.");
}
```

Methods Calling Methods (cont.)

- What is the output of the following program?

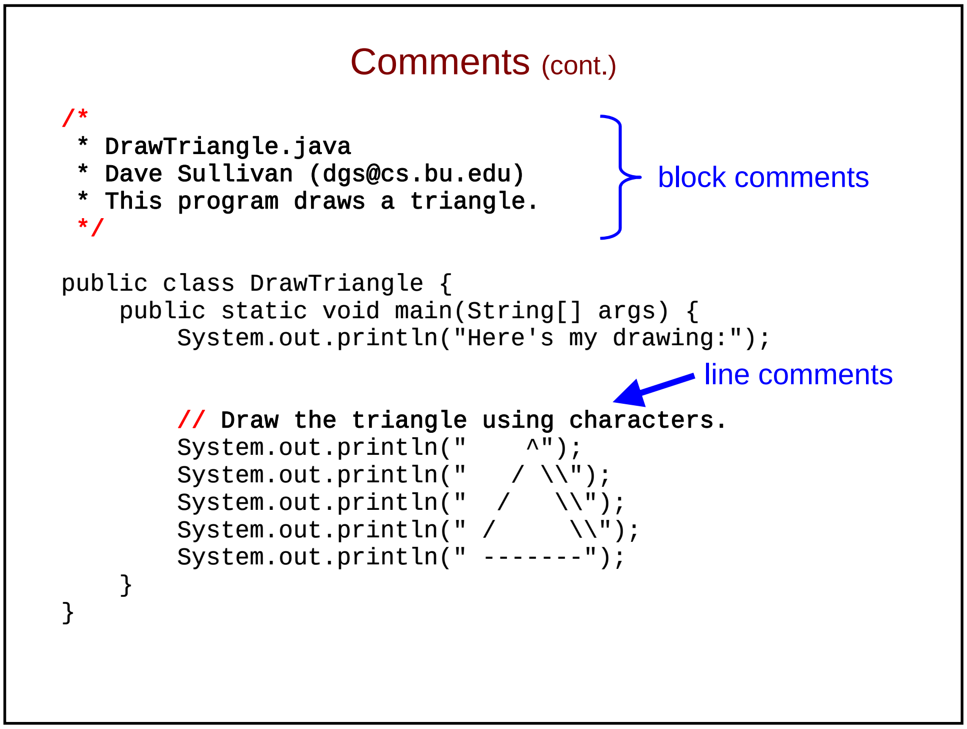
```
public class FlowControlTest2 {
    public static void methodOne() {
         System.out.println("boo");
         methodThree();
    }

    public static void methodTwo() {
         System.out.println("hoo");
         methodOne();
    }

    public static void methodThree() {
         System.out.println("foo");
    }

    public static void main(String[] args) {
         methodOne();
         methodThree();
         methodTwo();
    }
}
```

Comments

- Comments are text that is ignored by the compiler.
- Used to make programs more readable
- Two types:
  1. line comments: begin with `//`
     - compiler ignores from `//` to the end of the line
     - examples:

```
// this is a comment
System.out.println();  // so is this
```

  2. block comments: begin with `/*` and end with `*/`
     - compiler ignores everything in between
     - typically used at the top of each source file

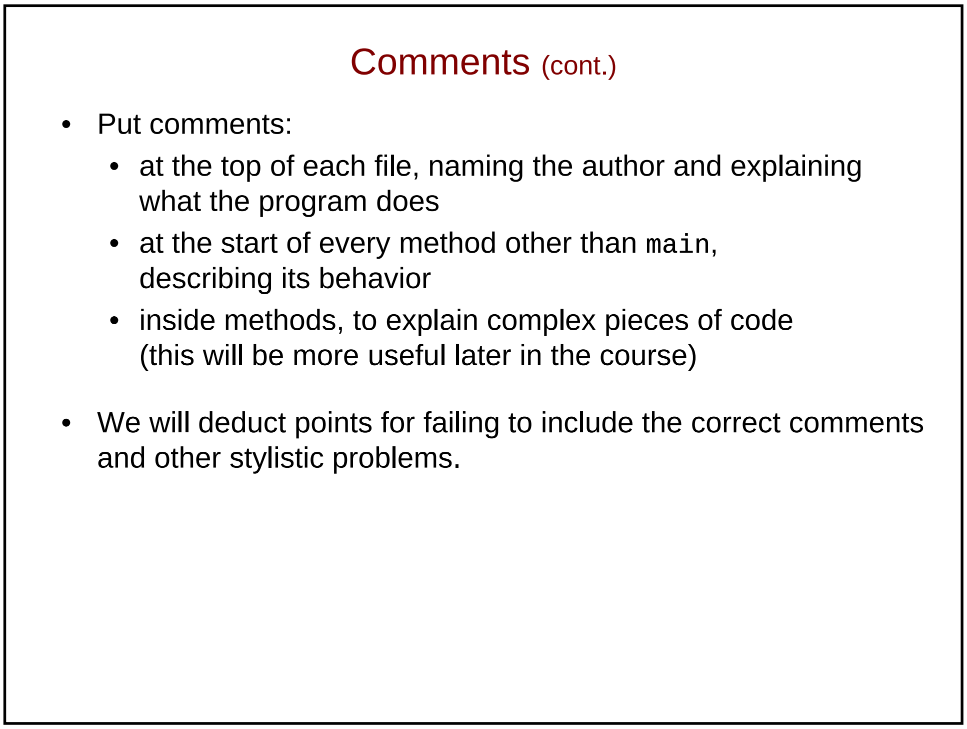


Comments (cont.)

```
/*
 * DrawTriangle.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program draws a triangle.
 */

public class DrawTriangle {
    public static void main(String[] args) {
        System.out.println("Here's my drawing:");

        // Draw the triangle using characters.
        System.out.println("    ^");
        System.out.println("   / \\");
        System.out.println("  /   \\");
        System.out.println(" /     \\");
        System.out.println(" -------");
    }
}
```

block comments

line comments



Comments (cont.)

- Put comments:
  - at the top of each file, naming the author and explaining what the program does
  - at the start of every method other than `main`, describing its behavior
  - inside methods, to explain complex pieces of code (this will be more useful later in the course)
- We will deduct points for failing to include the correct comments and other stylistic problems.

Unit 2, Part 1

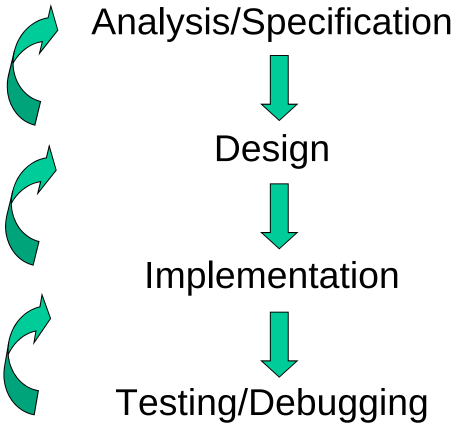
Primitive Data, Variables, and Expressions;

Simple Conditional Execution

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Overview of the Programming Process



Analysis/Specification

Design

Implementation

Testing/Debugging

Example Problem: Adding Up Your Change

- Let's say that we have a bunch of coins of various types, and we want to figure out how much money we have.
- Let’s begin the process of developing a program that does this.

Step 1: Analysis and Specification

- *Analyze* the problem (making sure that you understand it), and *specify* the problem requirements clearly and unambiguously.
- Describe exactly *what* the program will do, without worrying about *how* it will do it.

Step 2: Design

- Determine the necessary algorithms (and possibly other aspects of the program) and sketch out a design for them.
- This is where we figure out *how* the program will solve the problem.
- Algorithms are often designed using *pseudocode.*
  - more informal than an actual programming language
  - allows us to avoid worrying about the *syntax* of the language
  - example for our change-adder problem:

```
get the number of quarters
get the number of dimes
get the number of nickels
get the number of pennies
compute the total value of the coins
output the total value
```

Step 3: Implementation

- Translate your design into the programming language.

  pseudocode → code

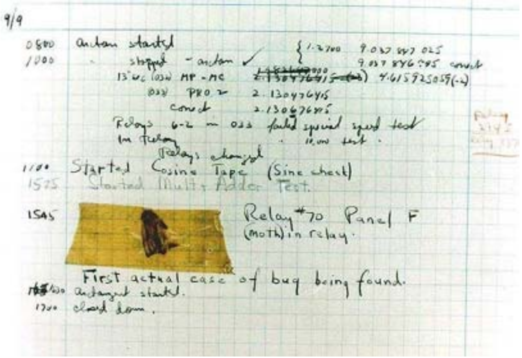
- We need to learn more Java before we can do this!
- Here's a portion or *fragment* of a Java program for computing the value of a particular collection of coins:

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- In a moment, we'll use this fragment to examine some of the fundamental building blocks of a Java program.

Step 4: Testing and Debugging

- A *bug* is an error in your program.
- *Debugging* involves finding and fixing the bugs.



9/9

0800 Antan started

1000 " stopped - antan ✓ {1.2700 9.037 847 025

9.037 846 995 conct

13" uc (033) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)

(033) PRO 2 2.130476415

conct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 11.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

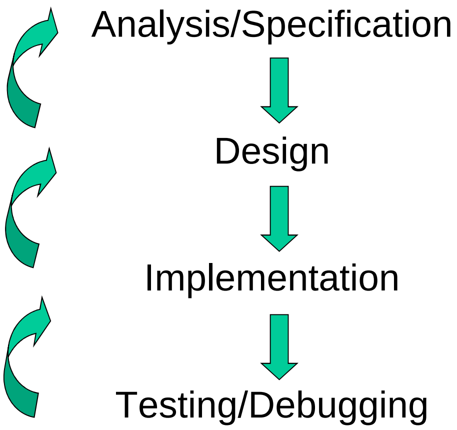
1630 antangent started.

1700 closed down.

The first program bug! Found by Grace Murray Hopper at Harvard.
(http://www.hopper.navy.mil/grace/grace.htm)

- Testing – trying the programs on a variety of inputs – helps us to find the bugs.

Overview of the Programming Process



Analysis/Specification

Design

Implementation

Testing/Debugging

Program Building Blocks: Literals

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- *Literals* specify a particular value.
- They include:
  - string literals: `"Your total in cents is:"`
    - are surrounded by double quotes
  - numeric literals: `25`   `3.1416`
    - commas are not allowed!

Program Building Blocks: Variables

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- We've already seen that variables are named memory locations that are used to store a value:

```
quarters  |  10 |
```

- Variable names must follow the rules for *identifiers* (see previous notes).

Program Building Blocks: Statements

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- In Java, a single-line statement typically ends with a semi-colon.
- Later, we will see examples of control statements that contain other statements, just as we did in Scratch.

Program Building Blocks: Expressions

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
System.out.println("Your total in cents is:");
System.out.println(cents);
```

- *Expressions* are pieces of code that evaluate to a value.
- They include:
  - literals, which evaluate to themselves
  - variables, which evaluate to the value that they represent
  - combinations of literals, variables, and *operators*:

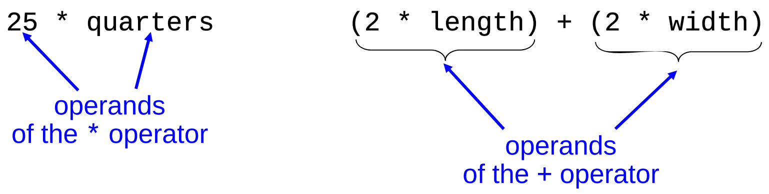
    `25*quarters + 10*dimes + 5*nickels + pennies`

Program Building Blocks: Expressions (cont.)

- Numerical operators include:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus or mod: gives the remainder of a division
	example: `11 % 3` evaluates to `2`

- Operators are applied to *operands:*



```
25 * quarters          (2 * length) + (2 * width)
```

operands
of the `*` operator

operands
of the + operator

Evaluating Expressions

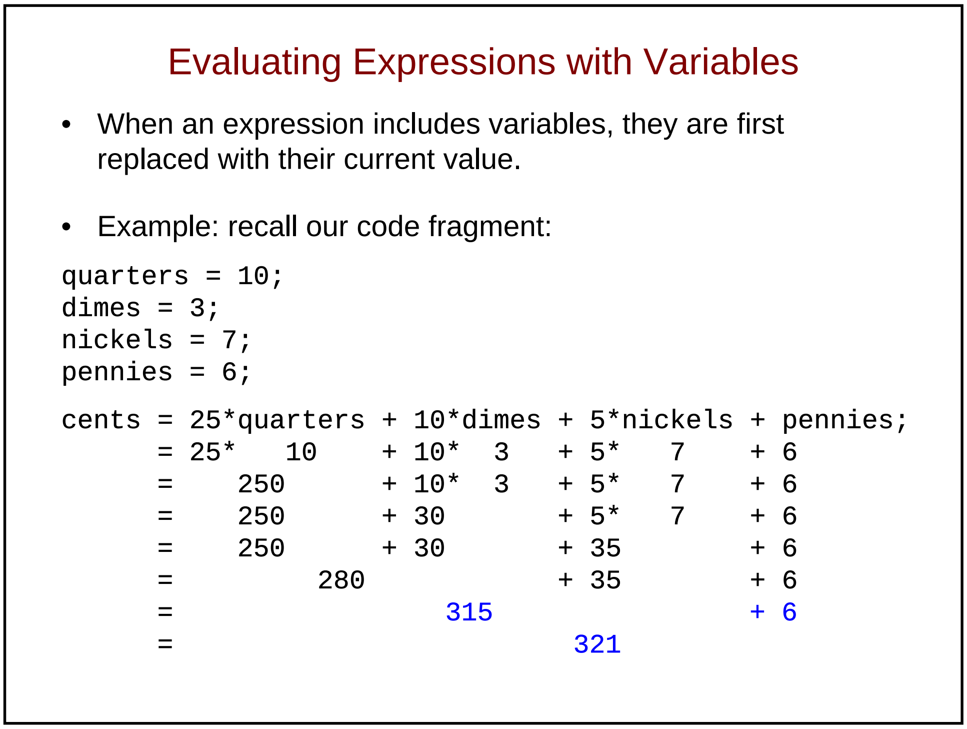
- With expressions that involve more than one mathematical operator, the usual order of operations applies.
  - example:

```
   3 + 4 * 3 / 2 – 7
=
=
=
=
```

- Use parentheses to:
  - force a different order of evaluation
    - example:

```
radius = circumference / (2 * pi);
```

  - make the standard order of operations obvious!

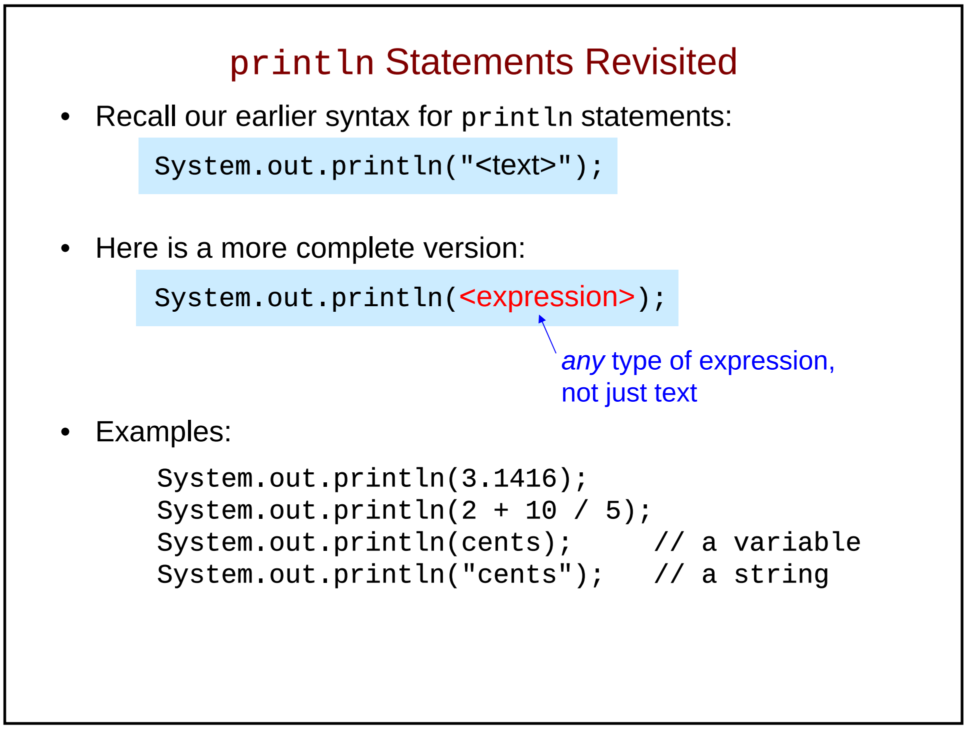


Evaluating Expressions with Variables

- When an expression includes variables, they are first replaced with their current value.

- Example: recall our code fragment:

```
quarters = 10;
dimes = 3;
nickels = 7;
pennies = 6;

cents = 25*quarters + 10*dimes + 5*nickels + pennies;
      = 25*   10    + 10*  3   + 5*   7    + 6
      =    250      + 10*  3   + 5*   7    + 6
      =    250      + 30       + 5*   7    + 6
      =    250      + 30       + 35        + 6
      =         280            + 35        + 6
      =                 315                + 6
      =                         321
```



`println` Statements Revisited

- Recall our earlier syntax for `println` statements:

```
System.out.println("<text>");
```

- Here is a more complete version:

```
System.out.println(<expression>);
```

any type of expression, not just text

- Examples:

```
System.out.println(3.1416);
System.out.println(2 + 10 / 5);
System.out.println(cents);     // a variable
System.out.println("cents");   // a string
```

`println` Statements Revisited (cont.)

- The expression is first evaluated, and then the value is printed.

```
System.out.println(2 + 10 / 5);
                    ↓
System.out.println(4);          // output: 4
----------------------------------------------
System.out.println(cents);
                     ↓
System.out.println(321);        // output: 321
----------------------------------------------
System.out.println("cents");
                       ↓
System.out.println("cents");    // output: cents
```

- Note that the surrounding quotes are *not* displayed when a string is printed.

`println` Statements Revisited (cont.)

- Another example:

```
System.out.println(10*dimes + 5*nickels);
                    ↓
System.out.println(10*3 + 5*7);
                    ↓
System.out.println(65);
```

Expressions in DrJava

- If you enter an expression in the Interactions Pane, DrJava evaluates it and displays the result.
  - examples:

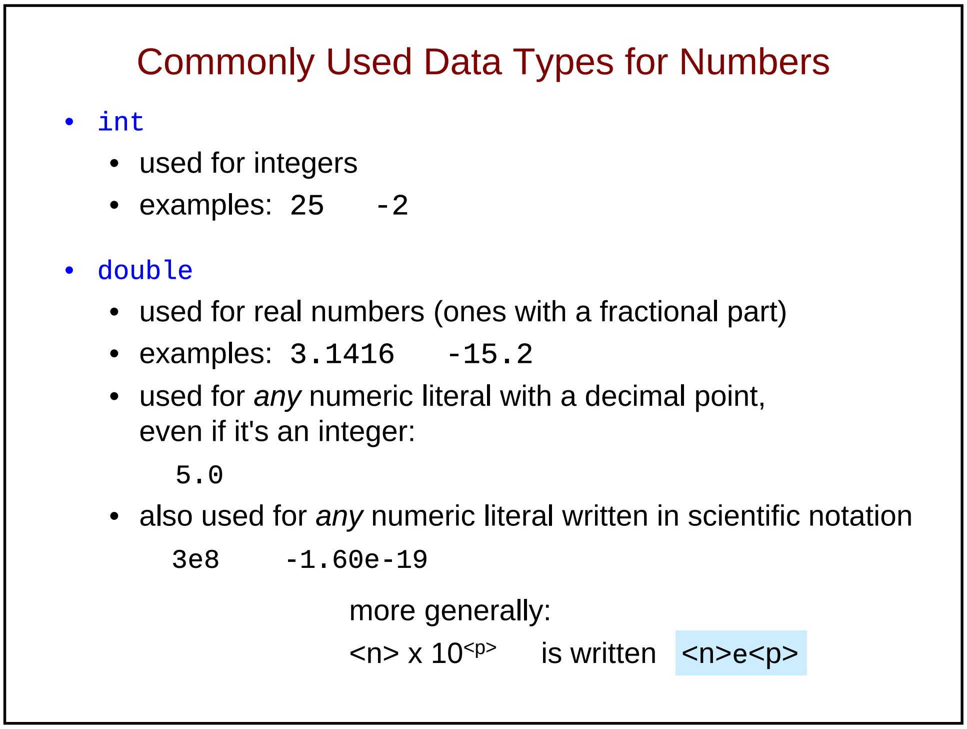
```
> "Hello world!"       // do not put a semi-colon!
"Hello world!"
> 5 + 10
15
> 10 * 4 + 5 * 2
50
> 10 * (4 + 5 * 2)
140
```

- Note: This type of thing does *not* work inside a program.

```
public static void main(String[] args) {
    5 + 10                     // not allowed!
    System.out.println(5 + 10); // do this instead
}
```

Data Types

- A *data type* is a set of related data values.
  - examples:
    - integers
    - strings
    - characters
- Every data type in Java has a name that we can use to identify it.



Commonly Used Data Types for Numbers

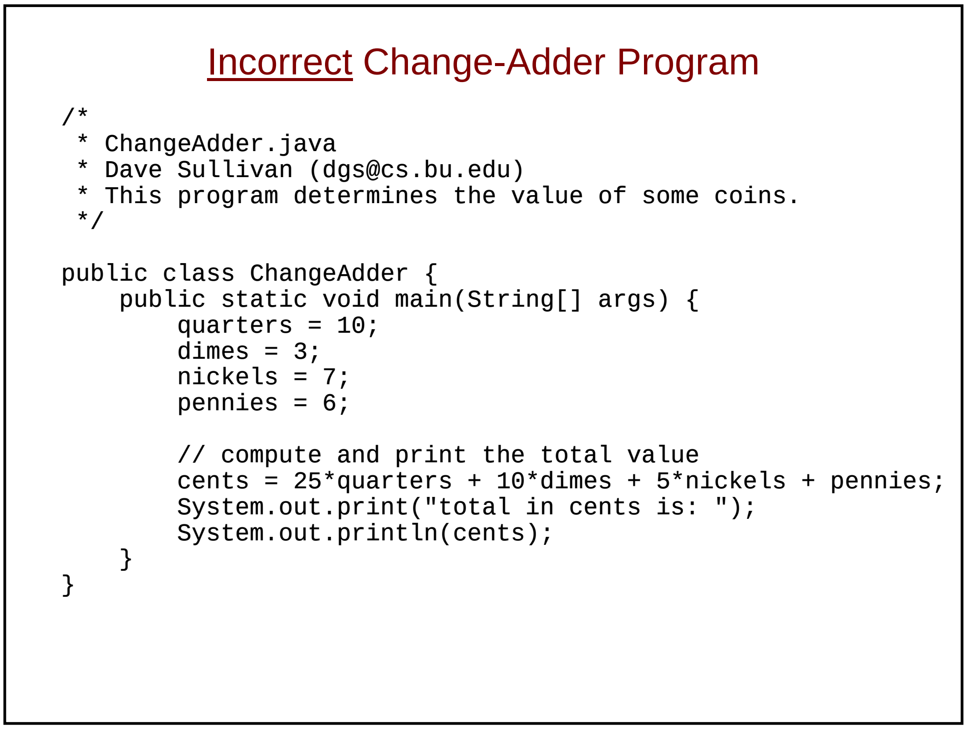
- `int`
  - used for integers
  - examples: `25` `-2`
- `double`
  - used for real numbers (ones with a fractional part)
  - examples: `3.1416` `-15.2`
  - used for *any* numeric literal with a decimal point, even if it's an integer:

    `5.0`
  - also used for *any* numeric literal written in scientific notation

    `3e8` `-1.60e-19`

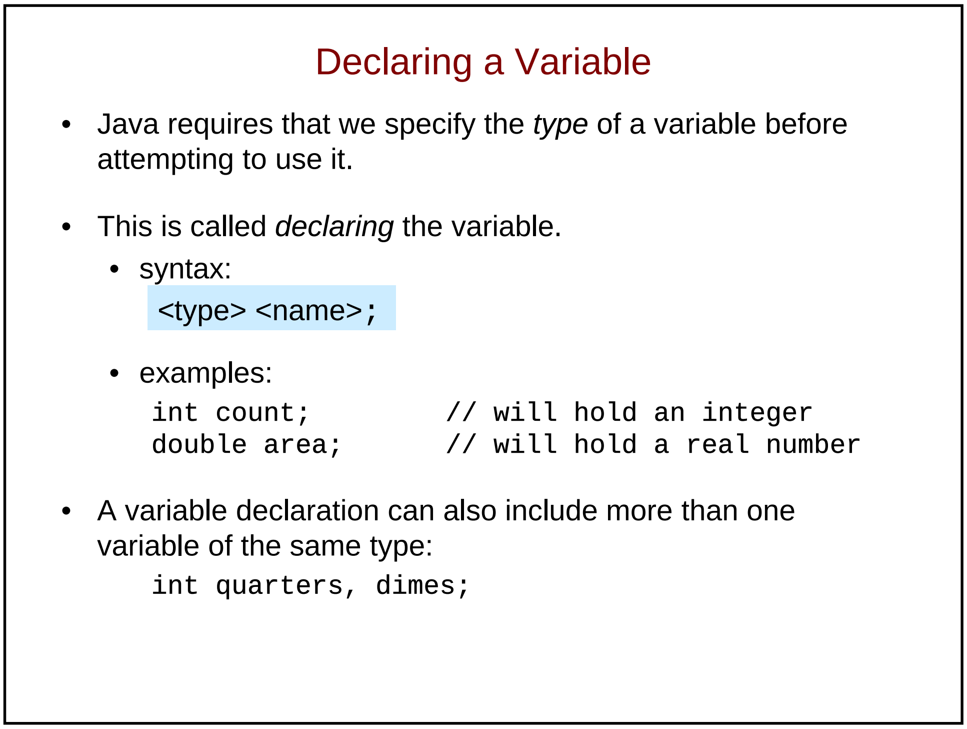
more generally:

<n> x $10^{<p>}$ is written <n>e<p>



Incorrect Change-Adder Program

```
/*
 * ChangeAdder.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder {
    public static void main(String[] args) {
        quarters = 10;
        dimes = 3;
        nickels = 7;
        pennies = 6;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
    }
}
```



Declaring a Variable

- Java requires that we specify the *type* of a variable before attempting to use it.
- This is called *declaring* the variable.
  - syntax:

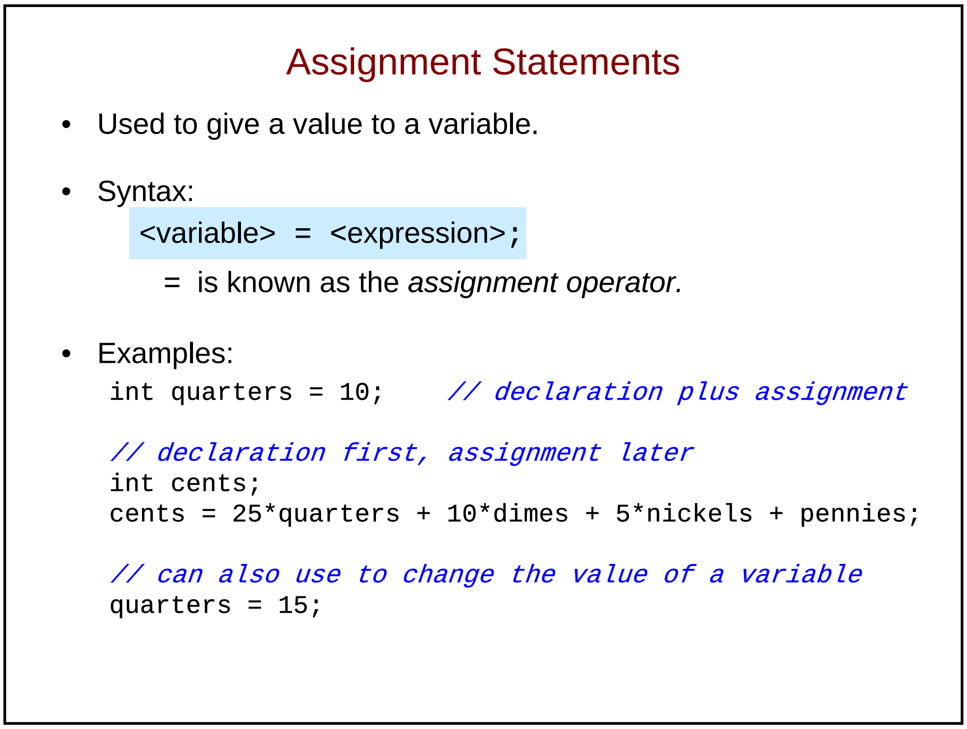
    <type> <name>;

  - examples:

```
int count;          // will hold an integer
double area;        // will hold a real number
```

- A variable declaration can also include more than one variable of the same type:

```
int quarters, dimes;
```



Assignment Statements

- Used to give a value to a variable.
- Syntax:

  <variable> = <expression>;

  = is known as the *assignment operator.*

- Examples:

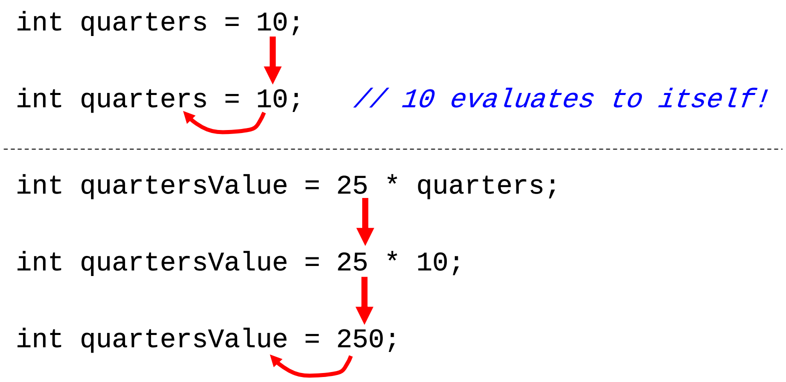
```
int quarters = 10;     // declaration plus assignment

// declaration first, assignment later
int cents;
cents = 25*quarters + 10*dimes + 5*nickels + pennies;

// can also use to change the value of a variable
quarters = 15;
```

Corrected Change-Adder Program

```
/*
 * ChangeAdder.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
    }
}
```

Assignment Statements (cont.)

- Steps in executing an assignment statement:
  1) evaluate the expression on the right-hand side of the =
  2) assign the resulting value to the variable on the left-hand side of the =
- Examples:



```
int quarters = 10;

int quarters = 10;    // 10 evaluates to itself!
```

```
int quartersValue = 25 * quarters;

int quartersValue = 25 * 10;

int quartersValue = 250;
```

Assignment Statements (cont.)

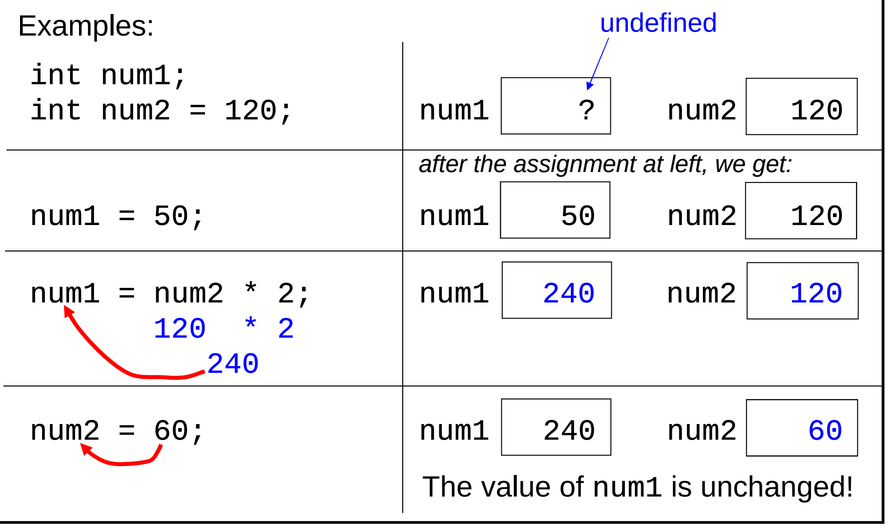
- An assignment statement does not create a permanent relationship between variables.
- Example from the DrJava Interactions Pane:

```
> int x = 10;
> int y = x + 2;
> y
12
> x = 20;
> y
12
```

  - changing the value of x does *not* change the value of y!
- You can only change the value of a variable by assigning it a new value.

Assignment Statements (cont.)

- As the values of variables change, it can be helpful to picture what's happening in memory.
- Examples:



`int num1;`
`int num2 = 120;`
	undefined
	num1 ? num2 120

`num1 = 50;`	*after the assignment at left, we get:* num1 50 num2 120

`num1 = num2 * 2;`
`120 * 2`
`240`
	num1 240 num2 120

`num2 = 60;`	num1 240 num2 60 The value of `num1` is unchanged!

Assignment Statements (cont.)

- A variable can appear on both sides of the assignment operator!
- Example (fill in the missing values):

`int sum = 13;`
`int val = 30;`
	`sum`	13	`val`	30

`sum = sum + val;`	`sum`		`val`	
`val = val * 2;`	`sum`		`val`	

Operators and Data Types

- Each data type has its own set of operators.
  - the `int` version of an operator produces an `int` result
  - the `double` version produces a `double` result
  - etc.
- Rules for numeric operators:
  - if the operands are both of type `int`,
    the `int` version of the operator is used.
    - examples: `15 + 30`
      `1 / 2`
      `25 * quarters`
  - if at least one of the operands is of type `double`,
    the `double` version of the operator is used.
    - examples: `15.5 + 30.1`
      `1 / 2.0`
      `25.0 * quarters`

Incorrect Extended Change-Adder Program

```
/*
 * ChangeAdder2.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder2 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
        double dollars = cents / 100;
        System.out.print("total in dollars is: ");
        System.out.println(dollars);
    }
}
```

Two Types of Division

- The `int` version of the `/` operator performs *integer division*, which discards the fractional part of the result (i.e., everything after the decimal).
  - examples:

```
> 5 / 3
1

> 11 / 5
2
```

- The `double` version of the `/` operator performs *floating-point division*, which keeps the fractional part.
  - examples:

```
> 5.0 / 3.0
1.6666666666666667

> 11 / 5.0
2.2
```

How Can We Fix Our Program?

```
/*
 * ChangeAdder2.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder2 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.print("total in cents is: ");
        System.out.println(cents);
        double dollars = cents / 100;
        System.out.print("total in dollars is: ");
        System.out.println(dollars);
    }
}
```

String Concatenation

- The meaning of the + operator depends on the types of the operands.
- When at least one of the operands is a string, the + operator performs string concatenation.
  - combines two or more strings into a single string
  - example:

    ```
    System.out.println("hello " + "world");
    ```

    is equivalent to

    ```
    System.out.println("hello world");
    ```

String Concatenation (cont.)

- If one operand is a string and the other is a number, the number is converted to a string and then concatenated.
  - example: instead of writing

    ```
    System.out.print("total in cents: ");
    System.out.println(cents);
    ```

    we can write

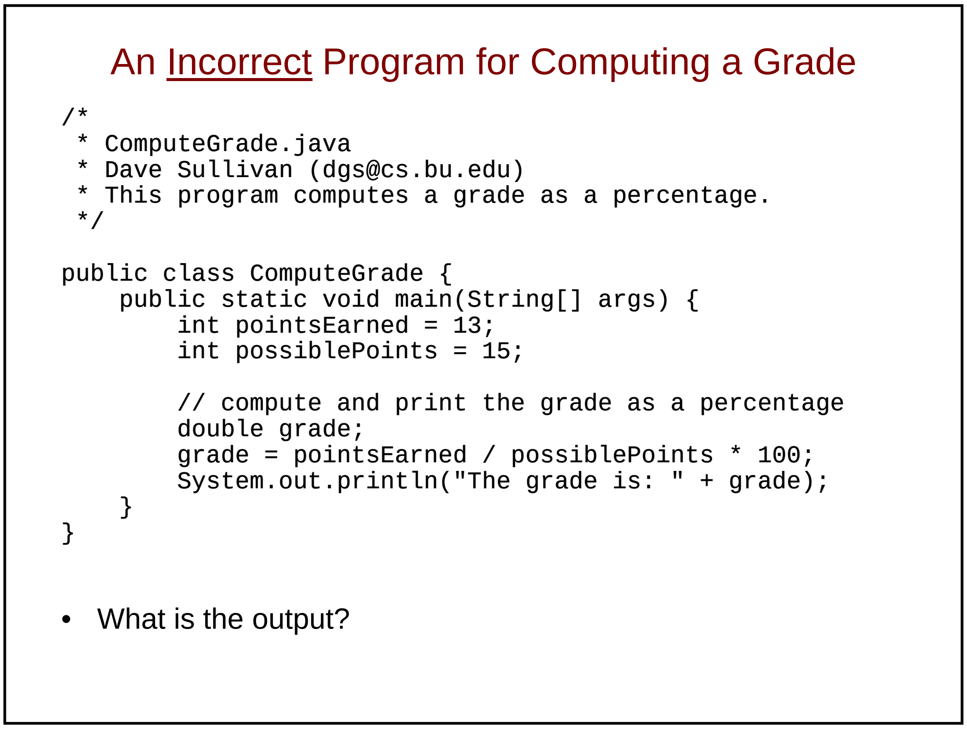
    ```
    System.out.println("total in cents: " + cents);
    ```

- Here's how the evaluation occurs:

```
int cents = 321;
System.out.println("total in cents: " + cents);
                   "total in cents: " + 321
                   "total in cents: " + "321"
                   "total in cents: 321"
```

Change-Adder Using String Concatenation

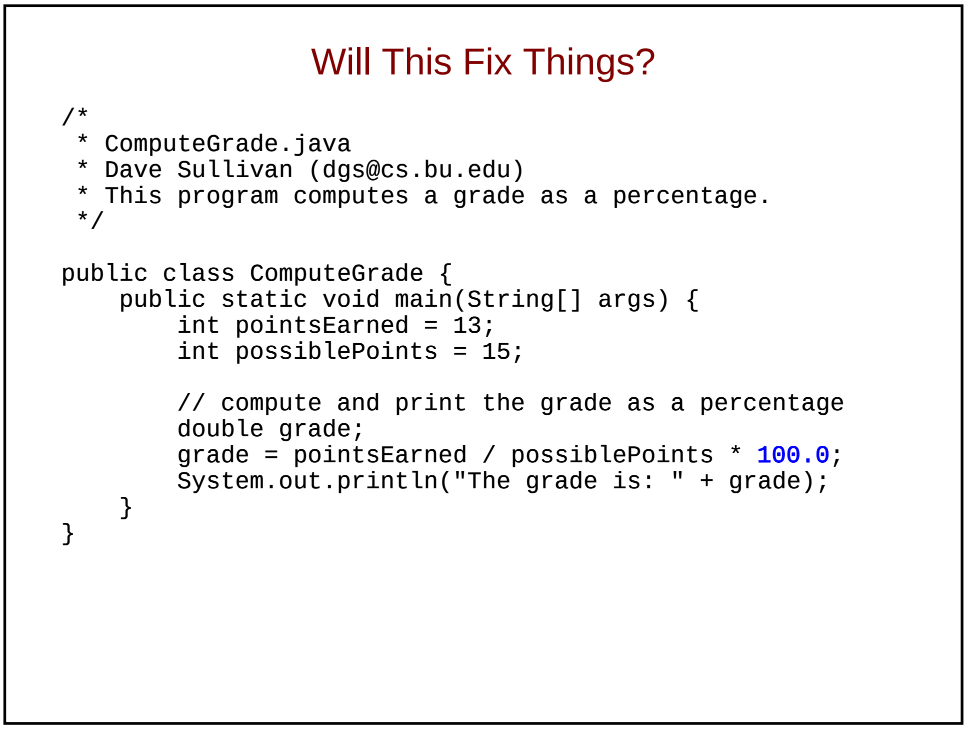
```
/*
 * ChangeAdder2.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program determines the value of some coins.
 */

public class ChangeAdder2 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int cents;

        // compute and print the total value
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;
        System.out.println("total in cents is: " + cents);
        double dollars = cents / 100.0;
        System.out.println("total in dollars is: " +
          dollars);
    }
}
```



An Incorrect Program for Computing a Grade

```
/*
 * ComputeGrade.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = pointsEarned / possiblePoints * 100;
        System.out.println("The grade is: " + grade);
    }
}
```

- What is the output?



Will This Fix Things?

```
/*
 * ComputeGrade.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = pointsEarned / possiblePoints * 100.0;
        System.out.println("The grade is: " + grade);
    }
}
```

Type Casts

- To compute the percentage, we need to tell Java to treat at least one of the operands as a `double`.
- We do so by performing a *type cast:*

```
grade = (double)pointsEarned / possiblePoints * 100;
```

or

```
grade = pointsEarned / (double)possiblePoints * 100;
```

- General syntax for a type cast:

  (<type>)<variable>

Corrected Program for Computing a Grade

```
/*
 * ComputeGrade.java
 * Dave Sullivan (dgs@cs.bu.edu)
 * This program computes a grade as a percentage.
 */

public class ComputeGrade {
    public static void main(String[] args) {
        int pointsEarned = 13;
        int possiblePoints = 15;

        // compute and print the grade as a percentage
        double grade;
        grade = (double)pointsEarned / possiblePoints * 100;
        System.out.println("The grade is: " + grade);
    }
}
```

Evaluating a Type Cast

- Example of evaluating a type cast:

```
pointsEarned = 13;
possiblePoints = 15;
grade = (double)pointsEarned / possiblePoints * 100;
        (double)13 / 15 * 100;
              13.0 / 15 * 100;
        0.8666666666666667 * 100;
        86.66666666666667;
```

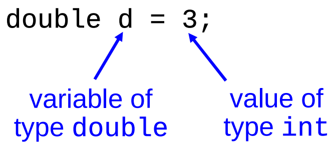
- Note that the type cast occurs *after* the variable is replaced by its value.

- It does *not* change the value that is actually stored in the variable.
  - in the example above, `pointsEarned` is still 13

Type Conversions

- Java will automatically convert values from one type to another *provided there is no potential loss of information.*

- Example: we can perform the following assignment without a type cast:



```
double d = 3;
```

variable of type `double`

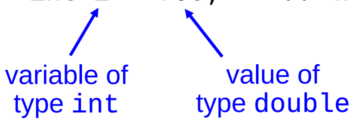
value of type `int`

  - the JVM will convert the integer value `3` to the floating-point value `3.0` and assign that value to `d`
  - *any* `int` can be assigned to a `double` without losing any information

Type Conversions (cont.)

- The compiler will complain if the necessary type conversion could (at least in some cases) lead to a loss of information:

```
int i = 7.5;    // won't compile
```



variable of type `int`

value of type `double`

- This is true regardless of the actual value being converted:

```
int i = 5.0;    // won't compile
```

- To make the compiler happy in such cases, we need to use a type cast:

```
double area = 5.7;
int approximateArea = (int)area;
System.out.println(approximateArea);
```

  - what would the output be?

Type Conversions (cont.)

- When an automatic type conversion is performed as part of an assignment, the conversion happens <u>after</u> the evaluation of the expression to the right of the =.

- Example:

```
double d = 1 / 3;
         = 0;     // uses integer division. why?
         = 0.0;
```

A Block of Code

- A *block* of code is a set of statements that is treated as a single unit.

- In Java, a block is typically surrounded by curly braces.

- Examples:
  - each class is a block
  - each method is a block

```
public class MyProgram {
    public static void main(String[] args) {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }
}
```

Variable Scope

- The *scope* of a variable is the portion of a program in which the variable can be used.

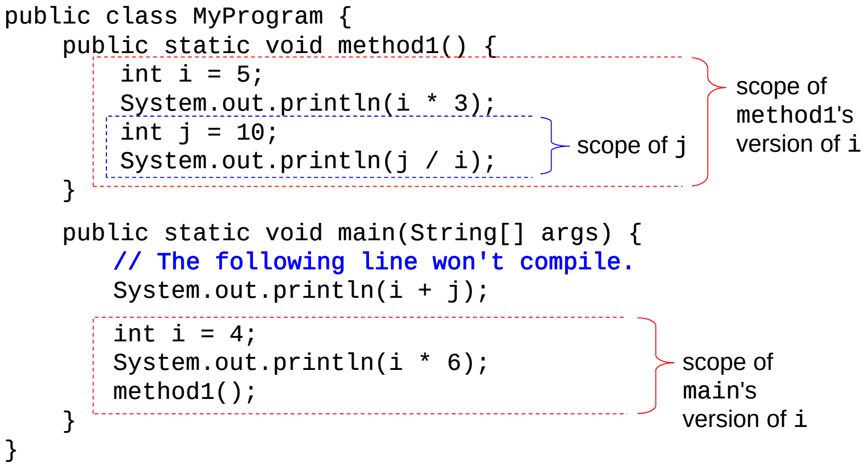
- By default, the scope of a variable:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration

```
public class MyProgram {
    public static void main(String[] args) {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }
}
```

- Because of these rules, a variable cannot be used outside of the block in which it is declared.

Variable Scope (cont.)

- Example:



```
public class MyProgram {
    public static void method1() {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }

    public static void main(String[] args) {
       // The following line won't compile.
       System.out.println(i + j);

       int i = 4;
       System.out.println(i * 6);
       method1();
    }
}
```

scope of j

scope of method1's version of i

scope of main's version of i

Local Variables

- The rules for scope mean that a variable declared inside of a method cannot be used outside of that method.
  - such variables are called *local variables*
  - they "belong to" that method
- In theory, we could use *global variables* that are available throughout the program.
  - declare them outside of any method:

```
public class MyProgram {
    static int x;    // a global variable

    public static void method1() {
        ...
```

- However, we generally avoid global variables.
  - can lead to problems in which one method accidentally affects the behavior of another method

Yet Another Change-Adder Program!

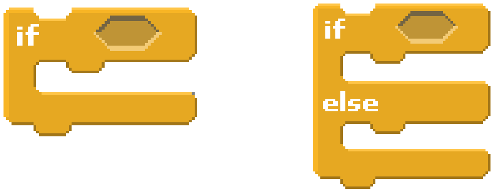
- Let's change it to print the result in dollars and cents.
  - 321 cents should print as 3 dollars, 21 cents

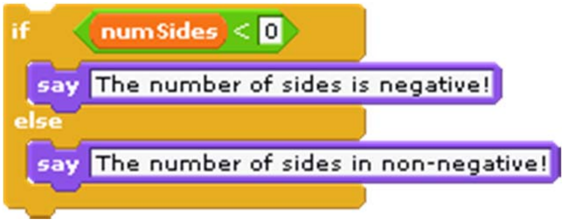
```
public class ChangeAdder3 {
    public static void main(String[] args) {
        int quarters = 10;
        int dimes = 3;
        int nickels = 7;
        int pennies = 6;
        int dollars, cents;

        cents = 25*quarters + 10*dimes + 5*nickels + pennies;

        // what should go here?



        System.out.println("dollars = " + dollars);
        System.out.println("cents = " + cents);
    }
}
```

The Need for Conditional Execution

- What if the user has 121 cents?
  - will print as 1 dollar**s**, 21 cents
  - would like it to print as 1 dollar, 21 cents
- We need a means of choosing what to print at runtime.

Recall: Conditional Execution in Scratch





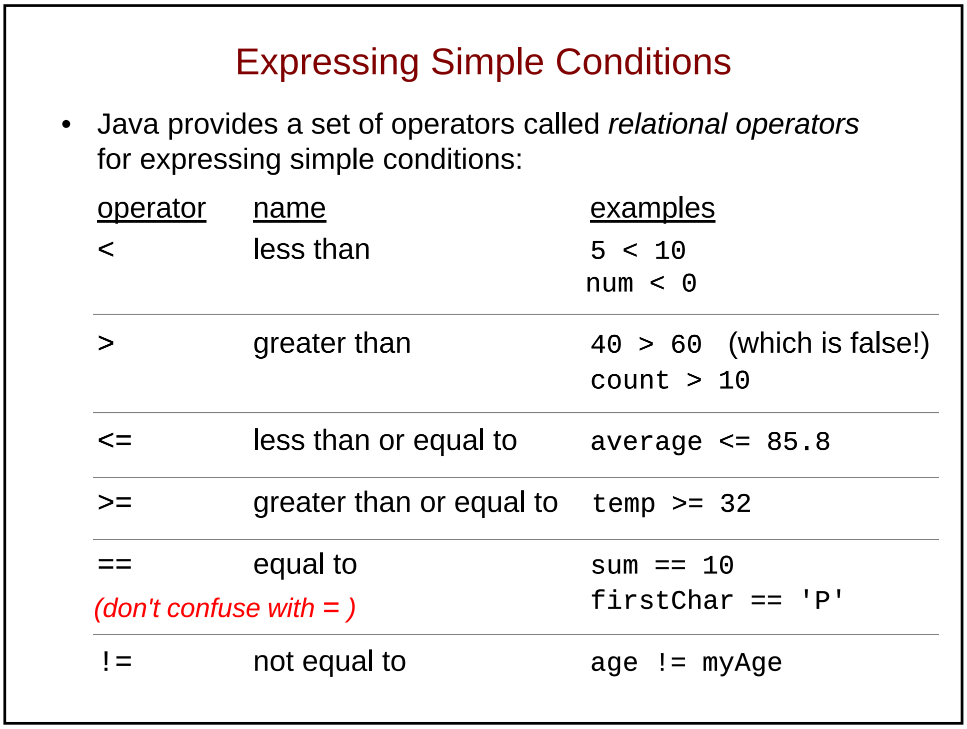
```
if numSides < 0
    say The number of sides is negative!
else
    say The number of sides in non-negative!
```

Conditional Execution in Java

```
if (<condition>) {
    <true block>
} else {
    <false block>
}
```

```
if (<condition>) {
    <true block>
}
```

- If the condition is true:
  - the statement(s) in the true block are executed
  - the statement(s) in the false block (if any) are skipped
- If the condition is false:
  - the statement(s) in the false block (if any) are executed
  - the statement(s) in the true block are skipped



Expressing Simple Conditions

- Java provides a set of operators called *relational operators* for expressing simple conditions:

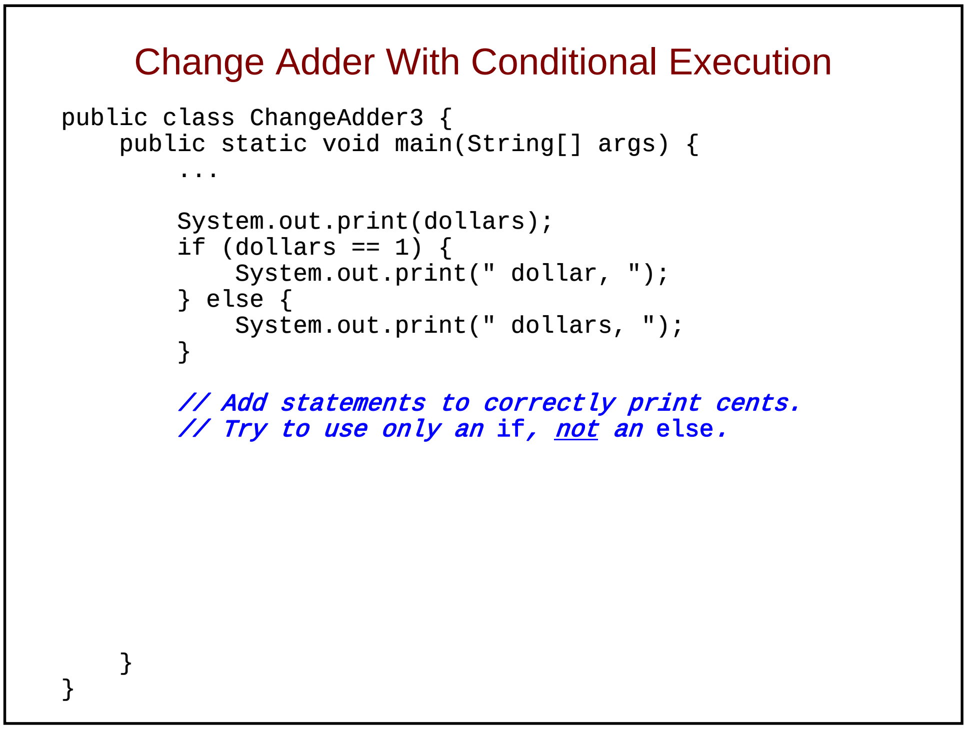
operator	name	examples

`<`	less than	`5 < 10`
		`num < 0`

`>`	greater than	`40 > 60` (which is false!) `count > 10`
`<=`	less than or equal to	`average <= 85.8`
`>=`	greater than or equal to	`temp >= 32`

`==`
(don't confuse with =)
	equal to	`sum == 10`
		`firstChar == 'P'`

`!=`	not equal to	`age != myAge`



Change Adder With Conditional Execution

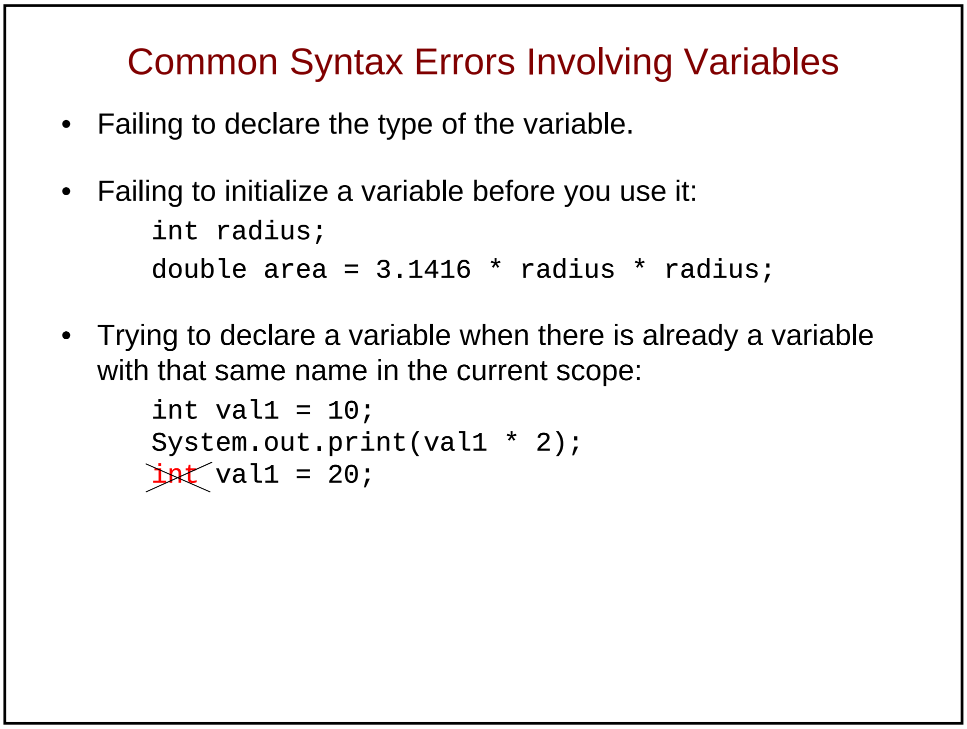
```
public class ChangeAdder3 {
    public static void main(String[] args) {
        ...

        System.out.print(dollars);
        if (dollars == 1) {
            System.out.print(" dollar, ");
        } else {
            System.out.print(" dollars, ");
        }

        // Add statements to correctly print cents.
        // Try to use only an if, not an else.






    }
}
```

Classifying Bugs

- Syntax errors
  - found by the compiler
  - occur when code doesn't follow the rules of the programming language
  - examples?

Classifying Bugs

- Syntax errors
  - found by the compiler
  - occur when code doesn't follow the rules of the programming language
  - examples?
- Logic errors
  - the code compiles, but it doesn’t do what you intended it to do
  - may or may not cause the program to crash
    - called *runtime errors* if the program crashes
  - often harder to find!



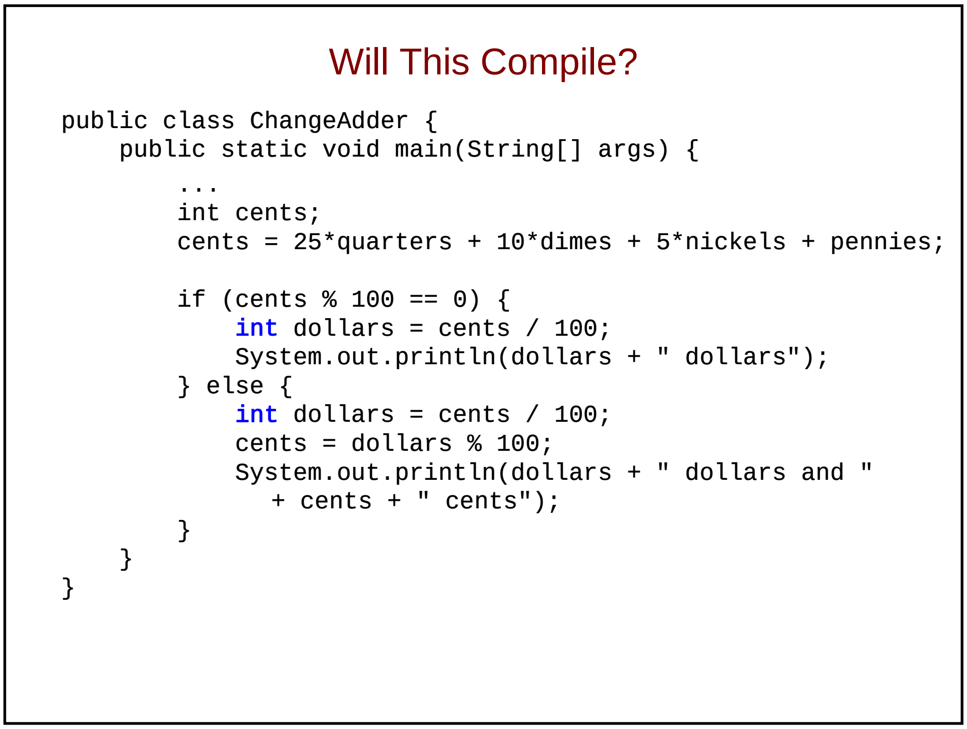
Common Syntax Errors Involving Variables

- Failing to declare the type of the variable.
- Failing to initialize a variable before you use it:

```
int radius;
double area = 3.1416 * radius * radius;
```

- Trying to declare a variable when there is already a variable with that same name in the current scope:

```
int val1 = 10;
System.out.print(val1 * 2);
~~int~~ val1 = 20;
```



Will This Compile?

```
public class ChangeAdder {
    public static void main(String[] args) {
        ...
        int cents;
        cents = 25*quarters + 10*dimes + 5*nickels + pennies;

        if (cents % 100 == 0) {
            int dollars = cents / 100;
            System.out.println(dollars + " dollars");
        } else {
            int dollars = cents / 100;
            cents = dollars % 100;
            System.out.println(dollars + " dollars and "
              + cents + " cents");
        }
    }
}
```

Representing Integers

- Like all values in a computer, integers are stored as binary numbers – sequences of *bits* (0s and 1s).
- With n bits, we can represent 2^n different values.
  - examples:
    - 2 bits give 2^2 = 4 different values

      00, 01, 10, 11
    - 3 bits give 2^3 = 8 different values

      000, 001, 010, 011, 100, 101, 110, 111
- When we allow for negative integers (which Java does) n bits can represent any integer from -2^{n-1} to $2^{n-1} - 1$.
  - there's one fewer positive value to make room for 0

Java’s Integer Types

- Java’s actually has four primitive types for integers, all of which represent signed integers.

type	# of bits	range of values
`byte`	8	-2^7 to $2^7 - 1$ (–128 to 127)
`short`	16	-2^{15} to $2^{15} - 1$ (–32768 to 32767)
`int`	32	-2^{31} to $2^{31} - 1$ (approx. +/–2 billion)
`long`	64	-2^{63} to $2^{63} - 1$

- We typically use `int`, unless there’s a good reason not to.

Java's Floating-Point Types

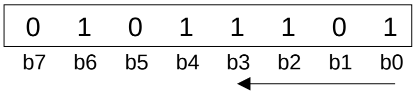
- Java has two primitive types for floating-point numbers:

type	# of bits	approx. range	approx. precision
`float`	32	+/–10^{-45} to +/–10^{38}	7 decimal digits
`double`	64	+/–10^{-324} to +/–10^{308}	15 decimal digits

- We typically use `double` because of its greater precision.

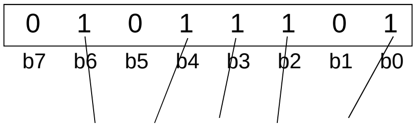
Binary to Decimal

- Number the bits from right to left
  - example:



0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

- For each bit that is 1, add 2^n, where n = the bit number
  - example:



0	1	0	1	1	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

decimal value = $2^6 + 2^4 + 2^3 + 2^2 + 2^0$

$64 + 16 + 8 + 4 + 1 = 93$

  - another example: what is the integer represented by 01001011?

Decimal to Binary

- Go in the reverse direction: determine which powers of 2 need to be added together to produce the decimal number.
  - example: $42 = 32 + 8 + 2$
    $= 2^5 + 2^3 + 2^1$
  - thus, bits 5, 3, and 1 are all 1s: 42 = 00101010

- Start with the largest power of 2 less than or equal to the number, and work down from there.
  - example: what is 21 in binary?

    16 is the largest power of 2 <= 21: $21 = 16 + 5$

    now, break the 5 into powers of 2: $21 = 16 + 4 + 1$

    1 is a power of 2 (2^0), so we're done: $21 = 16 + 4 + 1$
    $= 2^4 + 2^2 + 2^0$
    $= 00010101$

Decimal to Binary (cont.)

- Another example: what is 90 in binary?

printf: Formatted Output

- When printing a decimal number, you may want to limit yourself to a certain number of places after the decimal.
- You can do so using the `System.out.printf` method.
  - example:

    ```
    System.out.printf("%.2f", 1.0/3);
    ```

    will print

    ```
    0.33
    ```

  - the number after the decimal point in the first parameter indicates how many places after the decimal should be used
- There are other types of formatting that can also be performed using this method.
  - see Table 4.6 in the textbook for more examples

Review

- Consider the following code fragments

  1) `1000`
  2) `10 * 5`
  3) `System.out.println("Hello");`
  4) `hello`
  5) `num1 = 5;`
  6) `2*width + 2*length`
  7) `main`

- Which of them are examples of:
  - literals?
  - expressions?
  - identifiers?
  - statements?

Unit 2, Part 2

Definite Loops

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Using a Variable for Counting

- Let's say that we're using a variable `i` to count the number of times that something has been done:

```
int i = 0;                 i [    0 ]
```

- To increase the count, we can do this:

```
i = i + 1;
    0 + 1
      1                    i [    1 ]
```

- To increase the count again, we repeat the same assignment:

```
i = i + 1;
    1 + 1
      2                    i [    2 ]
```

Increment and Decrement Operators

- Instead of writing

```
i = i + 1;
```

we can use a shortcut and just write

```
i++;
```

- `++` is known as the *increment operator*.
  - increment = increase by 1
- Java also provides a *decrement operator* (`--`).
  - decrement = decrease by 1
  - example:

```
i--;
```

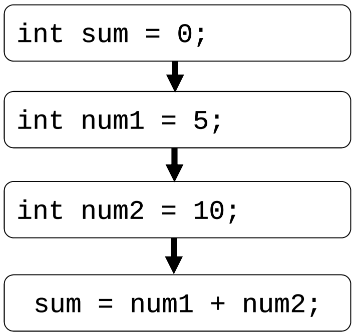
Review: Flow of Control

- Flow of control = the order in which instructions are executed
- By default, instructions are executed in sequential order.

<u>instructions</u>

```
int sum = 0;
int num1 = 5;
int num2 = 10;
sum = num1 + num2;
```

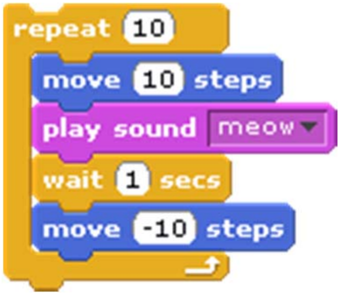
<u>flowchart</u>



- When we make a method call, the flow of control "jumps" to the method, and it "jumps" back when the method completes.

Altering the Flow of Control: Repetition

- To solve many types of problems, we need to be able to modify the order in which instructions are executed.
- One reason for doing this is to allow for *repetition.*
- We saw this in Scratch:



```
repeat 10
    move 10 steps
    play sound meow
    wait 1 secs
    move -10 steps
```

Example of the Need for Repetition

- Here's a method for writing a large block letter L:

```
public static void writeL() {
    System.out.println("|");
    System.out.println("|");
    System.out.println("|");
    System.out.println("|");
    System.out.println("|");
    System.out.println("|");
    System.out.println("|");
    System.out.println("+----------");
}
```

- Rather than duplicating the statement

  `System.out.println("|");`

  seven times, we'd like to have this statement appear just once and execute it seven times.

for Loops

- To repeat one or more statements multiple times, we can use a construct known as a *for loop.*
- Here's a revised version of our `writeL` method that uses one:

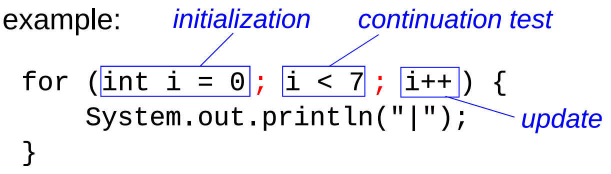
```
public static void writeL() {
    for (int i = 0; i < 7; i++) {
        System.out.println("|");
    }
    System.out.println("+----------");
}
```

for Loops

- Syntax:

```
for (<initialization>; <continuation test>; <update>) {
    <one or more statements>
}
```

- In our example:

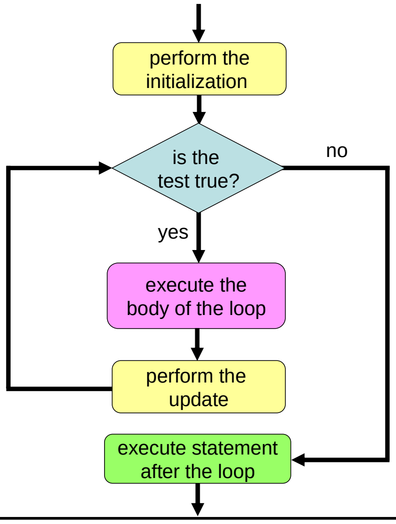


```
for (int i = 0; i < 7; i++) {
    System.out.println("|");
}
```

- The statements inside the loop are known as the *body* of the loop.
- In our example, we use the variable `i` to count the number of times that the body has been executed.

Executing a `for` Loop

```
for (<initialization>; <continuation test>; <update>) {
    <body of the loop>
}
```

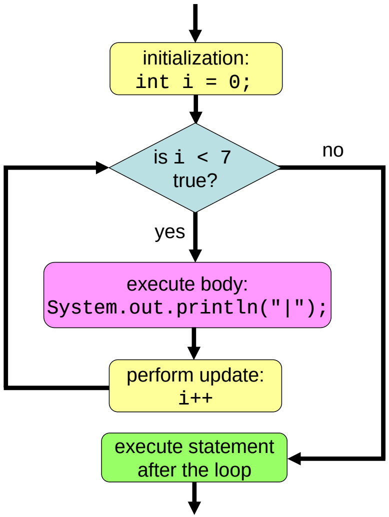

perform the initialization
is the test true?
no
yes
execute the body of the loop
perform the update
execute statement after the loop

Notes:

- the initialization is only performed once
- the body is only executed if the test is true
- we repeatedly do:
  test
  body
  update
  until the test is false

Executing Our `for` Loop

```
for (int i = 0; i < 7; i++) {
    System.out.println("|");
}
```



i	i < 7	action
0	true	print 1st "|"
1	true	print 2nd "|"
2	true	print 3rd "|"
3	true	print 4th "|"
4	true	print 5th "|"
5	true	print 6th "|"
6	true	print 7th "|"
7	false	execute stmt. after the loop

Definite Loops

- For now, we'll limit ourselves to *definite loops* – which repeat actions a fixed number of times.

- To repeat the body of a loop **<N>** times, we typically take one of the following approaches:

```
for (int i = 0; i < <N>; i++) {
    <body of the loop>
}
```

OR

```
for (int i = 1; i <= <N>; i++) {
    <body of the loop>
}
```

- Each time that the body of a loop is executed is known as an *iteration* of the loop.
  - the loops shown above perform <N> iterations

Other Examples of Definite Loops

- What does this loop do?

```
for (int i = 0; i < 3; i++) {
    System.out.println("Hip! Hip!");
    System.out.println("Hooray!");
}
```

- What does this loop do?

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Using Different Initializations, Tests, and Updates

- The second loop from the previous page would be clearer if we expressed it like this:

```
for (int i = 0; i <= 9; i++) {
    System.out.println(i);
}
```

- Different problems may require different initializations, continuation tests, and updates.

- What does this code fragment do?

```
for (int i = 2; i <= 10; i = i + 2) {
    System.out.println(i * 10);
}
```

Tracing a `for` Loop

- Let's trace through the final code fragment from the last slide:

```
for (int i = 2; i <= 10; i = i + 2) {
    System.out.println(i * 10);
}
```

`i`	`i <= 10`	value printed

Common Mistake

- You should not put a semi-colon after the for-loop header:

```
for (int i = 0; i < 7; i++); {
    System.out.println("|");
}
```

- The semi-colon ends the `for` statement.
  - thus, it doesn't repeat anything!
- The println is independent of the `for` statement, and only executes once.

Practice

- Fill in the blanks below to print the integers from 1 to 10:

```
for (____________; ____________; ____________) {
    System.out.println(i);
}
```

- Fill in the blanks below to print the integers from 10 to 20:

```
for (____________; ____________; ____________) {
    System.out.println(i);
}
```

- Fill in the blanks below to print the integers from 10 down to 1:

```
for (____________; ____________; ____________) {
    System.out.println(i);
}
```

Other Java Shortcuts

- Recall this code fragment:

```
for (int i = 2; i <= 10; i = i + 2) {
    System.out.println(i * 10);
}
```

- Instead of writing

```
i = i + 2;
```

  we can use a shortcut and just write

```
i += 2;
```

- In general

  <variable> += <expression>;

  is equivalent to

  <variable> = <variable> + (<expression>);

Java Shortcuts

- Java offers other shortcut operators as well.

- Here's a summary of all of them:

shortcut	equivalent to
<var>++;	<var> = <var> + 1;
<var>--;	<var> = <var> – 1;
<var> += <expr>;	<var> = <var> + (<expr>);
<var> -= <expr>;	<var> = <var> – (<expr>);
<var> *= <expr>;	<var> = <var> * (<expr>);
<var> /= <expr>;	<var> = <var> / (<expr>);
<var> %= <expr>;	<var> = <var> % (<expr>);

- Important: the = must come *after* the mathematical operator.

  += is correct

  =+ is not!

More Practice

- Fill in the blanks below to print the even integers in reverse order from 20 down to 6:

```
for (____________; ____________; ____________) {
    System.out.println(i);
}
```

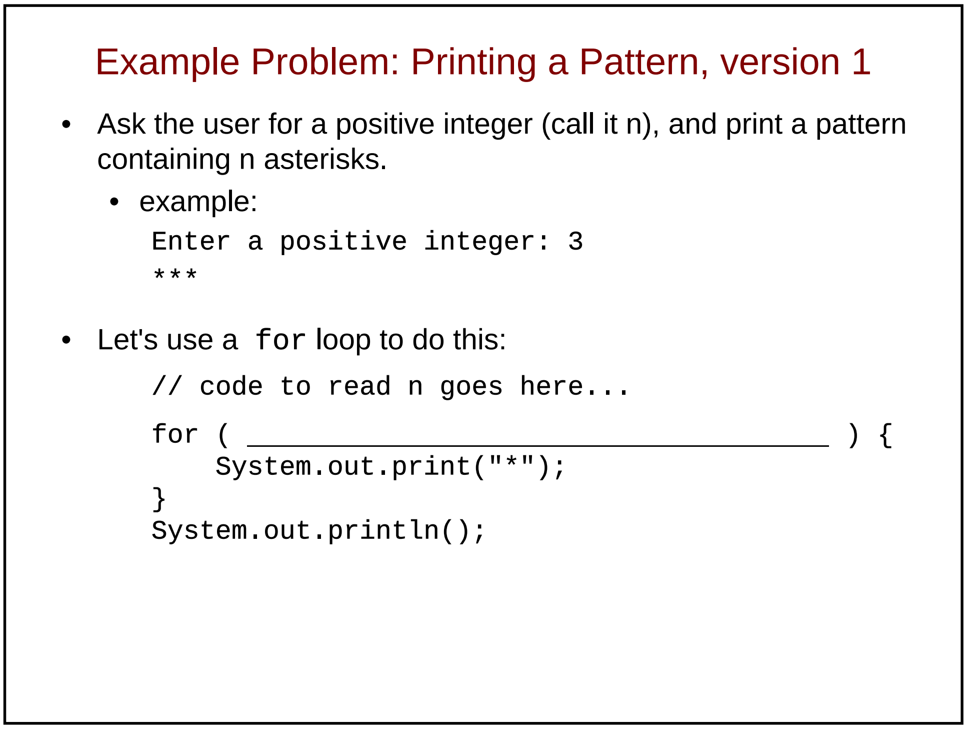
Find the Error

- Let's say that we want to print the numbers from 1 to n.

- Where is the error in the following code?

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

- This is an example of an *off-by-one error*. Beware of these when writing your loop conditions!



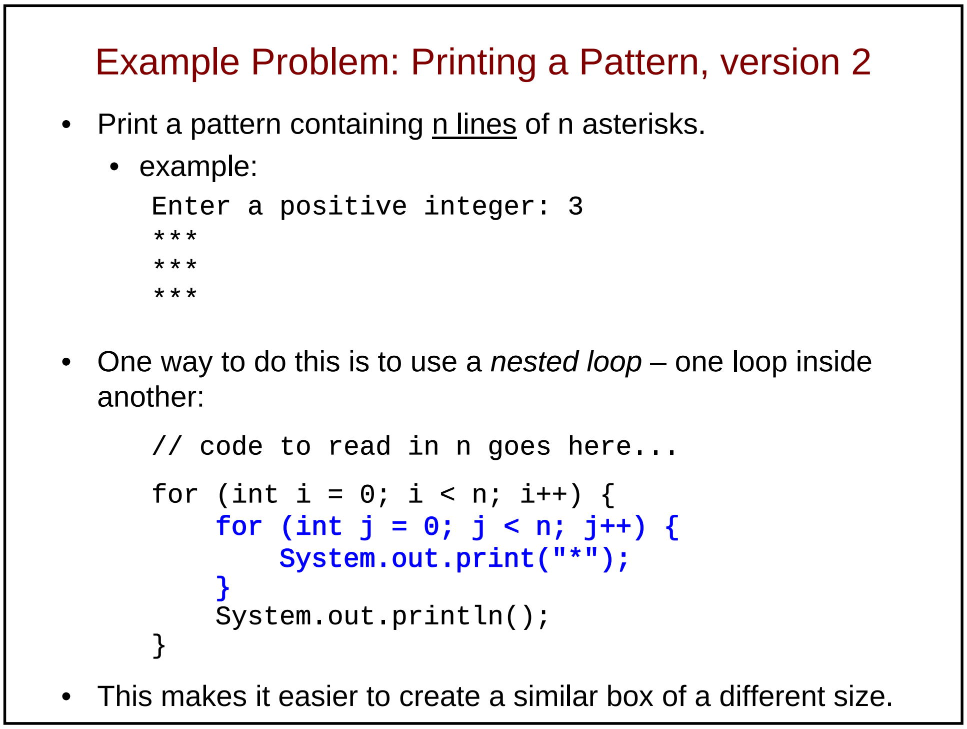
Example Problem: Printing a Pattern, version 1

- Ask the user for a positive integer (call it n), and print a pattern containing n asterisks.
  - example:

```
Enter a positive integer: 3
***
```

- Let's use a `for` loop to do this:

```
// code to read n goes here...

for ( ____________________________________ ) {
    System.out.print("*");
}
System.out.println();
```



Example Problem: Printing a Pattern, version 2

- Print a pattern containing n lines of n asterisks.
  - example:

```
Enter a positive integer: 3
***
***
***
```

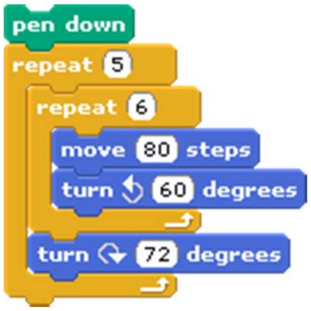
- One way to do this is to use a *nested loop* – one loop inside another:

```
// code to read in n goes here...

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

- This makes it easier to create a similar box of a different size.

Nested Loops

- When you have a nested loop, the inner loop is executed to completion for every iteration of the outer loop.
- Recall our Scratch drawing program:



```
pen down
repeat 5
    repeat 6
        move 80 steps
        turn ↺ 60 degrees
    turn ↻ 72 degrees
```

- How many times is the *move* statement executed?

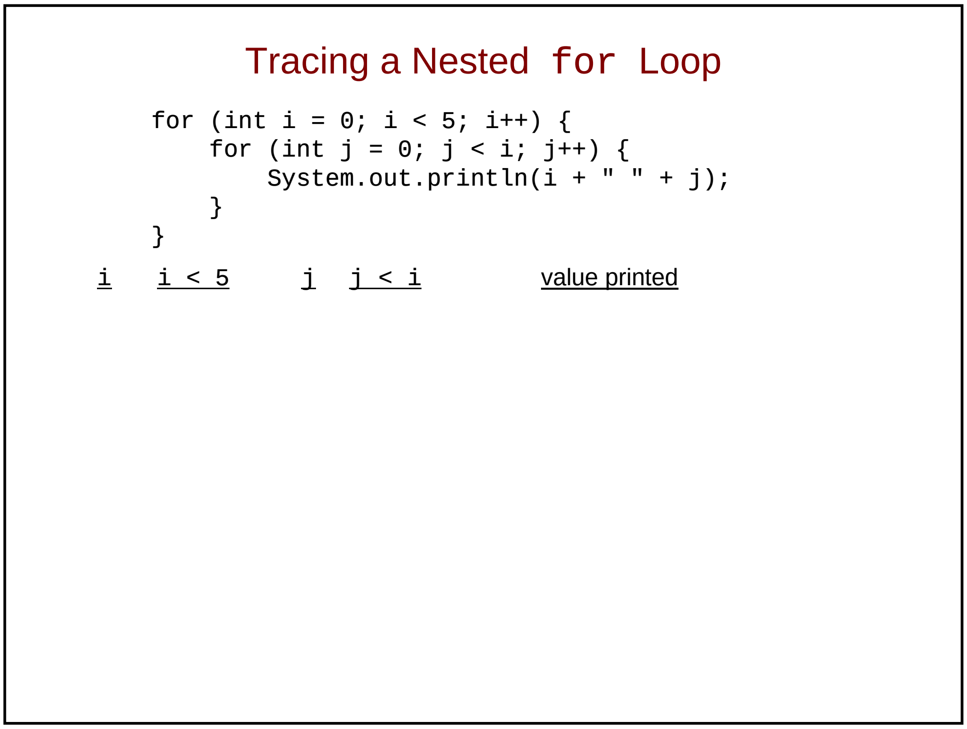
Nested Loops (cont.)

- How many times is the `println` statement executed below?

```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 7; j++) {
        System.out.println(i + " " + j);
    }
}
```

- How many times is the `println` statement executed below?

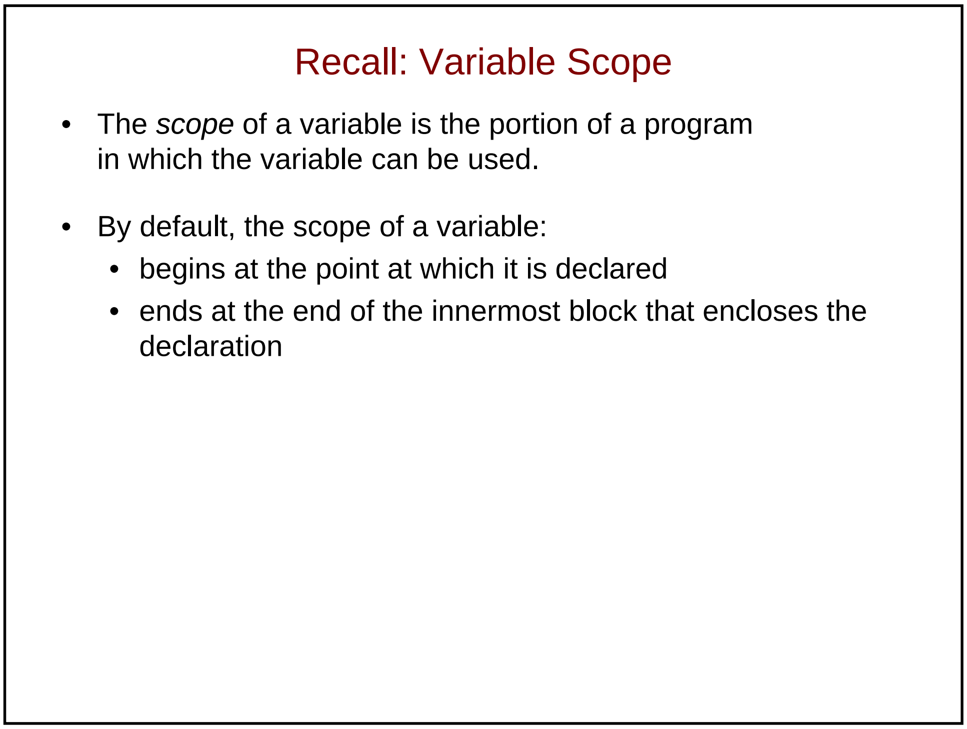
```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println(i + " " + j);
    }
}
```



Tracing a Nested `for` Loop

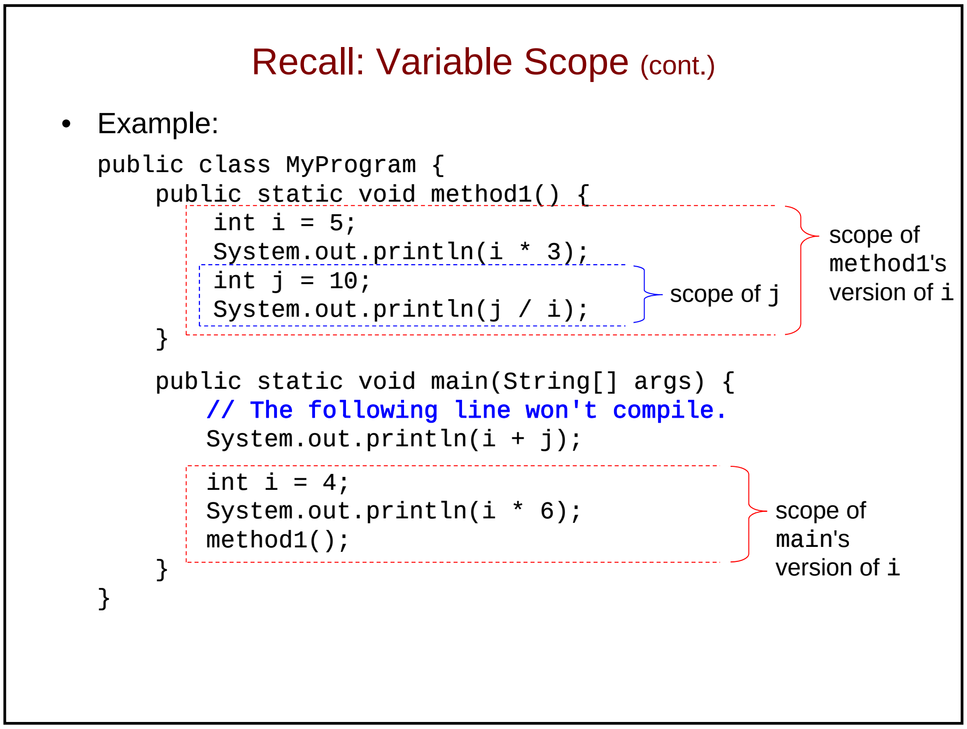
```
for (int i = 0; i < 5; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println(i + " " + j);
    }
}
```

i	i < 5	j	j < i	value printed



Recall: Variable Scope

- The *scope* of a variable is the portion of a program in which the variable can be used.
- By default, the scope of a variable:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration



Recall: Variable Scope (cont.)

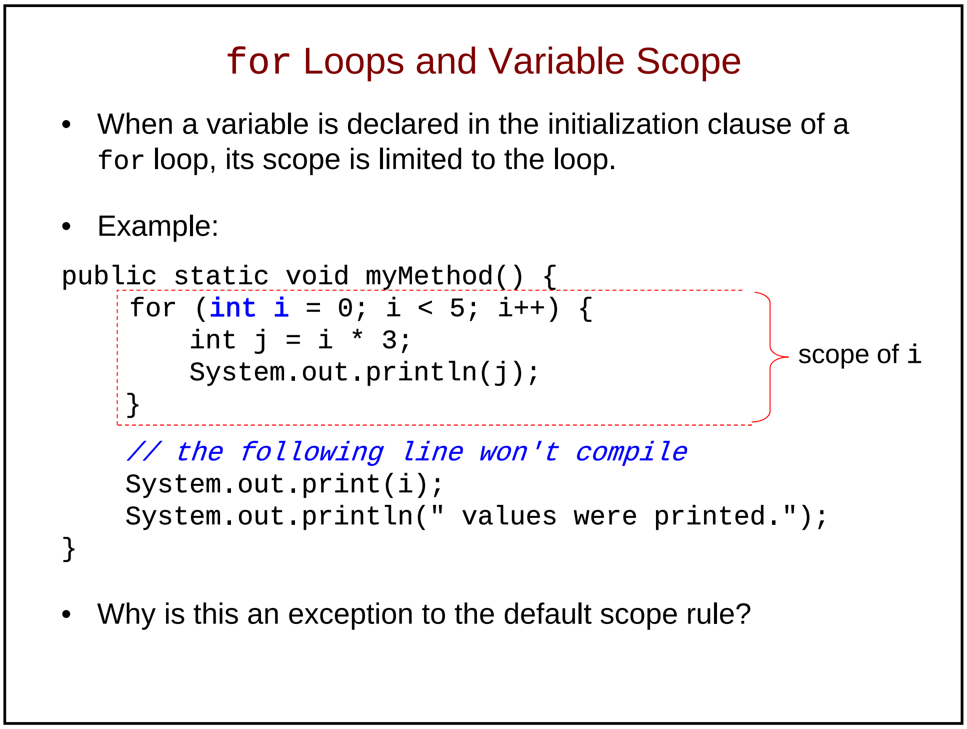
- Example:

```
public class MyProgram {
    public static void method1() {
        int i = 5;
        System.out.println(i * 3);
        int j = 10;
        System.out.println(j / i);
    }

    public static void main(String[] args) {
        // The following line won't compile.
        System.out.println(i + j);

        int i = 4;
        System.out.println(i * 6);
        method1();
    }
}
```

scope of j

scope of method1's version of i

scope of main's version of i



for Loops and Variable Scope

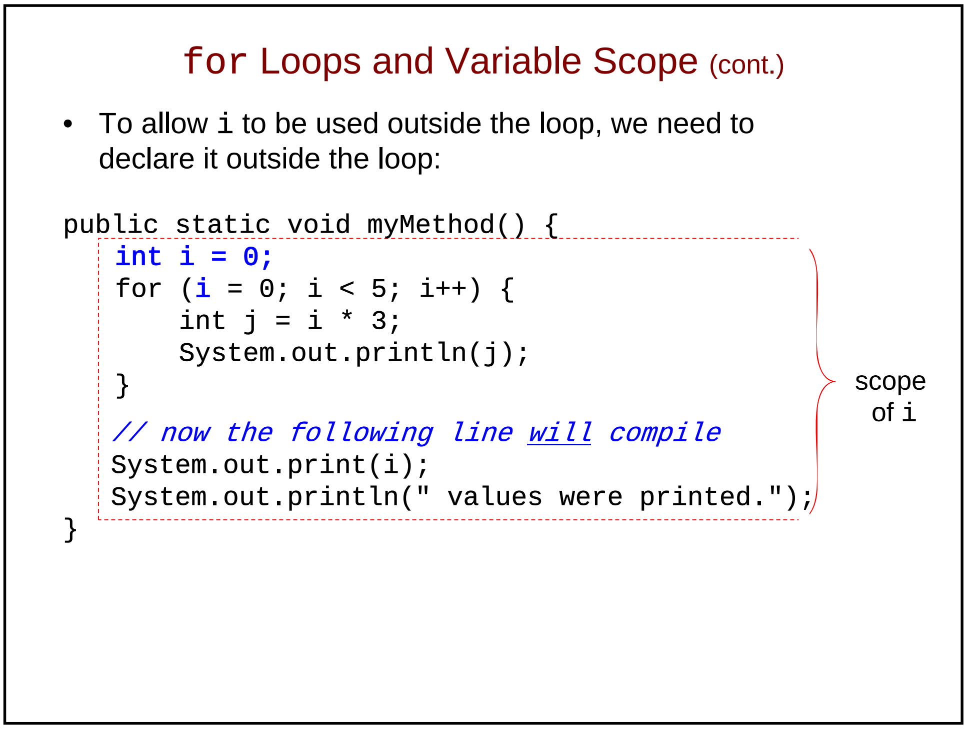
- When a variable is declared in the initialization clause of a for loop, its scope is limited to the loop.

- Example:

```
public static void myMethod() {
    for (int i = 0; i < 5; i++) {
        int j = i * 3;
        System.out.println(j);
    }
    // the following line won't compile
    System.out.print(i);
    System.out.println(" values were printed.");
}
```

scope of i

- Why is this an exception to the default scope rule?

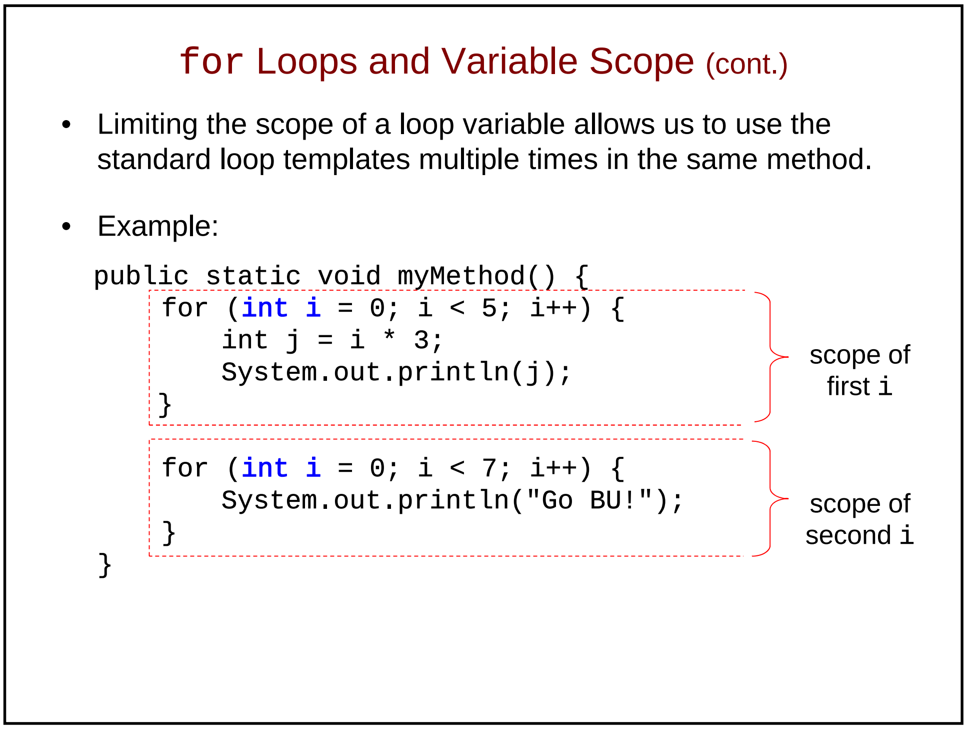


for Loops and Variable Scope (cont.)

- To allow i to be used outside the loop, we need to declare it outside the loop:

```
public static void myMethod() {
    int i = 0;
    for (i = 0; i < 5; i++) {
        int j = i * 3;
        System.out.println(j);
    }
   // now the following line will compile
   System.out.print(i);
   System.out.println(" values were printed.");
}
```

scope of i



for Loops and Variable Scope (cont.)

- Limiting the scope of a loop variable allows us to use the standard loop templates multiple times in the same method.

- Example:

```
public static void myMethod() {
    for (int i = 0; i < 5; i++) {
        int j = i * 3;
        System.out.println(j);
    }

    for (int i = 0; i < 7; i++) {
        System.out.println("Go BU!");
    }
}
```

scope of first i

scope of second i

Review: Simple Repetition Loops

- Recall our two templates for performing <N> repetitions:

```
for (int i = 0; i < <N>; i++) {
    // code to be repeated
}

for (int i = 1; i <= <N>; i++) {
    // code to be repeated
}
```

- How may repetitions will each of the following perform?

```
for (int i = 1; i <= 15; i++) {
    System.out.println("Hello");
    System.out.println("How are you?");
}

for (int i = 0; i < 2*j; i++) {
    …
}
```

More Practice: Tracing a Nested `for` Loop

```
for (int i = 1; i <= 3; i++) {
    for (int j = 0; j < 2*i + 1; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

i	i <= 3	j	j < 2*i + 1

output

Case Study: Drawing a Complex Figure

- Here's the figure:

```
   ()
  (())
 ((()))
(((())))
========
|::::::|
 |::::|
  |::|
  |::|
  |::|
  |::|
  +==+
```

- To begin with, we'll focus on creating this exact figure.

- Then we'll modify our code so that the size of the figure can easily be changed.
  - we'll use `for` loops to allow for this

Problem Decomposition

- We begin by breaking the problem into subproblems, looking for groups of lines that follow the same pattern:

```
   ()
  (())       ← flame
 ((()))
(((())))

========     ← rim of torch

|::::::|
 |::::|      ← top of torch

  |::|
  |::|
  |::|       ← handle of torch
  |::|

  +--+       ← bottom of torch
```

Problem Decomposition (cont.)

- This gives us the following initial pseudocode:

```
   ()
  (())
 ((()))
(((())))

========

|::::::|
 |::::|

  |::|
  |::|
  |::|
  |::|

  +--+
```

```
draw the flame
draw the rim of the torch
draw the top of the torch
draw the handle of the torch
draw the bottom of the torch
```

- This is a high-level description of what needs to be done.
- We'll gradually expand the pseudocode into more and more detailed instructions – until we're able to implement them in Java.

Drawing the Flame

- Let's begin by refining our specification for drawing the flame.

```
   ()
  (())
 ((()))
(((())))
```

- Here's our initial pseudocode for this task:

```
for (each of 4 lines) {
    print some spaces (possibly 0)
    print some left parentheses
    print some right parentheses
    go to a new line
}
```

- We need formulas for how many spaces and parens should be printed on a given line.

Finding the Formulas

```
1   ()
2  (())
3 ((()))
4(((())))
```

- To begin with, we:
  - number the lines in the flame
  - form a table of the number of spaces and parentheses on each line:

line	spaces	parens (each type)
1	3	1
2	2	2
3	1	3
4	0	4

- Then we find the formulas.
  - assume the formulas are *linear functions* of the line number:

    ```
    c1*line + c2
    ```

    where c1 and c2 are constants
  - parens = ?
  - spaces = ?

Refining the Pseudocode

- Given these formulas, we can refine our pseudocode:

```
for (each of 4 lines) {
    print some spaces (possibly 0)
    print some left parentheses
    print some right parentheses
    go to a new line
}
```

```
for (line going from 1 to 4) {
    print 4 – line spaces
    print line left parentheses
    print line right parentheses
    go to a new line
}
```

Implementing the Pseudocode in Java

- We use nested `for` loops:

```
for (line going from 1 to 4) {
    print 4 – line spaces
    print line left parentheses
    print line right parentheses
    go to a new line
}
```

```
for (int line = 1; line <= 4; line++) {
    for (int i = 0; i < 4 – line; i++) {
        System.out.print(" ");
    }
    for (int i = 0; i < line; i++) {
        System.out.print("(");
    }
    for (int i = 0; i < line; i++) {
        System.out.print(")");
    }
    System.out.println();
}
```

A Method for Drawing the Flame

- We put the code in its own static method, and add some explanatory comments:

```
public static void drawFlame() {
    for (int line = 1; line <= 4; line++) {
        // spaces to the left of the current line
        for (int i = 0; i < 4 – line; i++) {
            System.out.print(" ");
        }

        // left and right parens on the current line
        for (int i = 0; i < line; i++) {
            System.out.print("(");
        }
        for (int i = 0; i < line; i++) {
            System.out.print(")");
        }

        System.out.println();
    }
}
```

Drawing the Top of the Torch

```
1|::::::|
2 |::::|
```

- What's the initial pseudocode for this task?

```
for (each of 2 lines) {





}
```

- Here's a table for the number of spaces and number of colons:

line	spaces	colons
1	0	6
2	1	4

  - spaces = ?
  - colons decreases by 2 as line increases by 1
    ➔ colons = -2*line + c2 for some number c2
  - try different values, and eventually get: colons = ?

Refining the Pseudocode

- Once again, we use the formulas to refine our pseudocode:

```
for (each of 2 lines) {
    print some spaces (possibly 0)
    print a single vertical bar
    print some colons
    print a single vertical bar
    go to a new line
}
```

⇩

```
for (line going from 1 to 2) {
    print line - 1 spaces
    print a single vertical bar
    print -2*line + 8 colons
    print a single vertical bar
    go to a new line
}
```

A Method for Drawing the Top of the Torch

```
public static void drawTop() {
    for (int line = 1; line <= 2; line++) {
        // spaces to the left of the current line
        for (int i = 0; i < line - 1; i++) {
            System.out.print(" ");
        }

        // bars and colons on the current line
        System.out.print("|");
        for (int i = 0; i < -2*line + 8; i++) {
            System.out.print(":");
        }
        System.out.print("|");

        System.out.println();
    }
}
```

Drawing the Rim

- This always has only one line, so we *don't* need *nested* loops.

```
========
```

- However, we still need a single loop, because we want to be able to scale the size of the figure.
- What should the code look like?

```
for (              ;              ;              ) {


}
```

- This code also goes in its own method, called `drawRim()`

Incremental Development

- We take similar steps to implement methods for the remaining subtasks.
- After completing a given method, we test and debug it.
- The `main` method just calls the methods for the subtasks:

```
public static void main(String[] args) {
    drawFlame();
    drawRim();
    drawTop();
    drawHandle();
    drawBottom();
}
```

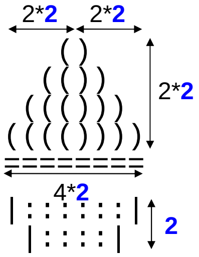
- See the example program `DrawTorch.java`

Using Class Constants

- To make the torch larger or smaller, we'd need to make many changes.
  - the size of the figure is hard-coded into most methods
- To make the program more flexible, we can store info. about the figure's dimensions in one or more *class constants*.
  - like variables, but their values are fixed
  - can be used throughout the program

Using Class Constants (cont.)

- We only need one constant for the torch.
  - for the default size, it equals 2
  - its connection to some of the dimensions is shown at right



- We declare it at the very start of the class:

```
public class DrawTorch2 {
    public static final int SCALE_FACTOR = 2;
    ...
```

- General syntax:

  `public static final` <type> <name> = <expression>`;`

  - conventions:
    - capitalize all letters in the name
    - put an underscore ('_') between multiple words

Scaling the Figure

- Here are some other versions of the figure:

```
     ()
    (())
   ((()))
  (((())))
 ((((()))))
(((((())))))
============
|::::::::::|
 |::::::::|
  |::::::|
   |::::|
   |::::|
   |::::|
   |::::|
   |::::|
   |::::|
   +====+
```

SCALE_FACTOR = 3

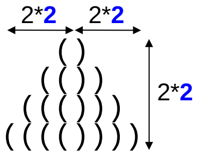
```
 ()
(())
====
|::|
 ||
 ||
 ++
```

SCALE_FACTOR = 1

Revised Method for Drawing the Flame

- We replace the two 4s with `2*SCALE_FACTOR`:

```
public static void drawFlame() {
    for (int line = 1; line <= 2*SCALE_FACTOR; line++) {
        // spaces to the left of the flame
        for (int i = 0; i < 2*SCALE_FACTOR – line; i++) {
            System.out.print(" ");
        }

        // the flame itself, both left and right halves
        for (int i = 0; i < line; i++) {
            System.out.print("(");
        }
        for (int i = 0; i < line; i++) {
            System.out.print(")");
        }

        System.out.println();
    }
}
```



```
   ()
  (())
 ((()))
(((())))
```

Making the Rim Scaleable

- How does the width of the rim depend on SCALE_FACTOR?

```
     ()                  ()            ()
    (())                (())          (())
   ((()))              ((()))         ====
  (((())))            (((())))
 ((((()))))           ========
(((((())))))
============
```

- Use a table!

SCALE_FACTOR	width of rim
1	4
2	8
3	12

width of rim = ?

Revised Method for Drawing the Rim

- Original version (for the default size):

```
public static void drawRim() {
    for (int i = 0; i < 8; i++) {
        System.out.print("=");
    }
    System.out.println();
}
```

- Scaleable version:

```
public static void drawRim() {
    for (int i = 0; i < 4*SCALE_FACTOR; i++) {
        System.out.print("=");
    }
    System.out.println();
}
```

Making the Top of the Torch Scaleable

- For SCALE_FACTOR = 2, we got:

```
1|::::::|
2 |::::|
```

number of lines = 2
spaces = line – 1
colons = -2 * line + 8

- What about SCALE_FACTOR = 3?

```
1|::::::::::|
2 |::::::::|
3  |::::::|
```

line	spaces	colons
1	0	10
2	1	8
3	2	6

number of lines = 3
spaces = ?
colons = ?

- in general, number of lines = ?

Making the Top of the Torch Scaleable (cont.)

- Compare the two sets of formulas:

SCALE_FACTOR = 2	SCALE_FACTOR = 3
spaces = line – 1	spaces = line – 1
colons = -2 * line + 8	colons = -2 * line + 12

- There's no change in:
  - the formula for spaces
  - the first constant in the formula for colons

- Use a table for the second constant:

SCALE_FACTOR	constant
2	8
3	12

constant = ?

- Scaleable formulas: spaces = line – 1
  colons = ?

Revised Method for Drawing the Top of the Torch

```
public static void drawTop() {
    for (int line = 1; line <= SCALE_FACTOR; line++) {
        // spaces to the left of the current line
        for (int i = 0; i < line – 1; i++) {
            System.out.print(" ");
        }

        // bars and colons on the current line
        System.out.print("|");
        for (int i = 0;i < -2*line + 4*SCALE_FACTOR;i++){
            System.out.print(":");
        }
        System.out.print("|");

        System.out.println();
    }
}
```

Practice: The Torch Handle

```
    ()
   (())
  ((()))
 (((())))
 ========
 |::::::|
  |::::|
1  |::|
2  |::|
3  |::|
4  |::|
   +==+
```

- Pseudocode for default size:

- Java code for default size:

```
public static void drawHandle() {



}
```

Practice: Making the Handle Scaleable

- We again compare two different sizes.

SCALE_FACTOR	# lines	spaces	colons
2	4	2	2
3	6	3	4

- number of lines = ?
  spaces = ?
  colons = ?

```
 |::::::|
  |::::|
1  |::|
2  |::|
3  |::|
4  |::|
```

```
 |::::::::::|
  |::::::::|
   |::::::|
1   |::::|
2   |::::|
3   |::::|
4   |::::|
5   |::::|
6   |::::|
```

Revised Method for Drawing the Handle

- What changes do we need to make?

```
public static void drawHandle() {
    for (int line = 1; line <= 4; line++) {
        for (int i = 0; i < 2; i++) {
            System.out.print(" ");
        }
        System.out.print("|");
        for (int i = 0; i < 2; i++) {
            System.out.print(":");
        }
        System.out.println("|");
    }
}
```

Extra Practice: Printing a Pattern, version 3

- Print a triangular pattern with lines containing n, n – 1, …, 1 asterisks.
  - example:

```
Enter a positive integer: 3
***
**
*
```

- How would we use a nested loop to do this?

```
for ( ______________________________ ) {

    for ( ________________________________ ) {
        System.out.print("*");
    }
    System.out.println();
}
```

Unit 3, Part 1

Methods with Parameters and Return Values

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Static Methods

- We've seen how we can use static methods to:
  1. capture the structure of a program – breaking a task into subtasks
  2. eliminate code duplication
- Thus far, our methods have been limited in their ability to accomplish these tasks.

A Limitation of Simple Static Methods

- For example, in our `DrawTorch` program, there are several `for` loops that each print a series of spaces, such as:

```
for (int i = 0; i < 4 – line; i++) {
    System.out.print(" ");
}

for (int i = 0; i < line – 1; i++) {
    System.out.print(" ");
}
```

- However, despite the fact that all of these loops print spaces, we can't replace them with a method that looks like this:

```
public static void printSpaces() {
    …
```

  Why not?

Parameters

- In order for a method that prints spaces to be useful, we need one that can print an *arbitrary number* of spaces.

- Such a method would allow us to write commands like these:

```
printSpaces(5);
printSpaces(4 – line);
```

  where the number of spaces to be printed is specified between the parentheses.

- To do so, we write a method that has a *parameter:*

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

Parameters (cont.)

- A parameter is a special type of variable that allows us to pass information into a method.
- Consider again this method:

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

- When we execute a method call like

```
printSpaces(10);
```

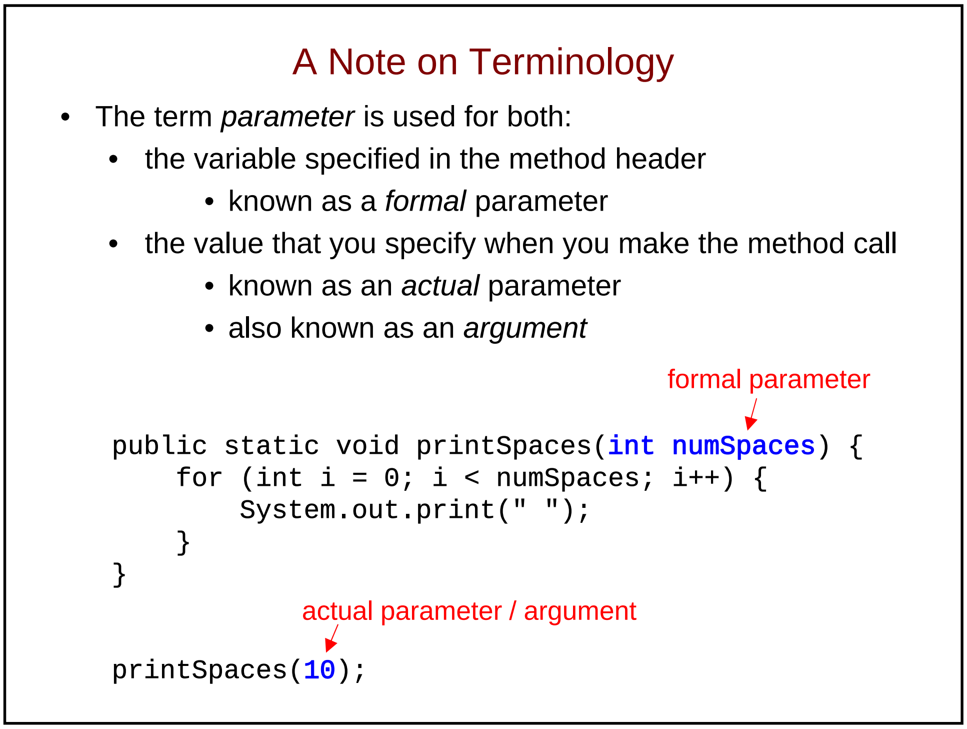
  the expression specified between the parentheses:
  - is evaluated
  - is assigned to the parameter
  - can thereby be used by the code inside the method

Parameters (cont.)

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

- Here's an example with a more complicated expression:

```
int line = 2;
printSpaces(4 – line);
            4 – 2
              2
```



A Note on Terminology

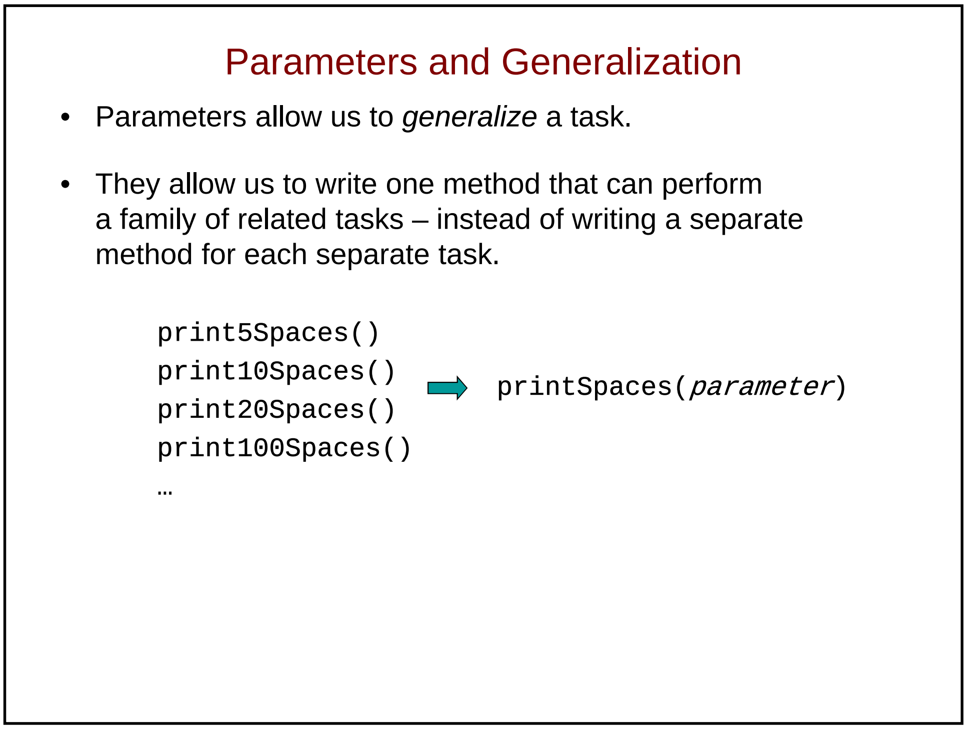
- The term *parameter* is used for both:
  - the variable specified in the method header
    - known as a *formal* parameter
  - the value that you specify when you make the method call
    - known as an *actual* parameter
    - also known as an *argument*

formal parameter

```
public static void printSpaces(int numSpaces) {
    for (int i = 0; i < numSpaces; i++) {
        System.out.print(" ");
    }
}
```

actual parameter / argument

```
printSpaces(10);
```



Parameters and Generalization

- Parameters allow us to *generalize* a task.

- They allow us to write one method that can perform a family of related tasks – instead of writing a separate method for each separate task.

```
print5Spaces()
print10Spaces()    ➡   printSpaces(parameter)
print20Spaces()
print100Spaces()
…
```

Representing Individual Characters

- So far we've learned about two data types:
  - `int`
  - `double`
- The `char` type is used to represent individual characters.
- To specify a `char` literal, we surround the character by single quotes:
  - examples: `'a'` `'Z'` `'0'` `'7'` `'?'` `'\\'`
  - can only represent single characters
  - don’t use double-quotes!

    `"a"` is a string, not a character

Methods with Multiple Parameters

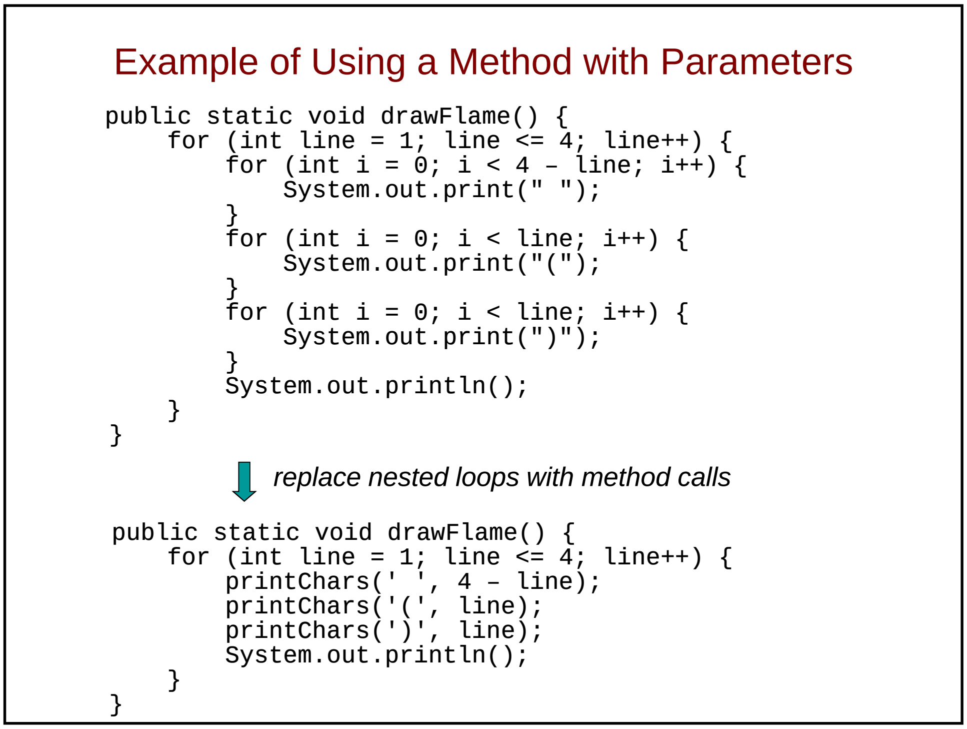
- Here's a method with more than one parameter:

```
public static void printChars(char ch, int num) {
    for (int i = 0; i < num; i++) {
        System.out.print(ch);
    }
}
```

- Example of calling this method:

```
printChars(' ', 10);
```

- Notes:
  - the parameters (both formal and actual) are separated by commas
  - each formal parameter must be preceded by its type
  - the actual parameters are evaluated and assigned to the corresponding formal parameters

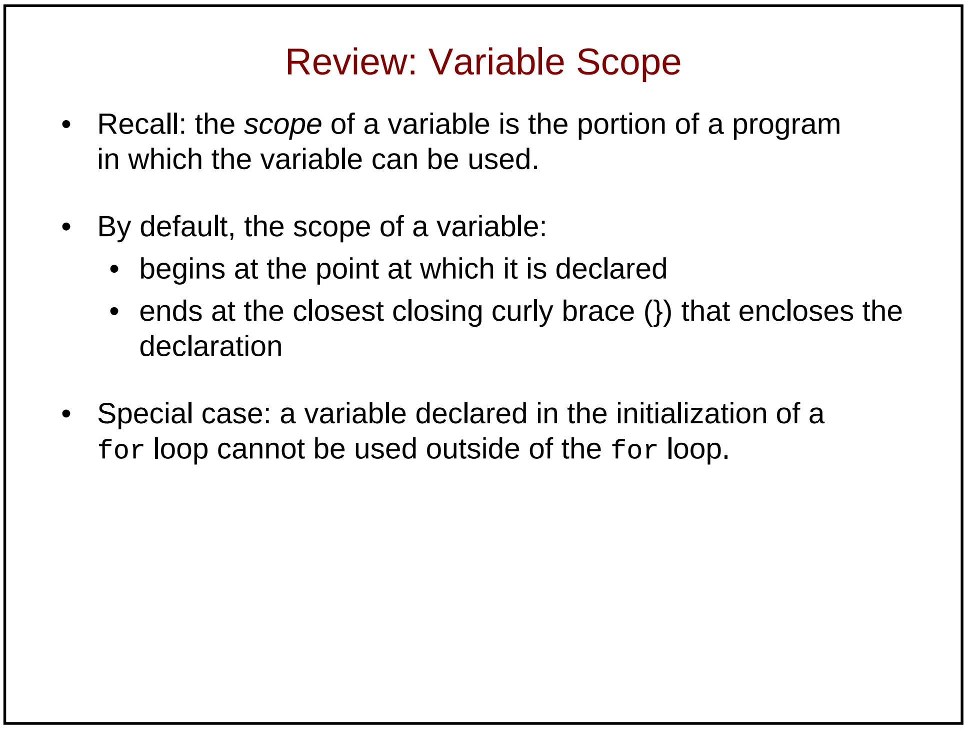


Example of Using a Method with Parameters

```
public static void drawFlame() {
    for (int line = 1; line <= 4; line++) {
        for (int i = 0; i < 4 – line; i++) {
            System.out.print(" ");
        }
        for (int i = 0; i < line; i++) {
            System.out.print("(");
        }
        for (int i = 0; i < line; i++) {
            System.out.print(")");
        }
        System.out.println();
    }
}
```

replace nested loops with method calls

```
public static void drawFlame() {
    for (int line = 1; line <= 4; line++) {
        printChars(' ', 4 – line);
        printChars('(', line);
        printChars(')', line);
        System.out.println();
    }
}
```



Review: Variable Scope

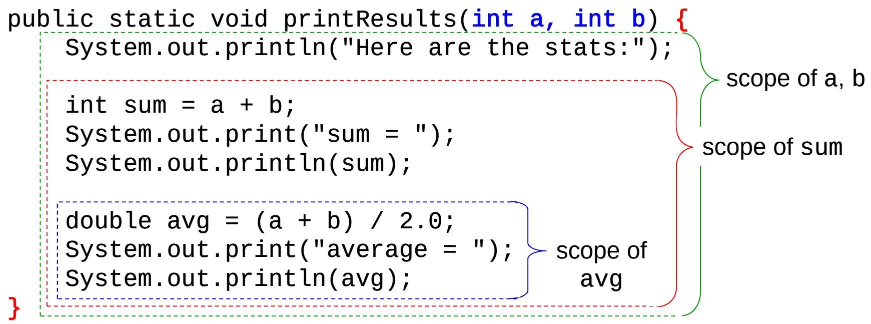
- Recall: the *scope* of a variable is the portion of a program in which the variable can be used.
- By default, the scope of a variable:
  - begins at the point at which it is declared
  - ends at the closest closing curly brace (}) that encloses the declaration
- Special case: a variable declared in the initialization of a `for` loop cannot be used outside of the `for` loop.

Variable Scope and Methods

- Recall: variables declared inside of a method (local variables) follow the standard scope rules.

- *Special case: The scope of the formal parameters of a method is the entire method.*

- Example:



```
public static void printResults(int a, int b) {
    System.out.println("Here are the stats:");

    int sum = a + b;
    System.out.print("sum = ");
    System.out.println(sum);

    double avg = (a + b) / 2.0;
    System.out.print("average = ");
    System.out.println(avg);
}
```

Practice with Scope

```
public static void drawRectangle(int height) {
    for (int i = 0; i < height; i++) {
         // which variables could be used here?
        int width = height * 2;
        for (int j = 0; j < width; j++) {
             System.out.print("*");
              // what about here?
        }
         // what about here?
        System.out.println();
    }
    // what about here?
}

public static void repeatMessage(int numTimes) {
     // what about here?
    for (int i = 0; i < numTimes; i++) {
        System.out.println("What is your scope?");
    }
}
```

Practice with Parameters

```
public static void printValues(int a, int b) {
    System.out.println(a + " " + b);
    b = 2 * a;
    System.out.println("b" + b);
}

public static void main(String[] args) {
    int a = 2;
    int b = 3;
    printValues(b, a);
    printValues(7, b * 3);
    System.out.println(a + " " + b);
}
```

- What's the output?

A Limitation of Parameters

- Parameters allow us to pass values into a method.
- They *don't* allow us to get a value out of a method.

A Limitation of Parameters (cont.)

- Example: using a method to compute the opposite of a number

- This *won't* work:

```
public static void opposite(int number) {
    number = number * -1;
}

public static void main(String[] args) {
    // read in points from the user

    opposite(points);
    …
}
```

  - the `opposite` method changes the value of `number`, but `number` can't be used outside of that method

  - the method *doesn't* change the value of `points`

Methods That Return a Value

- To compute the opposite of a number, we need a method that's able to *return* a value.

- Such a method would allow us to write statements like this:

```
int penalty = opposite(points);
```

- The value returned by the method would *replace* the method call in the original statement.

- Example:

```
int points = 10;
int penalty = opposite(points);
         ⇩
int penalty = -10;  // after the method completes
```

Defining a Method that Returns a Value

- Here's a method that computes and returns the opposite of a number:

```
public static int opposite(int number) {
    return number * -1;
}
```

- In the header of the method, `void` is replaced by `int`, which is the type of the returned value.

- The returned value is specified using a `return` statement. Syntax:

  `return` <expression>`;`

  - <expression> is evaluated
  - the resulting value replaces the method call in the statement that called the method

Defining a Method that Returns a Value (cont.)

- The complete syntax for the header of a static method is:

  `public static` <return type> <name>`(`<type1> <param1>`,`
  <type2> <param2>`,` … `)`

- Note: a method call is a type of expression!
  - it evaluates to its return value

```
int opp = opposite(10);
                 ↓
int opp = -10;
```

- In our earlier methods, the return type was always `void`:

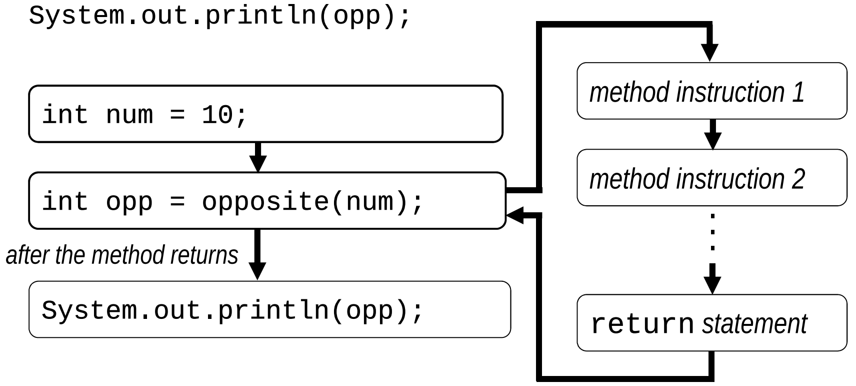
```
public static void printSpaces(int numSpaces) {
    ...
```

  This is a special return type that indicates that no value is returned.

Flow of Control with Methods That Return a Value

- The flow of control jumps to a method until it returns.
- The flow jumps back, and the returned value replaces the call.
- Example:

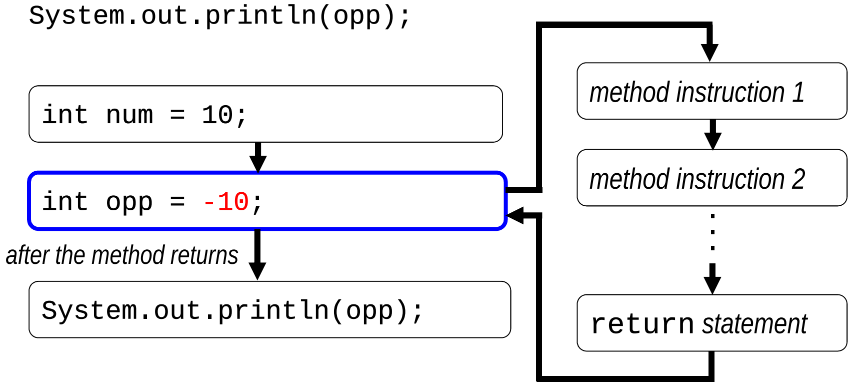
```
int num = 10;
int opp = opposite(num);
System.out.println(opp);
```



Flow of Control with Methods That Return a Value

- The flow of control jumps to a method until it returns.
- The flow jumps back, and *the returned value replaces the call*.
- Example:

```
int num = 10;
int opp = opposite(num);
System.out.println(opp);
```



Returning vs. Printing

- Instead of returning a value, we could write a method that prints the value:

```
public static void printOpposite(int number) {
    System.out.println(number * -1);
}
```

- However, a method that returns a value is typically more useful.
- With such a method, you can still print the value by printing what the method returns:

```
System.out.println(opposite(num));
```

  - the return value replaces the method call and is printed
- In addition, you can do other things besides printing:

```
int penalty = opposite(num);
```

Practice: Computing the Volume of a Cone

- volume of a cone = $\frac{\text{base * height}}{3}$
- Let's write a method named `coneVol` for computing it.
  - parameters and their types?
  - return type?
  - method definition:

```
public static ________ coneVol(______________________________) {



}
```

The `Math` Class

- Java's built-in `Math` class contains static methods for mathematical operations.
- These methods return the result of applying the operation to the parameters.
- Examples:

  `round(double value)` – returns the result of rounding `value` to the nearest integer

  `abs(double value)` – returns the absolute value of `value`

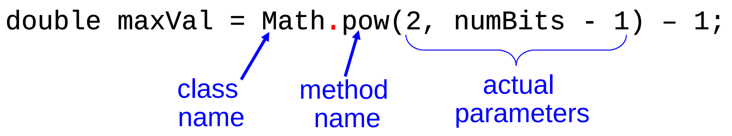
  `pow(double base, double expon)` – returns the result of raising `base` to the `expon` power

  `sqrt(double value)` – returns the square root of `value`
- Table 3.2 in the textbook includes other examples.

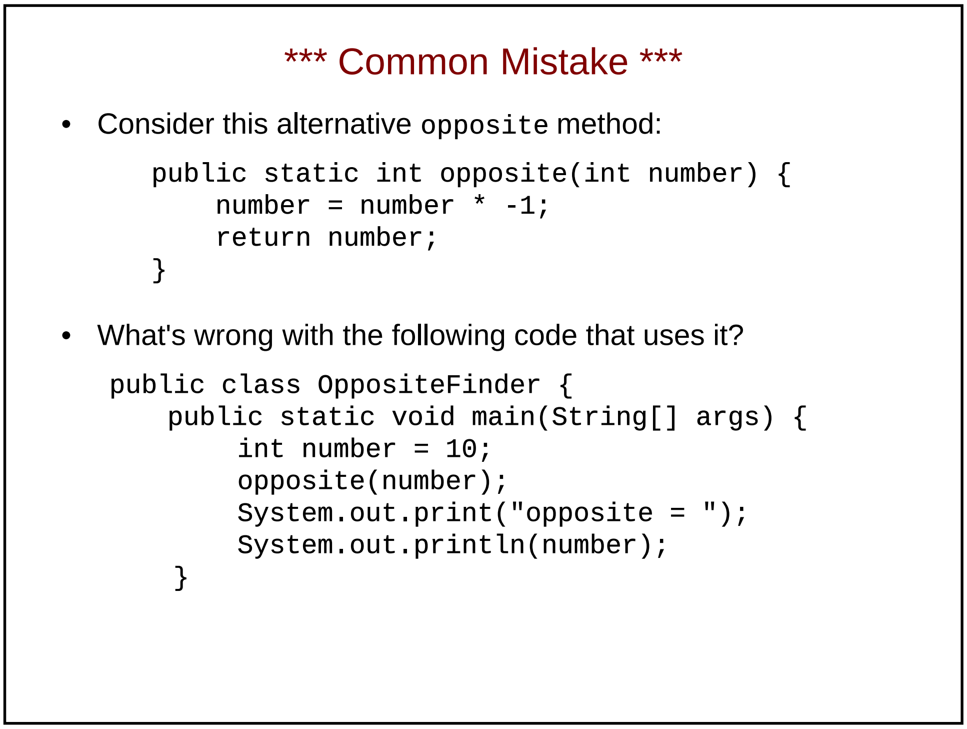
The `Math` Class (cont.)

- To use a static method defined in another class, we need to use the name of the class when we call it.
- We use what's known as *dot notation.*
- Syntax:

  `<class name>.<method name>(<param1>, <param2>, …)`
- Example:



```
double maxVal = Math.pow(2, numBits - 1) – 1;
```



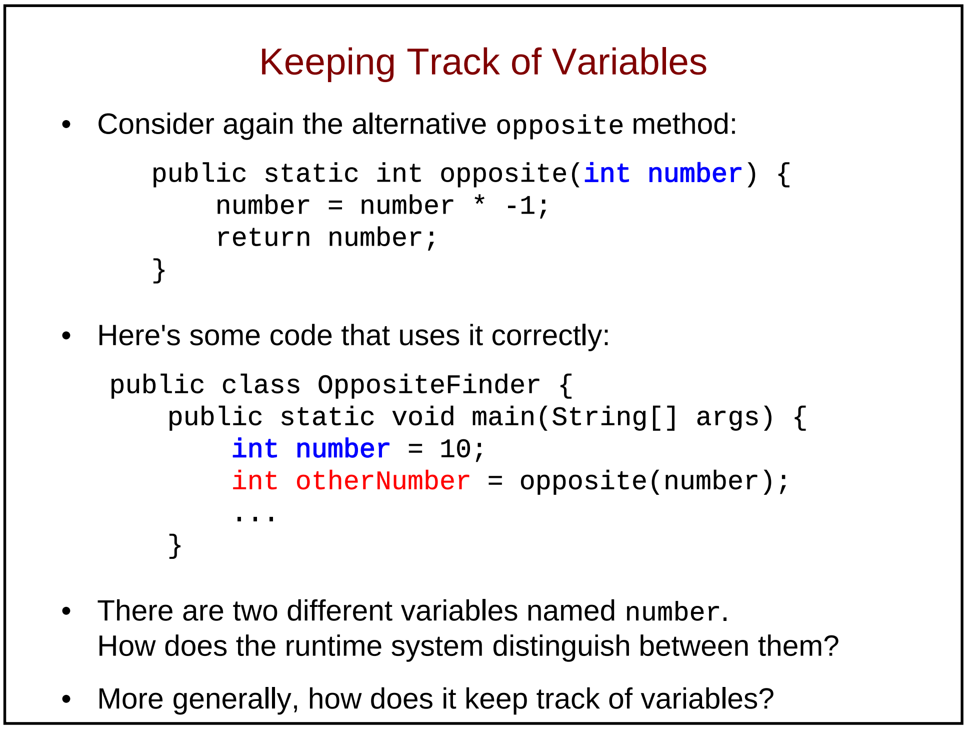
*** Common Mistake ***

- Consider this alternative `opposite` method:

```
public static int opposite(int number) {
    number = number * -1;
    return number;
}
```

- What's wrong with the following code that uses it?

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        opposite(number);
        System.out.print("opposite = ");
        System.out.println(number);
    }
```



Keeping Track of Variables

- Consider again the alternative `opposite` method:

```
public static int opposite(int number) {
    number = number * -1;
    return number;
}
```

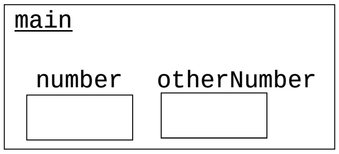
- Here's some code that uses it correctly:

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        ...
    }
```

- There are two different variables named `number`. How does the runtime system distinguish between them?
- More generally, how does it keep track of variables?

Keeping Track of Variables (cont.)

- When you make a method call, the Java runtime sets aside a block of memory known as the *frame* of that method call.

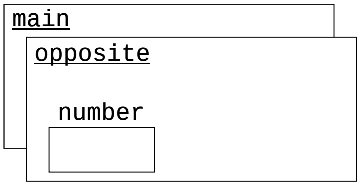


note: we're ignoring `main`*'s parameter for now*

- The frame is used to store:
  - the formal parameters of the method
  - any local variables – variables declared within the method
- A given frame can only be accessed by statements that are part of the corresponding method call.

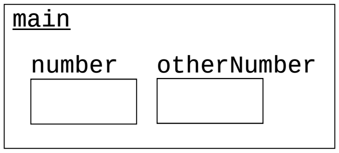
Keeping Track of Variables (cont.)

- When a method (*method1*) calls another method (*method2*), the frame of *method1* is set aside temporarily.
  - *method1*'s frame is "covered up" by the frame of *method2*
  - example: after `main` calls `opposite`, we get:



- When the runtime system encounters a variable, it uses the one from the current frame (the one on top).
- When a method returns, its frame is removed, which "uncovers" the frame of the method that called it.

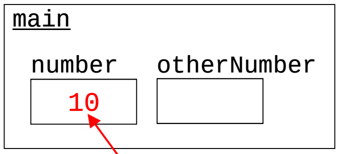
Example: Tracing Through a Program



- A frame is created for the `main` method.

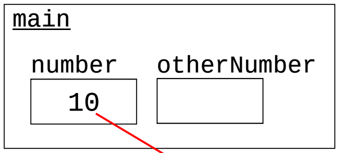
```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

Example: Tracing Through a Program



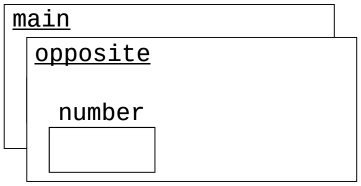
```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

Example: Tracing Through a Program



```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(number);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

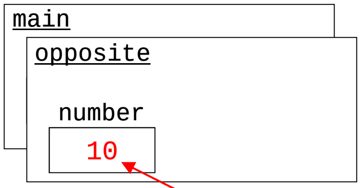
Example: Tracing Through a Program



- A frame is created for the opposite method, and that frame "covers up" the frame for main.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

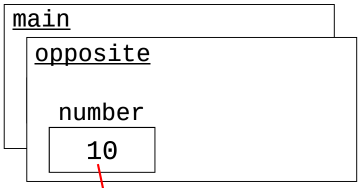
Example: Tracing Through a Program



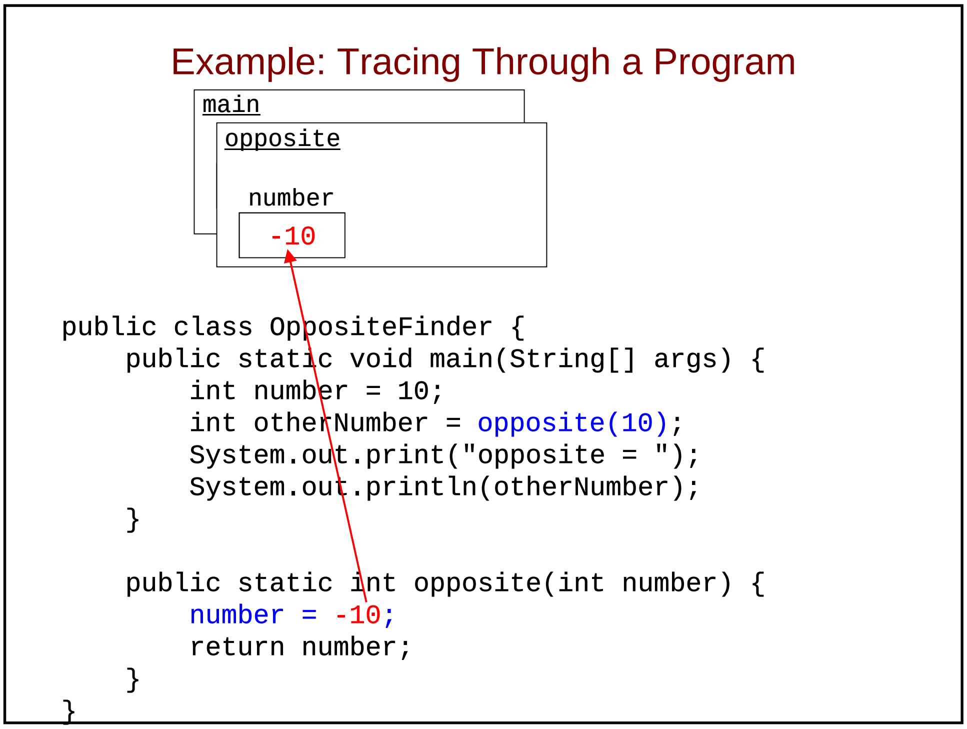
- The actual parameter is passed in and is assigned to the formal parameter.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```

Example: Tracing Through a Program

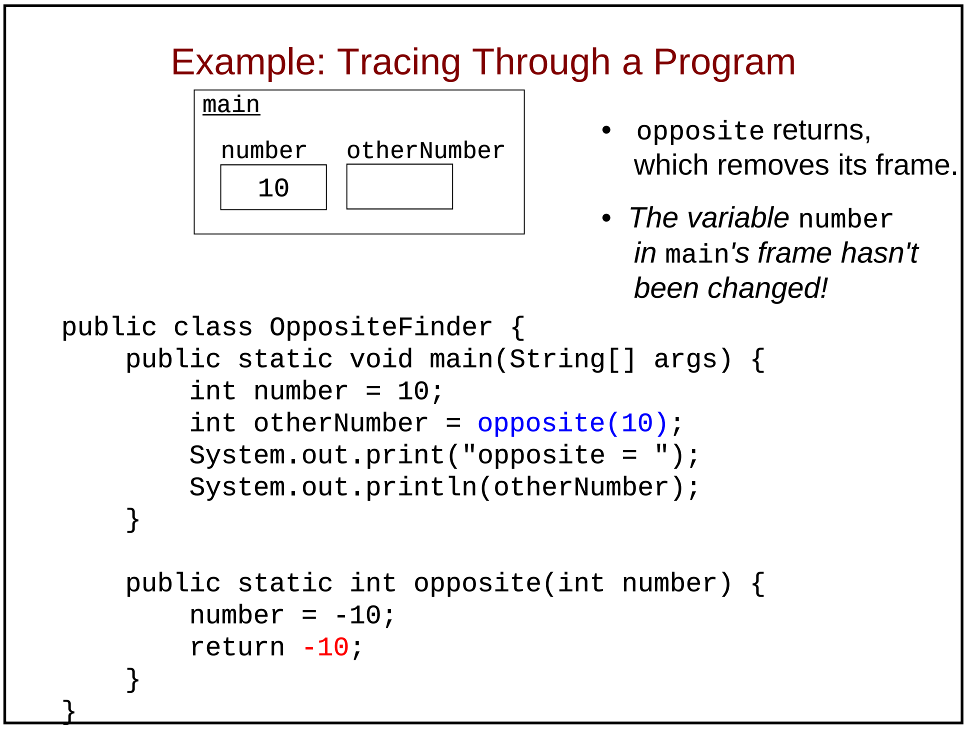


```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = number * -1;
        return number;
    }
}
```



Example: Tracing Through a Program

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return number;
    }
}
```

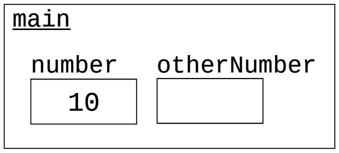


Example: Tracing Through a Program

- `opposite` returns, which removes its frame.
- *The variable* `number` *in* `main`*'s frame hasn't been changed!*

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

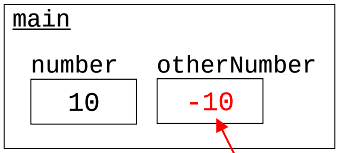
Example: Tracing Through a Program


main
number
otherNumber
10

- The returned value replaces the method call.

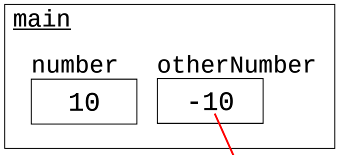
```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = opposite(10);
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Example: Tracing Through a Program


main
number
otherNumber
10
-10

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = -10;
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Example: Tracing Through a Program



```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = -10;
        System.out.print("opposite = ");
        System.out.println(otherNumber);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Example: Tracing Through a Program

- main returns, which removes its frame.

```
public class OppositeFinder {
    public static void main(String[] args) {
        int number = 10;
        int otherNumber = -10;
        System.out.print("opposite = ");
        System.out.println(-10);
    }

    public static int opposite(int number) {
        number = -10;
        return -10;
    }
}
```

Practice

- What is the output of the following program?

```
public class MethodPractice {
    public static int triple(int x) {
         x = x * 3;
         return x;
    }

    public static void main(String[] args) {
         int y = 2;
         y = triple(y);
         System.out.println(y);
         triple(y);
         System.out.println(y);
    }
}
```

More Practice

foo
x | y

```
public class Mystery {
     public static int foo(int x, int y) {
         y = y + 1;
         x = x + y;
         System.out.println(x + " " + y);
         return x;
     }

     public static void main(String[] args) {
         int x = 2;
         int y = 0;

         y = foo(y, x);
         System.out.println(x + " " + y);

         foo(x, x);
         System.out.println(x + " " + y);

         System.out.println(foo(x, y));
         System.out.println(x + " " + y);
     }
}
```

main
x | y

output

From Unstructured to Structured

```
public class TwoTriangles {
    public static void main(String[] args) {
        char ch = '*';       // character used in printing
        int smallBase = 5;   // base length of smaller triangle

        // Print the small triangle.
        for (int line = 1; line <= smallBase; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print(ch);
            }
            System.out.println();
        }

        // Print the large triangle.
        for (int line = 1; line <= 2 * smallBase; line++) {
            for (int i = 0; i < line; i++) {
                System.out.print(ch);
            }
            System.out.println();
        }
     }
}
```

From Unstructured to Structured (cont.)

```
public class TwoTriangles {
    public static void main(String[] args) {
        char ch = '*';       // character used in printing
        int smallBase = 5;   // base length of smaller triangle

        // Print the small triangle.

        printTriangle(_________________________________);

        // Print the large triangle.

        printTriangle(_________________________________);
    }

    public static void printTriangle(_______________________) {




    }
}
```

Unit 3, Part 2

Using Objects from Existing Classes

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

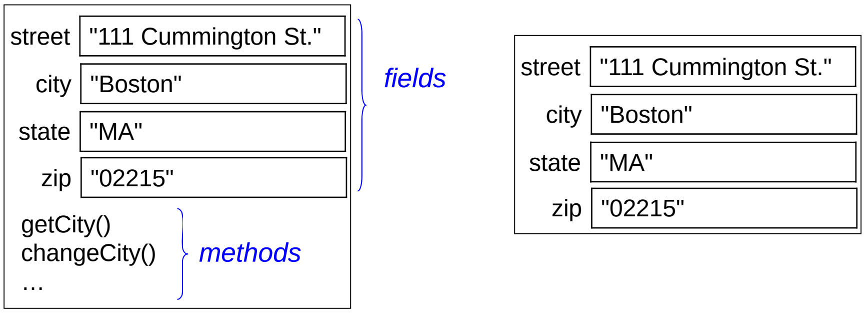
Combining Data and Operations

- The data types that we've seen thus far are referred to as *primitive* data types.
  - `int`, `double`, `char`
  - several others
- Java allows us to use another kind of data known as an *object*.
- An object groups together:
  - one or more data values (the object's *fields*)
  - a set of operations (the object's *methods*)
- Objects in a program are often used to model real-world objects.

Combining Data and Operations (cont.)

- Example: an `Address` object
  - possible fields: *street*, *city*, *state*, *zip*
  - possible operations: *get the city*, *change the city*,
    check if two addresses are equal

- Here are two ways to visualize an `Address` object:



street "111 Cummington St."
city "Boston"
state "MA"
zip "02215"
fields

getCity()
changeCity()
…
methods

street "111 Cummington St."
city "Boston"
state "MA"
zip "02215"

Classes as Blueprints

- We've been using classes as containers for our programs.

- A class can also serve as a blueprint – as the definition of a new type of object.

- The objects of a given class are built according to its blueprint.

- Another analogy:
  - class = cookie cutter
    objects = cookies

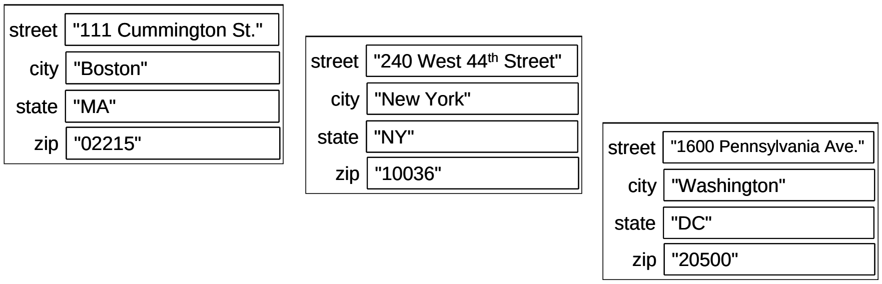
- The objects of a class are also referred to as *instances* of the class.

Class vs. Object

- The `Address` class is a blueprint:

```
public class Address {
    // definitions of the fields
    ...

    // definitions of the methods
    ...
}
```

- `Address` objects are built according to that blueprint:



street	"111 Cummington St."
city	"Boston"
state	"MA"
zip	"02215"

street	"240 West 44th Street"
city	"New York"
state	"NY"
zip	"10036"

street	"1600 Pennsylvania Ave."
city	"Washington"
state	"DC"
zip	"20500"

Using Objects from Existing Classes

- Later in the course, you'll learn how to create your own classes that act as blueprints for objects.
- For now, we'll focus on learning how to use objects from existing classes.

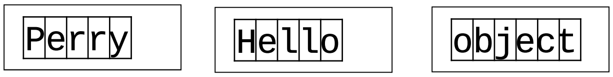
String Objects

- In Java, a string (like `"Hello, world!"`) is actually represented using an object.
    - data values: the characters in the string
    - operations: get the length of the string, get a substring, etc.

- The `String` class defines this type of object:

```
public class String {
    // definitions of the fields
    ...

    // definitions of the methods
    ...
}
```

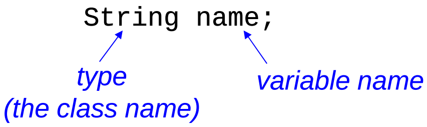
- Individual `String` objects are instances of the `String` class:



Variables for Objects

- When we use a variable to represent an object, the type of the variable is the name of the object's class.

- Here's a declaration of a variable for a `String` object:



```
String name;
```

type (the class name)

variable name

    - we capitalize `String`, because it's a class name

Creating `String` Objects

- One way to create a `String` object is to specify a string literal:

```
String name = "Perry Sullivan";
```

- We create a new `String` from existing `Strings` when we use the + operator to perform concatenation:

```
String firstName = "Perry";
String lastName = "Sullivan";
String fullName = firstName + " " + lastName;
```

- Recall that we can concatenate a `String` with other types of values:

```
String msg = "Perry is " + 6;

// msg now represents "Perry is 6"
```

Using an Object's Methods

- An object's methods are different from the static methods that we've seen thus far.
  - they're called *non-static* or *instance* methods
- An object's methods *belong to* the object. They specify the operations that the object can perform.
- To use a non-static method, we have to specify the object to which the method belongs.
  - use *dot notation*, preceding the method name with the object's variable:

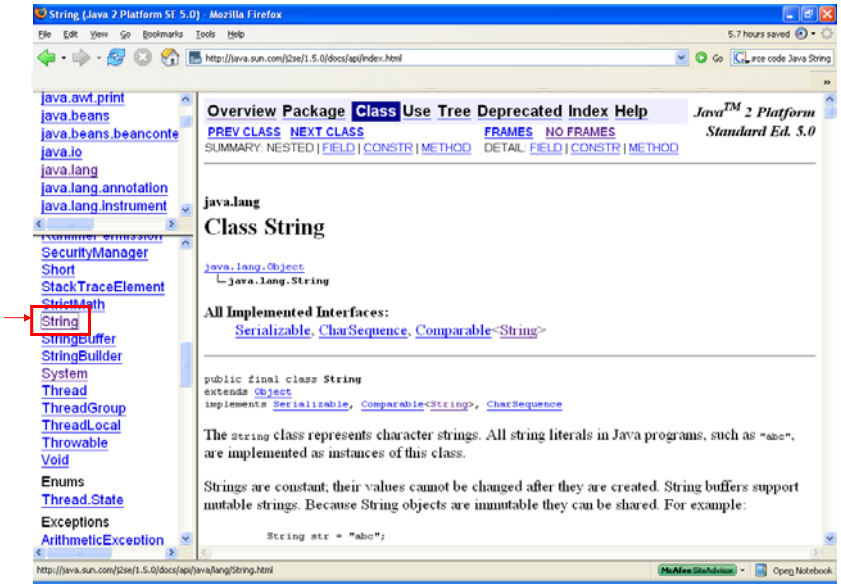
```
String firstName = "Perry";
int len = firstName.length();
```

- Using an object's method is like sending a message to the object, asking it to perform that operation.

The API of a Class

- The methods defined within a class are known as the *API* of that class.
  - API = application programming interface
- We can consult the API of an existing class to determine which operations are supported.
- The API of all classes that come with Java is available here:
  `http://download.oracle.com/javase/7/docs/api/`
  - there's a link on the resources page of the course website

Consulting the Java API



java.lang

Class String

```
java.lang.Object
  └java.lang.String
```

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

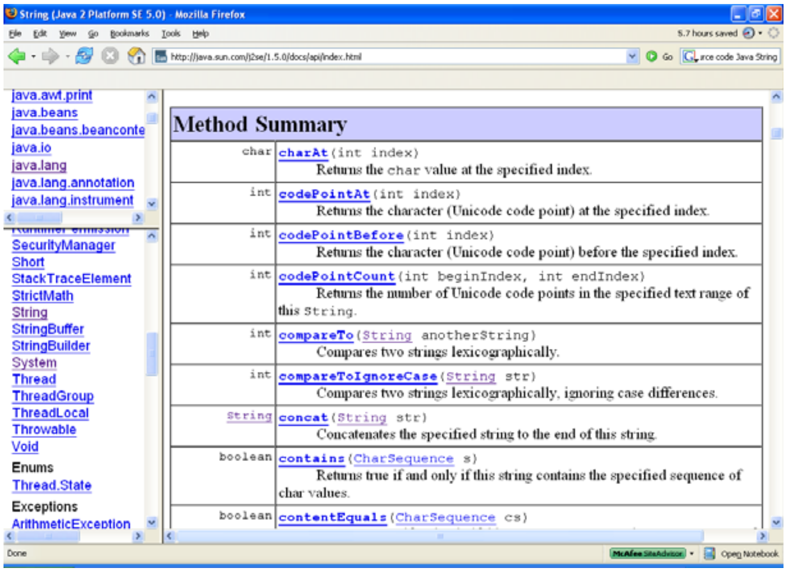
Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

select the class name

Consulting the Java API (cont.)

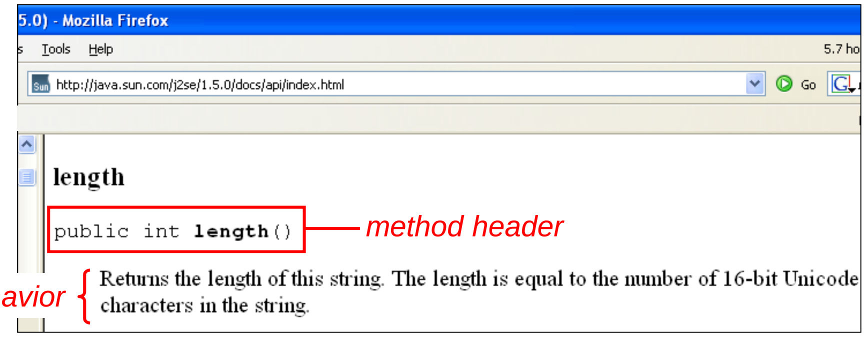
- Scroll down to see a summary of the available methods:



Method Summary	
char	`charAt(int index)` Returns the `char` value at the specified index.
int	`codePointAt(int index)` Returns the character (Unicode code point) at the specified index.
int	`codePointBefore(int index)` Returns the character (Unicode code point) before the specified index.
int	`codePointCount(int beginIndex, int endIndex)` Returns the number of Unicode code points in the specified text range of this `String`.
int	`compareTo(String anotherString)` Compares two strings lexicographically.
int	`compareToIgnoreCase(String str)` Compares two strings lexicographically, ignoring case differences.
String	`concat(String str)` Concatenates the specified string to the end of this string.
boolean	`contains(CharSequence s)` Returns true if and only if this string contains the specified sequence of char values.
boolean	`contentEquals(CharSequence cs)`

Consulting the Java API (cont.)

- Clicking on a method name gives you more information:



length

```
public int length()
```

method header

behavior { Returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.

- From the header, we can determine:
  - the return type: `int`
  - the parameters we need to supply:
    the empty `()` indicates that `length` has no parameters

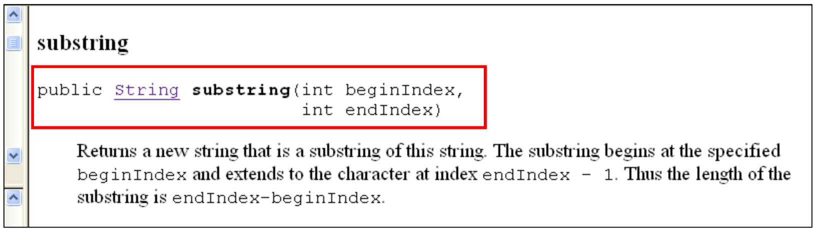
Numbering the Characters in a `String`

- The characters are numbered from left to right, starting from 0.

```
0 1 2 3 4
P e r r y
```

- The position of a character in a string is known as its *index*.
  - `'P'` has an index of 0 in `"Perry"`
  - `'y'` has an index of 4

`substring` Method



substring

```
public String substring(int beginIndex,
                        int endIndex)
```

Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Thus the length of the substring is `endIndex-beginIndex`.

<u>`String substring(int beginIndex, int endIndex)`</u>

- return type: `?`
- parameters: `?`
- behavior: returns the substring that:
  - begins at `beginIndex`
  - ends at `endIndex` – `1`

`substring` Method (cont.)

- To extract a substring of length *N*, you can just figure out `beginIndex` and do:

  ```
  substring(beginIndex, beginIndex + N)
  ```

  - example: consider again this string:

    ```
    String name = "Perry Sullivan";
    ```

    To extract a substring containing the first 5 characters, we can do this:

    ```
    String first = name.substring(0, 5);
    ```

Review: Calling a Method

- Consider this code fragment:

  ```
  String name = "Perry Sullivan";
  int start = 6;
  String last = name.substring(start, start + 8);
  ```

- Steps for executing the method call:
  1. the actual parameters are evaluated to give:
     ```
     String last = name.substring(6, 14);
     ```
  2. a frame is created for the method, and the actual parameters are assigned to the formal parameters
  3. flow of control jumps to the method, which creates and returns the substring `"Sullivan"`
  4. flow of control jumps back, and the returned value replaces the method call:
     ```
     String last = "Sullivan";
     ```

Common Syntax Errors

- Specifying the types of the actual parameters:

```
String name = "Perry Sullivan";
int start = 0;
int end = 5;

// this won't compile
String first = name.substring(~~int~~ start, ~~int~~ end);

// this will
String first = name.substring(start, end);
```

- Calling a method using a variable that hasn't been initialized:

```
String name;
int len = name.length();  // produces an error
```

Another `String` Method

`String toUpperCase()`
returns a new `String` in which all of the letters in the original `String` are converted to upper-case letters

- Example:

```
String warning = "Start the problem set ASAP!";
System.out.println(warning.toUpperCase());
                              ↓
System.out.println("START THE PROBLEM SET ASAP!");
```

- `toUpperCase()` creates and returns a new `String`.
  It does *not* change the original `String`.
- In fact, it's *never* possible to change an existing `String` object.
- We say that `String`s are *immutable* objects.

charAt Method

char charAt(int index)

- return type: char
- parameter list: (int index)
- returns the character at position index in the string
- example:

```
String name = "Perry Sullivan";
System.out.println(name.charAt(0) + "" +
  name.charAt(6));
```

indexOf Method

int indexOf(char ch)

- return type: int
- parameter list: (char ch)
- returns:
  - the index of the first occurrence of ch in the string
  - -1 if the ch does not appear in the string
- examples:

```
String name = "Perry Sullivan";
System.out.println(name.indexOf('r'));
System.out.println(name.indexOf('X'));
```

The Signature of a Method

- The *signature* of a method consists of:
    - its name
    - the number and types of its parameters

```
public String substring(int beginIndex, int endIndex)
```

the signature

- A class cannot include two methods with the same signature.

Two Methods with the Same Name

- There are actually two `String` methods named `substring`:

```
String substring(int beginIndex, int endIndex)

String substring(int beginIndex)
```

  - returns the substring that begins at `beginIndex` and continues to the end of the string

- Do these two methods have the same signature?

- Giving two methods the same name is known as *method overloading*.

- When you call an overloaded method, the compiler uses the number and types of the actual parameters to figure out which version to use.

Console Input Using a `Scanner` Object

- We’ve been printing text in the console window.
- You can also ask the user to enter a value in that window.
  - known as console input
- To do so, we use a type of object known as a `Scanner`.
  - recall PS 2

Packages

- Java groups related classes into *packages*.
- Many classes are part of the `java.lang` package.
  - examples: `String`, `Math`
  - We don't need to tell the compiler where to find these classes.
- If a class is in another package, we need to use an `import` statement so that the compiler will be able to find it.
  - put it *before* the definition of the class
- The `Scanner` class is in the `java.util` package, so we do this:

```
import java.util.*;

public class MyProgram {
    ...
```

Creating an Object

- `String` objects are different from other objects, because we're able to create them using literals.
- To create an object, we typically use a special method known as a *constructor*.
- Syntax:

```
<variable> = new <ClassName>(<parameters>);
or
<type> <variable> = new <ClassName>(<parameters>);
```

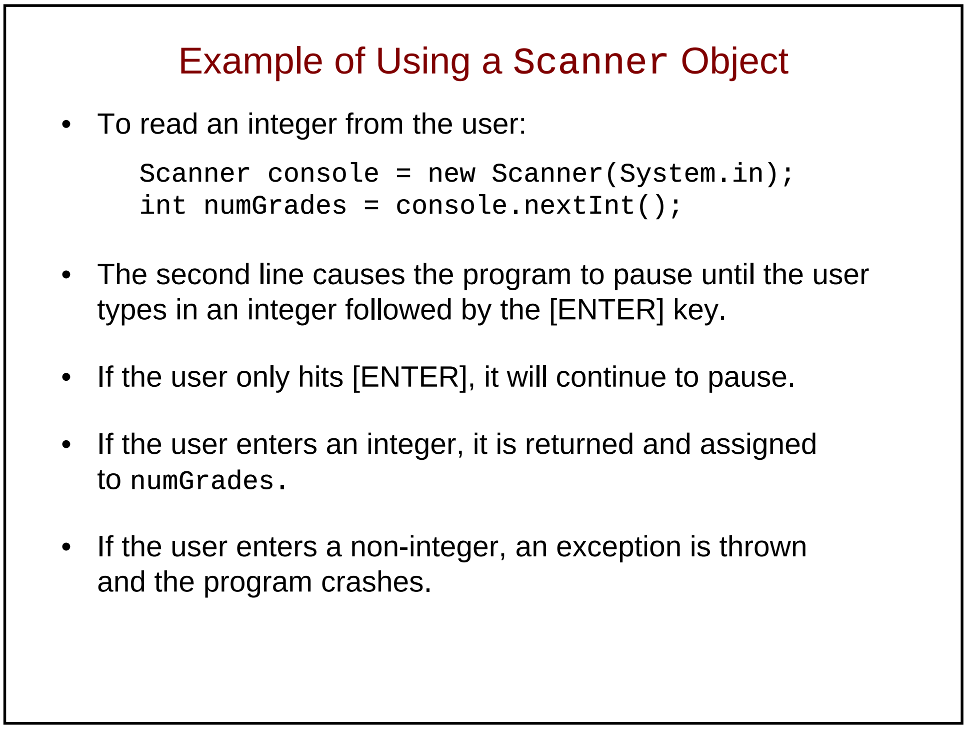
- To create a `Scanner` object for console input:

```
Scanner console = new Scanner(System.in);
```

the parameter tells the constructor that we want the `Scanner` to read from the *standard input* (i.e., the keyboard)

`Scanner` Methods: A Partial List

- `String next()`
  - read in a single "word" and return it
- `int nextInt()`
  - read in an integer and return it
- `double nextDouble()`
  - read in a floating-point value and return it
- `String nextLine()`
  - read in a "line" of input (could be multiple words) and return it

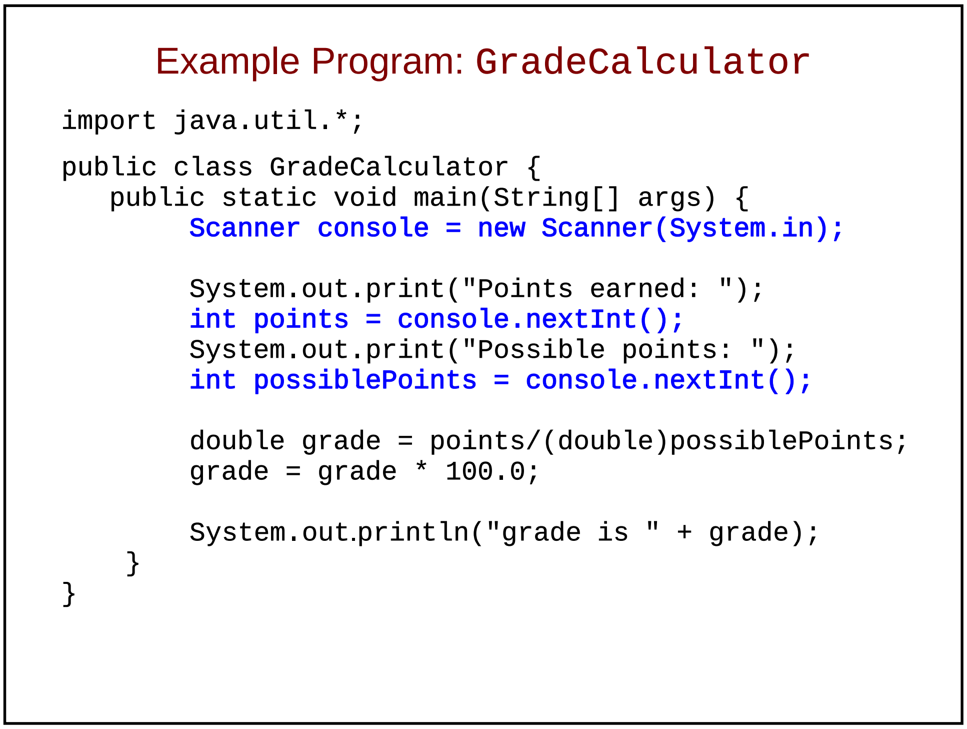


Example of Using a Scanner Object

- To read an integer from the user:

```
Scanner console = new Scanner(System.in);
int numGrades = console.nextInt();
```

- The second line causes the program to pause until the user types in an integer followed by the [ENTER] key.
- If the user only hits [ENTER], it will continue to pause.
- If the user enters an integer, it is returned and assigned to `numGrades.`
- If the user enters a non-integer, an exception is thrown and the program crashes.



Example Program: GradeCalculator

```
import java.util.*;

public class GradeCalculator {
   public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Points earned: ");
        int points = console.nextInt();
        System.out.print("Possible points: ");
        int possiblePoints = console.nextInt();

        double grade = points/(double)possiblePoints;
        grade = grade * 100.0;

        System.out.println("grade is " + grade);
    }
}
```

Important Note About Console Input

- When writing an interactive program that involves user input in methods other than `main`, you should:
  - *create a **single*** `Scanner` *object in **the first line** of the* `main` *method*
  - pass that object into any other method that needs it
- This allows you to avoid creating multiple objects that all do the same thing.
- It also facilitates our grading, because it allows us to provide a series of inputs using a file instead of the keyboard.

Important Note About Console Input (cont.)

- Example:

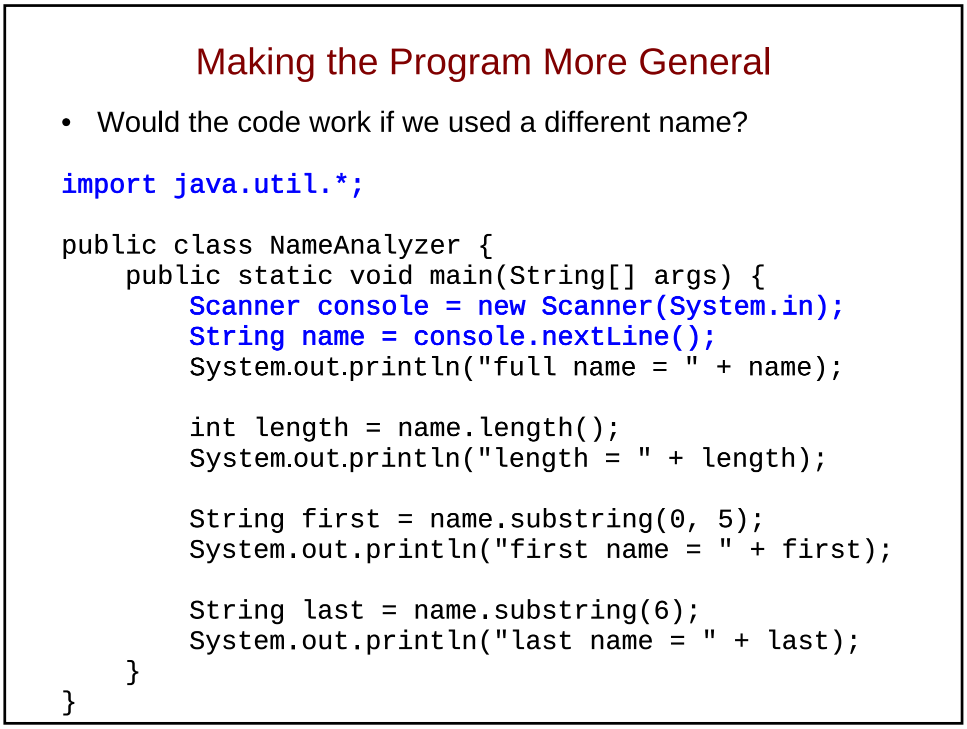
```
public class MyProgram {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        String str1 = getString(console);
        String str2 = getString(console);
        System.out.println(str1 + " " + str2);
    }

    public static String getString(Scanner console) {
        System.out.print("Enter a string: ");
        String str = console.next();
        return str;
    }
}
```

What's Wrong with the Following?

```
public class LengthConverter {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        int cm = getInches(console) * 2.54;
        System.out.println(getInches(console)
          + " inches = " + cm + " cm");
    }

    public static int getInches(Scanner console) {
        System.out.print("Enter a length in inches: ");
        int inches = console.nextInt();
        return int;
    }
}
```

Exercise: Analyzing a Name: First Version

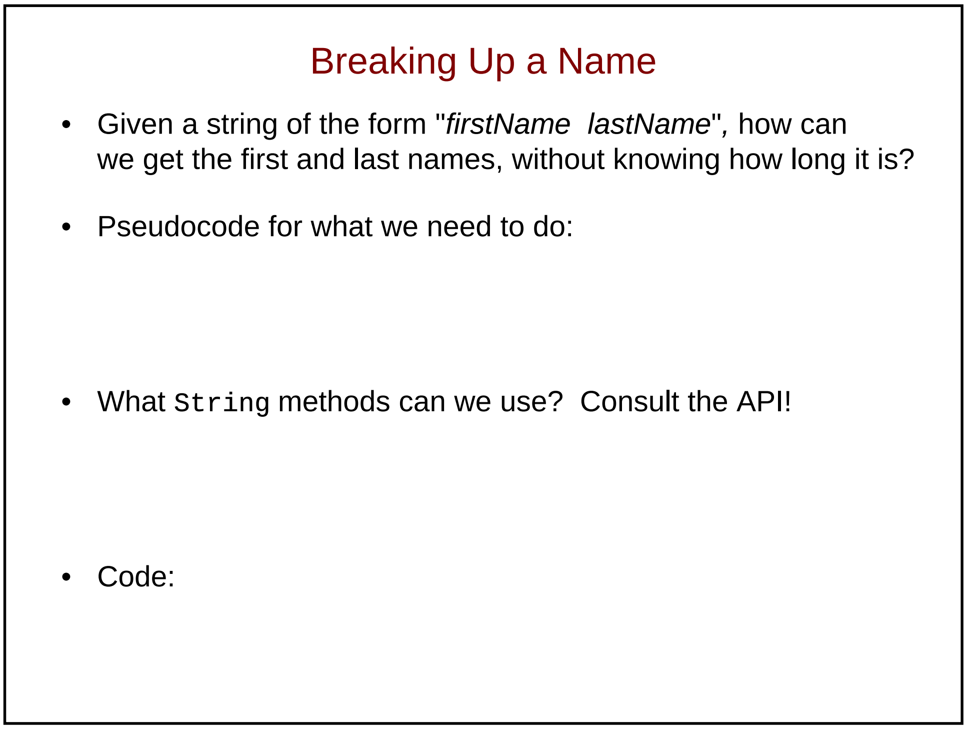
```
public class NameAnalyzer {
    public static void main(String[] args) {
        String name = "Perry Sullivan";
        System.out.println("full name = " + name);

        int length = name.length();
        System.out.println("length = " + length);

        String first = name.substring(0, 5);
        System.out.println("first name = " + first);

        String last = name.substring(6);
        System.out.println("last name = " + last);
    }
}
```



Making the Program More General

- Would the code work if we used a different name?

```
import java.util.*;

public class NameAnalyzer {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        String name = console.nextLine();
        System.out.println("full name = " + name);

        int length = name.length();
        System.out.println("length = " + length);

        String first = name.substring(0, 5);
        System.out.println("first name = " + first);

        String last = name.substring(6);
        System.out.println("last name = " + last);
    }
}
```



Breaking Up a Name

- Given a string of the form "*firstName lastName*", how can we get the first and last names, without knowing how long it is?
- Pseudocode for what we need to do:
- What `String` methods can we use? Consult the API!
- Code:

Static Methods for Breaking Up a Name

- How could we rewrite our name analyzer to use separate methods for extracting the first and last names?

```
public static _________ firstName(_______________) {





}

public static _________ lastName(_______________) {





}
```

Using the Static Methods

- Given the methods from the previous slide, what would the `main` method now look like?

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    String name = console.nextLine();
    System.out.println("full name = " + name);

    int length = name.length();
    System.out.println("length = " + length);








}
```

Processing a String One Character at a Time

- Write a method for printing the name vertically, one char per line.

```
import java.util.*;

public class NameAnalyzer {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        String name = console.nextLine();
        System.out.println("full name = " + name);
        ...
        printVertical(name);
    }

    public static _____ printVertical(_______________){

        for (int i = 0; i < _______________; i++) {



        }
    }
}
```

Scanner Objects and Tokens

- Most `Scanner` methods read one *token* at a time.

- Tokens are separated by whitespace (spaces, tabs, newlines).
  - example: if the user enters the line

    ```
    wow, I slept for 9 hours!\n
    ```

    newline character, which you get when you hit [ENTER]

    there are six tokens:
    - `wow,`
    - `I`
    - `slept`
    - `for`
    - `9`
    - `hours!`

Scanner Objects and Tokens (cont.)

- Consider the following lines of code:

```
System.out.print("Enter the length and width: ");
int length = console.nextInt();
int width = console.nextInt();
```

- Because the `nextInt()` method reads one token at a time, the user can either:
  - enter the two numbers on the same line, separated by one or more whitespace characters

    ```
    Enter the length and width: 30 15
    ```

  - enter the two numbers on different lines

    ```
    Enter the length and width: 30
    15
    ```

nextLine Method

- The `nextLine()` method does not just read a single token.
- Using `nextLine` can lead to unexpected behavior, for reasons that we'll discuss later on.
- Avoid it for now!

Additional Terminology

- To avoid having too many new terms at once, I've limited the terminology introduced in these notes.

- Here are some additional terms related to classes, objects, and methods:
  - *invoking* a method = calling a method
  - method *invocation* = method call
  - the *called object* = the object used to make a method call
  - *instantiate* an object = create an object
  - *members* of a class = the fields and methods of a class

Unit 3, Part 3

Conditional Execution

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Simple Conditional Execution in Java

```
if (<condition>) {
    <true block>
} else {
    <false block>
}
```

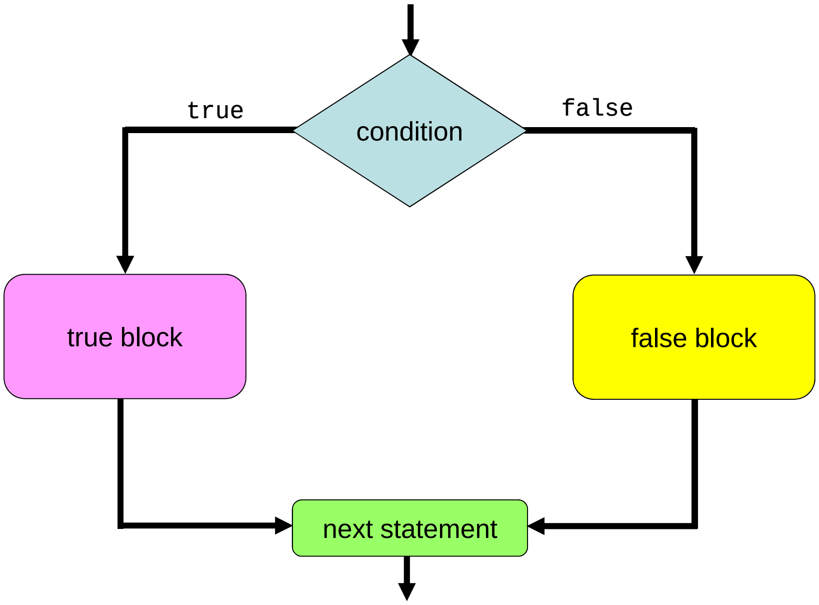
```
if (<condition>) {
    <true block>
}
```

- If the condition is true:
  - the statement(s) in the true block are executed
  - the statement(s) in the false block (if any) are skipped
- If the condition is false:
  - the statement(s) in the false block (if any) are executed
  - the statement(s) in the true block are skipped

Example: Analyzing a Number

```
Scanner console = new Scanner(System.in);
System.out.print("Enter an integer: ");
int num = console.nextInt();

if (num % 2 == 0) {
    System.out.println(num + " is even.");
} else {
    System.out.println(num + " is odd.");
}
```

Flowchart for an `if-else` Statement



Common Mistake

- You should not put a semi-colon after an if-statement header:

```
if (num % 2 == 0); {
    System.out.println(…);
    ...
}
```

- The semi-colon ends the `if` statement.
  - thus, it has an empty true block
- The println and other statements are independent of the `if` statement, and always execute.

Choosing at Most One of Several Options

- Consider this code:

```
if (num < 0) {
    System.out.println("The number is negative.");
}
if (num > 0) {
    System.out.println("The number is positive.");
}
if (num == 0) {
    System.out.println("The number is zero.");
}
```

- All three conditions are evaluated, but at most one of them can be true (in this case, *exactly* one).

Choosing at Most One of Several Options (cont.)

- We can do this instead:

```
if (num < 0) {
    System.out.println("The number is negative.");
}
else if (num > 0) {
    System.out.println("The number is positive.");
}
else if (num == 0) {
    System.out.println("The number is zero.");
}
```

- If the first condition is true, it will skip the second and third.
- If the first condition is false, it will evaluate the second, and if the second condition is true, it will skip the third.
- If the second condition is false, it will evaluate the third, etc.

Choosing at Most One of Several Options (cont.)

- We can also make things more compact as follows:

```
if (num < 0) {
    System.out.println("The number is negative.");
} else if (num > 0) {
    System.out.println("The number is positive.");
} else if (num == 0) {
    System.out.println("The number is zero.");
}
```

- This emphasizes that the entire thing is one compound statement.

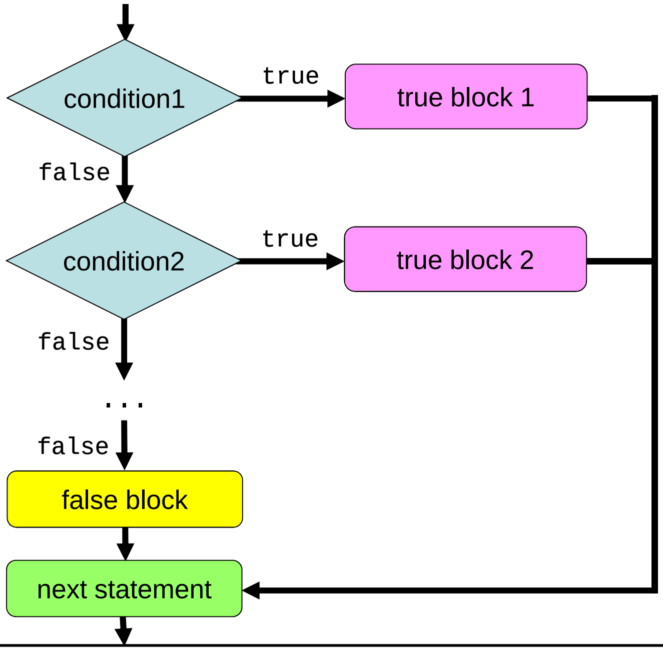
`if-else if` Statements

- Syntax:

```
if (<condition1>) {
    <true block for condition1>
} else if (<condition2>) {
    <true block for condition2>
}
…
} else {
    <false block for all of the conditions>
}
```

- The conditions are evaluated in order.
  The true block of the *first* true condition is executed.
  All of the remaining conditions and their blocks are skipped.
- If no condition is true, the false block (if any) is executed.

Flowchart for an `if-else if` Statement



Choosing Exactly One Option

- Consider again this code fragment:

```
if (num < 0) {
    System.out.println("The number is negative.");
} else if (num > 0) {
    System.out.println("The number is positive.");
} else if (num == 0) {
    System.out.println("The number is zero.");
}
```

- One of the conditions must be true, so we can omit the last one:

```
if (num < 0) {
    System.out.println("The number is negative.");
} else if (num > 0) {
    System.out.println("The number is positive.");
} else {
    System.out.println("The number is zero.");
}
```

Types of Conditional Execution

- If it want to execute **any number** of several conditional blocks, use **sequential `if` statements**:

```
if (num < 0) {
    System.out.println("The number is negative.");
}
if (num % 2 == 0) {
    System.out.println("The number is even.");
}
```

- If you want to execute **at most one (i.e., 0 or 1)** of several blocks, use an **`if-else if` statement ending in `else if`**:

```
if (num < 0) {
    System.out.println("The number is negative.");
} else if (num > 0) {
    System.out.println("The number is positive.");
}
```

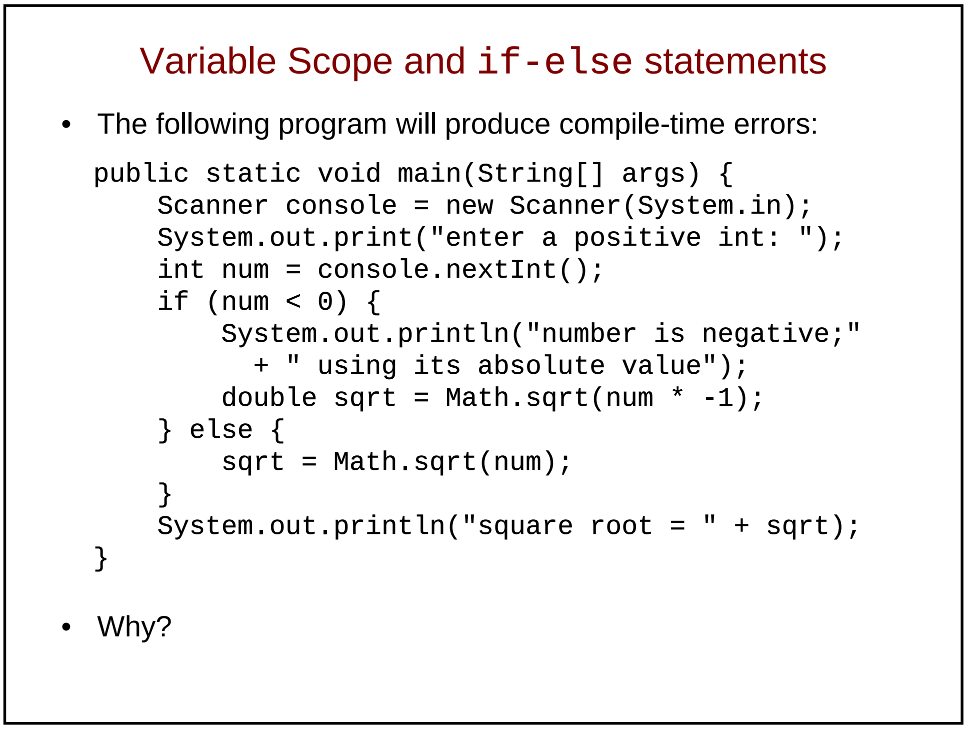
- If you want to execute **exactly one** of several blocks, use an **`if-else if` ending in just `else`** (see bottom of last slide).

Find the Logic Error

```
Scanner console = new Scanner(System.in);

System.out.print("Enter the student's score: ");
int score = console.nextInt();

String grade;
if (score >= 90) {
    grade = "A";
}
if (score >= 80) {
    grade = "B";
}
if (score >= 70) {
    grade = "C";
}
if (score >= 60) {
    grade = "D";
}
if (score < 60) {
    grade = "F";
}
```

Review: Variable Scope

- Recall: the *scope* of a variable is the portion of a program in which the variable can be used.
- By default, the scope of a variable:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration
- Because of these rules, a variable cannot be used outside of the block in which it is declared.

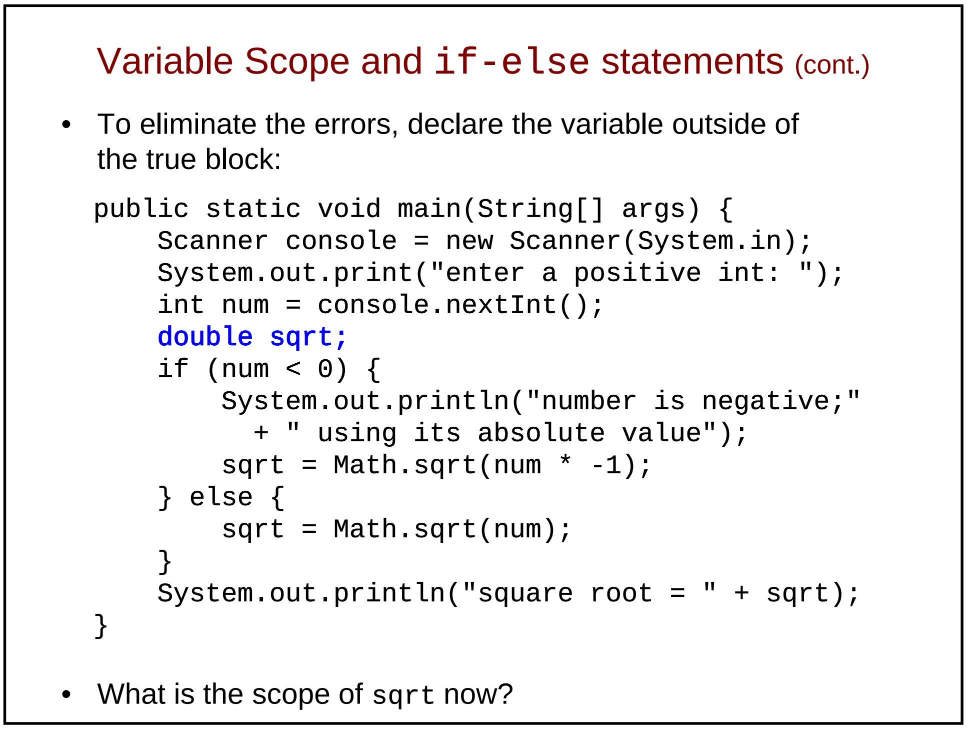


Variable Scope and `if-else` statements

- The following program will produce compile-time errors:

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("enter a positive int: ");
    int num = console.nextInt();
    if (num < 0) {
        System.out.println("number is negative;"
          + " using its absolute value");
        double sqrt = Math.sqrt(num * -1);
    } else {
        sqrt = Math.sqrt(num);
    }
    System.out.println("square root = " + sqrt);
}
```

- Why?



Variable Scope and `if-else` statements (cont.)

- To eliminate the errors, declare the variable outside of the true block:

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("enter a positive int: ");
    int num = console.nextInt();
    double sqrt;
    if (num < 0) {
        System.out.println("number is negative;"
          + " using its absolute value");
        sqrt = Math.sqrt(num * -1);
    } else {
        sqrt = Math.sqrt(num);
    }
    System.out.println("square root = " + sqrt);
}
```

- What is the scope of `sqrt` now?

Review: Loop Patterns for n Repetitions

- Thus far, we've mainly used `for` loops to repeat something a definite number of times.
- We've seen two different patterns for this:
  - pattern 1:

    ```
    for (int i = 0; i < n; i++) {
        <statements to repeat>
    }
    ```

  - pattern 2:

    ```
    for (int i = 1; i <= n; i++) {
        <statements to repeat>
    }
    ```

Another Loop Pattern: Cumulative Sum

- We can also use a `for` loop to add up a set of numbers.
- Basic pattern (using pseudocode):

  ```
  sum = 0
  for (all of the numbers that we want to sum) {
      num = the next number
      sum = sum + num
  }
  ```

Example of Using a Cumulative Sum

```
public class GradeAverager {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("number of grades? ");
        int numGrades = console.nextInt();

        if (numGrades <= 0) {
            System.out.println("nothing to average");
        } else {
            int sum = 0;
            for (int i = 1; i <= numGrades; i++) {
                System.out.print("grade #" + i + ": ");
                int grade = console.nextInt();
                sum = sum + grade;
            }

            System.out.println("The average is " +
              (double)sum / numGrades);
        }
    }
}
```

- Note the use of an `if-else` statement to handle invalid user inputs.

Tracing Through a Cumulative Sum

- Let's trace through this code.

```
int sum = 0;
for (int i = 1; i <= numGrades; i++) {
    System.out.print("grade #" + i + ": ");
    int grade = console.nextInt();
    sum = sum + grade;
}
```

assuming that the user enters these grades: 80, 90, 84.

`numGrades = 3`

i	i <= numGrades	grade	sum

Conditional Execution and Return Values

- With conditional execution, it's possible to write a method with more than one `return` statement.
  - example:

```
public static int min(int a, int b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

- Only one of the `return` statements is executed.
- As soon as you reach a return statement, the method's execution stops and the specified value is returned.
  - the rest of the method is not executed

Conditional Execution and Return Values (cont.)

- Instead of writing the method this way:

```
public static int min(int a, int b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

we could instead write it like this, without the `else`:

```
public static int min(int a, int b) {
    if (a < b) {
        return a;
    }
    return b;
}
```

- Why is this equivalent?

Conditional Execution and Return Values (cont.)

- Consider this method, which has a compile-time error:

```
public static int compare(int a, int b) {
    if (a < b) {
        return -1;
    } else if (a > b) {
        return 1;
    } else if (a == b) {
        return 0;
    }
}
```

- Because all of the `return` statements are connected to conditions, the compiler worries that no value will be returned.

Conditional Execution and Return Values (cont.)

- Here's one way to fix it:

```
public static int compare(int a, int b) {
    if (a < b) {
        return -1;
    } else if (a > b) {
        return 1;
    } else {
        return 0;
    }
}
```

Conditional Execution and Return Values (cont.)

- Here's another way:

```
public static int compare(int a, int b) {
    if (a < b) {
        return -1;
    } else if (a > b) {
        return 1;
    }

    return 0;
}
```

- Both fixes allow the compiler to know for certain that a value will *always* be returned.

Returning From a `void` Method

```
public static void repeat(String msg, int n) {
    if (n <= 0) {     // special cases
        return;
    }

    for (int i = 0; i < n; i++) {
        System.out.println(msg);
    }
}
```

- Note that this method has a return type of `void`.
  - it doesn't return a value.
- However, it still has a `return` statement.
  - used to break out of the method
  - note that there's nothing between the `return` and the `;`

Testing for Equivalent Primitive Values

- The `==` and `!=` operators are used when comparing primitives.
  - `int`, `double`, `char`, etc.

- Example:

```
Scanner console = new Scanner(System.in);
...
System.out.print("Do you have another (y/n)? ");
char choice = console.next().charAt(0);
if (choice == 'y') {    // this works just fine
    processItem();
} else if (choice == 'n') {
    return;
} else {
    System.out.println("invalid input");
}
```

Testing for Equivalent Objects

- The `==` and `!=` operators do *not* typically work when comparing *objects*. (We'll see why this is later.)

- Example:

```
Scanner console = new Scanner(System.in);
System.out.print("regular or diet? ");
String choice = console.next();
if (choice == "regular") {  // doesn't work
    processRegular();
} else {
    ...
}
```

- `choice == "regular"` compiles, but it evaluates to `false`, even when the user does enter `"regular"`!

Testing for Equivalent Objects (cont.)

- We use a special method called the `equals` method to test if two objects are equivalent.
  - example:

```
Scanner console = new Scanner(System.in);
System.out.print("regular or diet? ");
String choice = console.next();
if (choice.equals("regular")) {
    processRegular();
} else {
    ...
}
```

- `choice.equals("regular")` compares the string represented by the variable `choice` with the string `"regular"`
  - returns `true` when they are equivalent
  - returns `false` when they are not

equalsIgnoreCase()

- We often want to compare two strings without paying attention to the case of the letters.
  - example: we want to treat as equivalent:

    `"regular"`

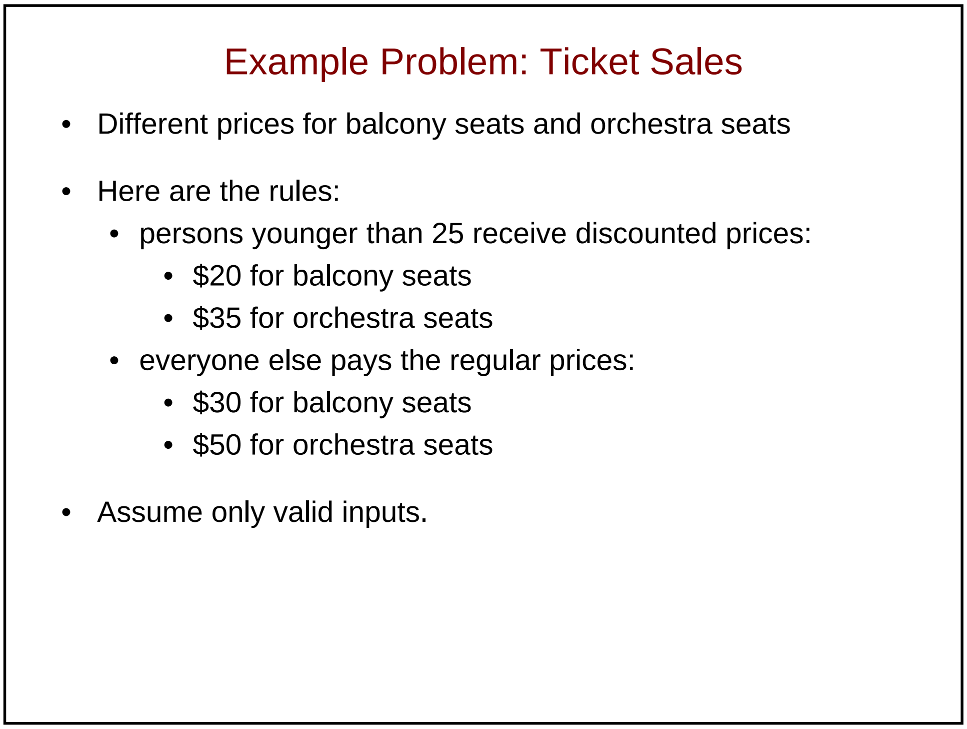
    `"Regular"`

    `"REGULAR"`

    etc.

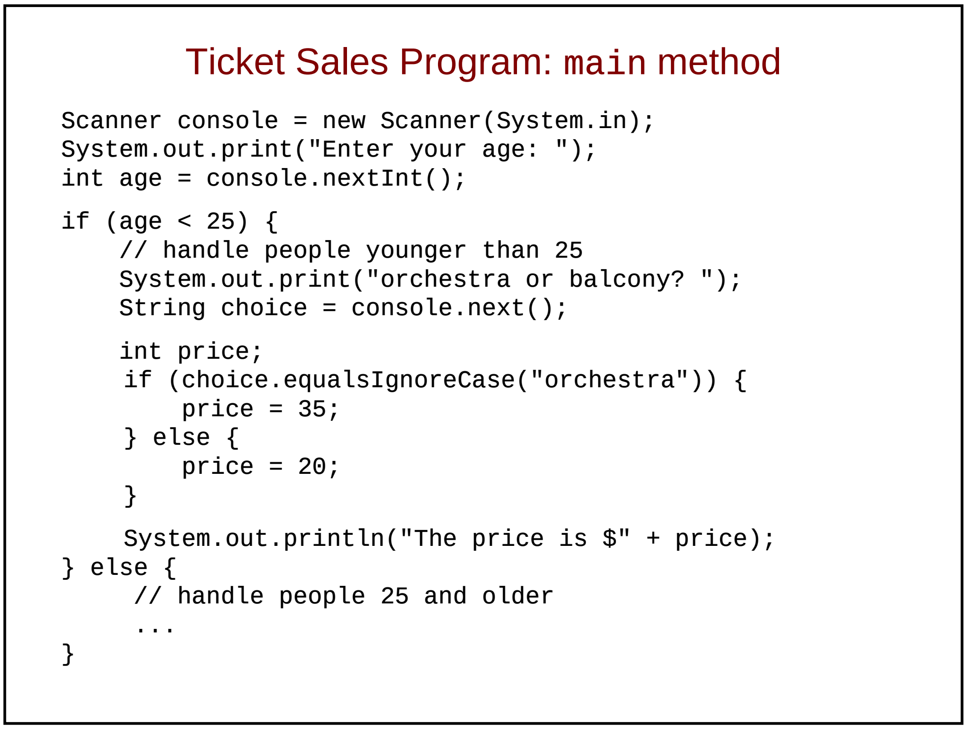
- The `String` class has a method called `equalsIgnoreCase` that can be used for this purpose:

```
if (choice.equalsIgnoreCase("regular")) {
    ...
}
```



Example Problem: Ticket Sales

- Different prices for balcony seats and orchestra seats
- Here are the rules:
  - persons younger than 25 receive discounted prices:
    - $20 for balcony seats
    - $35 for orchestra seats
  - everyone else pays the regular prices:
    - $30 for balcony seats
    - $50 for orchestra seats
- Assume only valid inputs.



Ticket Sales Program: `main` method

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();

if (age < 25) {
    // handle people younger than 25
    System.out.print("orchestra or balcony? ");
    String choice = console.next();

    int price;
    if (choice.equalsIgnoreCase("orchestra")) {
        price = 35;
    } else {
        price = 20;
    }

    System.out.println("The price is $" + price);
} else {
    // handle people 25 and older
    ...
}
```

Ticket Sales Program: `main` method (cont.)

```
...

} else {
     // handle people 25 and older
    System.out.print("orchestra or balcony? ");
    String choice = console.next();

    int price;
    if (choice.equalsIgnoreCase("orchestra")) {
        price = 50;
    } else {
        price = 30;
    }

    System.out.println("The price is $" + price);
}
```

Where Is the Code Duplication?

```
...
if (age < 25) {
    System.out.print("orchestra or balcony? ");
    String choice = console.next();

    int price;
    if (choice.equalsIgnoreCase("orchestra")) {
        price = 35;
    } else {
        price = 20;
    }

    System.out.println("The price is $" + price);
} else {
    System.out.print("orchestra or balcony? ");
    String choice = console.next();

    int price;
    if (choice.equalsIgnoreCase("orchestra")) {
        price = 50;
    } else {
        price = 30;
    }

    System.out.println("The price is $" + price);
}
```

Factoring Out Code Common to Multiple Cases

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();

System.out.print("orchestra or balcony? ");
String choice = console.next();

if (age < 25) {
  int price;
  if (choice.equalsIgnoreCase("orchestra")) {
    price = 35;
  } else {
    price = 20;
  }
} else {
  int price;
  if (choice.equalsIgnoreCase("orchestra")) {
    price = 50;
  } else {
    price = 30;
  }
}

System.out.println("The price is $" + price);
```

What Other Change Is Needed?

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();

System.out.print("orchestra or balcony? ");
String choice = console.next();

if (age < 25) {
  int price;
  if (choice.equalsIgnoreCase("orchestra")) {
    price = 35;
  } else {
    price = 20;
  }
} else {
  int price;
  if (choice.equalsIgnoreCase("orchestra")) {
    price = 50;
  } else {
    price = 30;
  }
}

System.out.println("The price is $" + price);
```

Now Let's Make It Structured

```
public static void main(String[] args) {
    …
    int age = console.nextInt();

    System.out.print("orchestra or balcony? ");
    String choice = console.next();
    int price;

    if (age < 25) {

        ____________________________________________________;
    } else {
       …
    }

    System.out.println("The price is $" + price);
}
public static ________ discountPrice(___________________) {





}
```

Expanded Ticket Sales Problem

- One additional case:
  - **persons younger than 13 cannot buy a ticket**
  - persons whose age is **13-24** receive discounted prices:
    - $20 for balcony seats
    - $35 for orchestra seats
  - everyone else pays the regular prices:
    - $30 for balcony seats
    - $50 for orchestra seats

Here's the Unfactored Version

```
...
if (age < 13) {
    System.out.println("You cannot buy a ticket.");
} else if (age < 25) {
    System.out.print("orchestra or balcony? ");
    String choice = console.next();

    int price;
    if (choice.equalsIgnoreCase("orchestra")) {
        price = 35;
    } else {
        price = 20;
    }

    System.out.println("The price is $" + price);
} else {
    System.out.print("orchestra or balcony? ");
    String choice = console.next();

    int price;
    if (choice.equalsIgnoreCase("orchestra")) {
        price = 50;
    } else {
        price = 30;
    }

    System.out.println("The price is $" + price);
}
```

We now have code common to the 2nd and 3rd cases, but not the 1st.

Group the Second and Third Cases Together

```
...
if (age < 13) {
    System.out.println("You cannot buy a ticket.");
} else {
     if (age < 25) {
        System.out.print("orchestra or balcony? ");
        String choice = console.next();

        int price;
        if (choice.equalsIgnoreCase("orchestra")) {
            price = 35;
        } else {
            price = 20;
        }

        System.out.println("The price is $" + price);
     } else {
        System.out.print("orchestra or balcony? ");
        String choice = console.next();

        ...

        System.out.println("The price is $" + price);
    }
}
```

Then Factor Out the Common Code

```
...
if (age < 13) {
     System.out.println("You cannot buy a ticket.");
} else {
     System.out.print("orchestra or balcony? ");
     String choice = console.next();

     int price;
     if (age < 25) {
         if (choice.equalsIgnoreCase("orchestra")) {
             price = 35;
         } else {
             price = 20;
         }
     } else {
         if (choice.equalsIgnoreCase("orchestra")) {
             price = 50;
         } else {
             price = 30;
         }
     }

     System.out.println("The price is $" + price);
}
```

Case Study: Coffee Shop Price Calculator

- Relevant info:
  - brewed coffee prices by size:
    - tiny: $1.60
    - medio: $1.80
    - gigundo: $2.00
  - latte prices by size:
    - tiny: $2.80
    - medio: $3.20
    - gigundo: $3.60

    plus, add 50 cents for a latte with flavored syrup
  - sales tax:
    - students: no tax
    - non-students: 6.25% tax

Case Study: Coffee Shop Price Calculator (cont.)

- Developing a solution:
  1. Begin with an *unstructured* solution.
     - everything in the `main` method
     - use if-else-if statement(s) to handle the various cases
  2. Next, *factor out* code that is common to multiple cases.
     - put it either before or after the appropriate if-else-if statement
  3. Finally, create a fully *structured* solution.
     - use procedural decomposition to capture logical pieces of the solution

Case Study: Coffee Shop Price Calculator (cont.)

Optional: Comparing Floating-Point Values

- Because the floating-point types have limited precision, it's possible to end up with *roundoff errors*.
- Example:

```
double sum = 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
sum = sum  +  0.1 + 0.1 + 0.1 + 0.1 + 0.1;
System.out.println(sum);
// get 0.9999999999999999!
```

- Thus when trying to determine if two floating-point values are equal, we usually do *not* use the == operator.
- Instead, we test if the difference between the two values is less than some small *threshold* value:

threshold

```
if (Math.abs(sum – 1.0) < 0.0000001) {
    System.out.println(sum + " == 1.0");
}
```

Optional: Another Cumulative Computation

- The same pattern can be used for other types of computations.
- Example: counting the occurrences of a character in a string.
- Let's write a static method called `numOccur` that does this.
    - examples:

      `numOccur('l', "hello")` should return `2`
      `numOccur('s', "Mississippi")` should return `4`

```
public static ___ numOccur(_____________________) {



}
```

Unit 3, Part 4

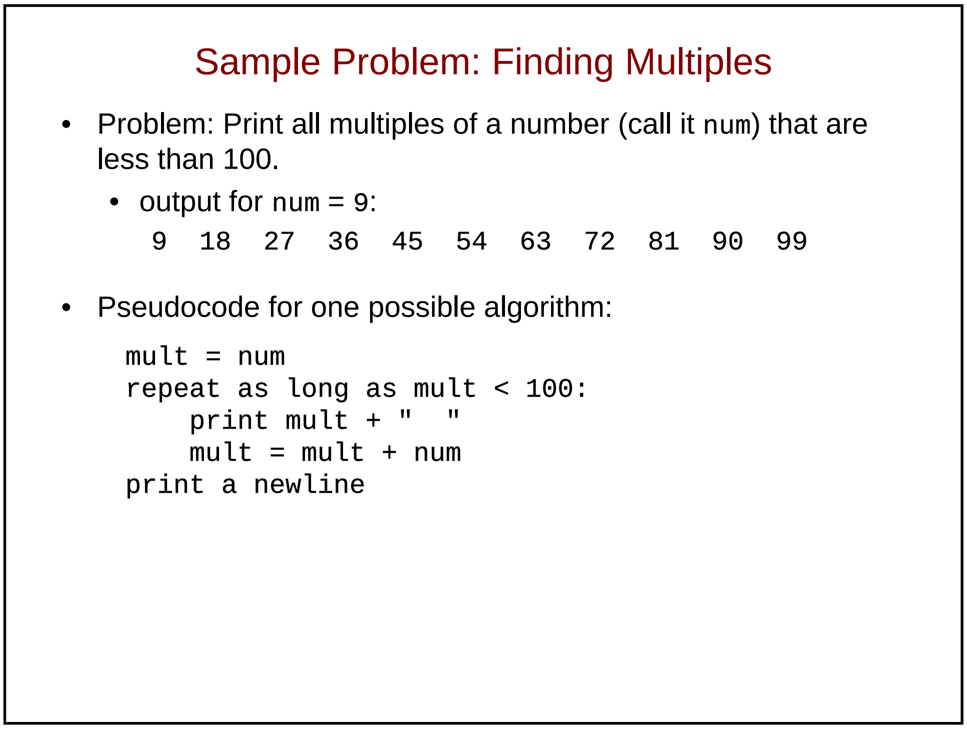
Indefinite Loops and Boolean Expressions

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Definite Loops

- The loops that we've seen thus far have been *definite loops*.
  - we know exactly how many iterations will be performed before the loop even begins
- In an *indefinite loop*, the number of iterations is either:
  - not as obvious
  - impossible to determine before the loop begins



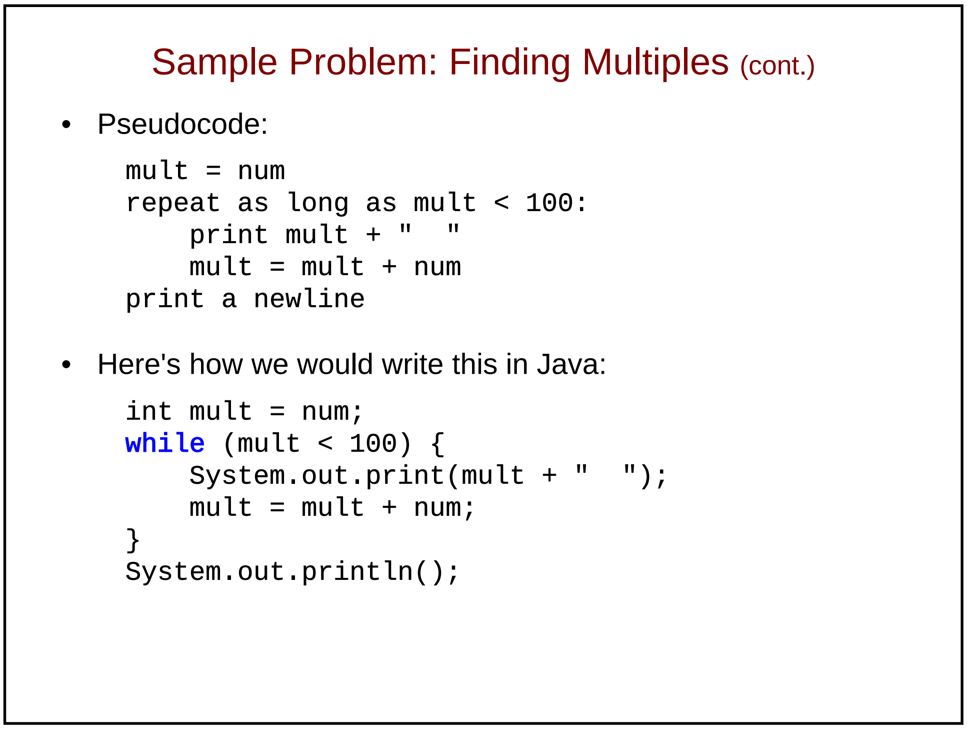
Sample Problem: Finding Multiples

- Problem: Print all multiples of a number (call it `num`) that are less than 100.
  - output for `num` = 9:

```
9   18   27   36   45   54   63   72   81   90   99
```

- Pseudocode for one possible algorithm:

```
mult = num
repeat as long as mult < 100:
    print mult + "  "
    mult = mult + num
print a newline
```



Sample Problem: Finding Multiples (cont.)

- Pseudocode:

```
mult = num
repeat as long as mult < 100:
    print mult + "  "
    mult = mult + num
print a newline
```

- Here's how we would write this in Java:

```
int mult = num;
while (mult < 100) {
    System.out.print(mult + "  ");
    mult = mult + num;
}
System.out.println();
```

while Loops

- In general, a `while` loop has the form

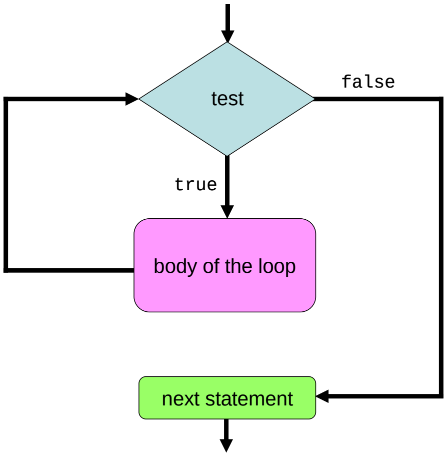
```
while (<test>) {
    <one or more statements>
}
```

- As with `for` loops, the statements in the block of a `while` loop are known as the *body* of the loop.

Evaluating a while Loop

Steps:

1. evaluate the test
2. if it's false, skip the statements in the body
3. if it's true, execute the statements in the body, and go back to step 1



Tracing a `while` Loop

- Let's trace through our code when `num` has the value `15`:

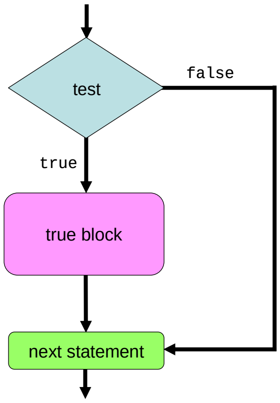
```
int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    mult = mult + num;
}
```

	output thus far	`mult`
before entering the loop		`15`
after the first iteration	`15`	`30`
after the second iteration	`15 30`	`45`
after the third iteration	`15 30 45`	`60`
after the fourth iteration	`15 30 45 60`	`75`
after the fifth iteration	`15 30 45 60 75`	`90`
after the sixth iteration	`15 30 45 60 75 90`	`105`

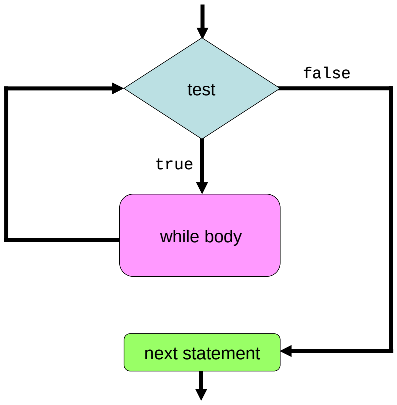
and now `(mult < 100)` is `false`, so we exit the loop

Comparing `if` and `while`

if statement



while statement



- The true block of an if statement is evaluated at most once.
- The body of a while statement can be evaluated multiple times, provided the test remains true.

Typical `while` Loop Structure

- Typical structure:

```
initialization statement(s)
while (test) {
    other statements
    update statement(s)
}
```

- In our example:

```
int mult = num;                     // initialization
while (mult < 100) {
    System.out.print(mult + "  ");
    mult = mult + num;              // update
}
```

Comparing `for` and `while` loops

- `while` loop (typical structure):

```
initialization
while (test) {
    other statements
    update
}
```

- `for` loop:

```
for (initialization; test; update) {
     one or more statements
}
```

Infinite Loops

- Let's say that we change the condition for our `while` loop:

```
int mult = num;
while (mult != 100) {     // replaced < with !=
    System.out.print(mult + " ");
    mult = mult + num;
}
```

- When `num` is 15, the condition will always be true.
  - why?
  - an *infinite loop* – the program will hang (or repeatedly output something), and needs to be stopped manually
  - what class of error is this (syntax or logic)?
- It's generally better to use <, <=, >, >= in a loop condition, rather than == or `!=`

Infinite Loops (cont.)

- Another common source of infinite loops is forgetting the update statement:

```
int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    // update should go here
}
```

A Need for Error-Checking

- Let's return to our original version:

```
int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    mult = mult + num;
}
```

- This could still end up in an infinite loop! How?

Using a Loop When Error-Checking

- We need to check that the user enters a positive integer.
- If the number is <= 0, ask the user to try again.
- Here's one way of doing it using a `while` loop:

```
Scanner console = new Scanner(System.in);
System.out.print("Enter a positive integer: ");
int num = console.nextInt();
while (num <= 0) {
    System.out.print("Enter a positive integer: ");
    num = console.nextInt();
}
```

- Note that we end up duplicating code.

Error-Checking Using a `do-while` Loop

- Java has a second type of loop statement that allows us to eliminate the duplicated code in this case:

```
Scanner console = new Scanner(System.in);
int num;
do {
    System.out.print("Enter a positive integer: ");
    num = console.nextInt();
} while (num <= 0);
```

- The code in the body of a `do-while` loop is always executed at least once.

`do-while` Loops

- In general, a `do-while` statement has the form

```
do {
    <one or more statements>
} while (<test>);
```

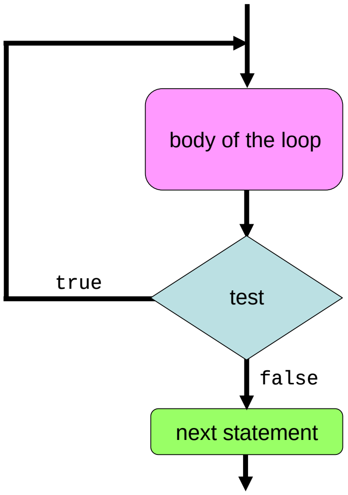
- Note the need for a semi-colon after the condition.
- We do *not* need a semi-colon after the condition in a `while` loop.
  - beware of using one – it can actually create an infinite loop!

Evaluating a `do-while` Loop

<u>Steps:</u>

1. execute the statements in the body
2. evaluate the test
3. if it's true, go back to step 1

(if it's false, continue to the next statement)



Formulating Loop Conditions

- We often need to repeat actions *until* a condition is met.
  - example: keep reading a value *until* the value is positive
  - such conditions are *termination* conditions – they indicate when the repetition should stop
- However, loops in Java repeat actions *while* a condition is met.
  - they use *continuation* conditions
- As a result, you may need to convert a termination condition into a continuation condition.

Which Type of Loop Should You Use?

- Use a `for` loop when the number of repetitions is known in advance – i.e., for a definite loop.
- Otherwise, use a `while` loop or `do-while` loop:
    - use a `while` loop if the body of the loop may not be executed at all
        - i.e., if the condition may be `false` at the start of the loop
    - use a `do-while` loop if:
        - the body will always be executed at least once
        - doing so will allow you to avoid duplicating code

Find the Error…

- Where is the syntax error below?

```
Scanner console = new Scanner(System.in);

do {
    System.out.print("Enter a positive integer: ");
    int num = console.nextInt();
} while (num <= 0);

System.out.println("\nThe multiples of " + num +
  " less than 100 are:");

int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    mult = mult + num;
}

System.out.println();
```

Practice with `while` loops

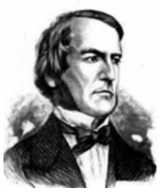
- What does the following loop output?

```
int a = 10;
while (a > 2) {
    a = a – 2;
    System.out.println(a * 2);
}
```

	a > 2	a	output
before loop			
1st iteration			
2nd iteration			
3rd iteration			
4th iteration			

`boolean` Data Type

- A condition like `mult < 100` has one of two values: `true` or `false`
- In Java, these two values are represented using the `boolean` data type.
    - one of the primitive data types (like `int`, `double`, and `char`)
    - `true` and `false` are its two literal values
- This type is named after the 19th-century mathematician George Boole, who developed the system of logic called *boolean algebra*.



`boolean` Expressions

- We have seen a number of constructs that use a "test".
  - loops
  - if statements

- A more precise term for a "test" is a *boolean expression*.

- A boolean expression is any expression that evaluates to `true` or `false`.
  - examples:

```
num > 0
false
firstChar == 'P'
score != 20
```

`boolean` Expressions (cont.)

- Recall this line from our ticket-price program:

```
if (choice.equals("orchestra")) …
```

a boolean expression, because it evaluates to `true` *or* `false`

  - if we look at the `String` class in the Java API, we see that the `equals` method has this header:

```
public boolean equals(...)
```

it returns either `true` *or* `false`

Forming More Complex Conditions

- We often need to make a decision based on more than one condition – or based on the opposite of a condition.
  - examples in pseudocode:

    if the number is even AND it is greater than 100…

    if it is NOT the case that your grade is > 80…

- Java provides three *logical operators* for this purpose:

operator	name	example
`&&`	and	`age >= 18 && age <= 35`
`||`	or	`age < 3 || age > 65`
`!`	not	`!(grade > 80)`

Truth Tables

- The logical operators operate on boolean expressions.
  - let `a` and `b` represent two such expressions
- We can define the logical operators using *truth tables*.

truth table for && (and)

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

truth table for || (or)

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

truth table for ! (not)

a	!a
false	true
true	false

Truth Tables (cont.)

- Example: evaluate the following expression:

```
(20 >= 0) && (30 % 4 == 1)
```

- First, evaluate each of the operands:

```
(20 >= 0) && (30 % 4 == 1)
   true   &&    false
```

- Then, consult the appropriate row of the truth table:

a	b	a && b
false	false	false
false	true	false
true	**false**	**false**
true	true	true

- Thus, `(20 >= 0) && (30 % 4 == 1)` evaluates to `false`

Practice with Boolean Expressions

- Let's say that we wanted to express the following English condition in Java:

  "num is not equal to either 0 or 1"

- Which of the following boolean expression(s) would work?

  a) `num != 0 || 1`

  b) `num != 0 || num != 1`

  c) `!(num == 0 || num == 1)`

- Is there a different boolean expression that would work here?

`boolean` Variables

- We can declare variables of type `boolean`, and assign the values of boolean expressions to them:

```
int num = 10;
boolean isPos = (num > 0);
boolean isDone = false;
```

  - these statements give us the following picture in memory:

```
isPos  | true |     isDone | false |
```

- Using a boolean variable can make your code more readable:

```
if (value % 2 == 0) {
    ...
```

⇩

```
boolean isEven = (value % 2 == 0);
if (isEven == true) {
    ...
```

`boolean` Variables (cont.)

- Instead of doing this:

```
boolean isEven = (num % 2 == 0);
if (isEven == true) {
    ...
```

  you could just do this:

```
boolean isEven = (num % 2 == 0);
if (isEven) {
    ...
```

  The extra comparison isn't necessary!

- Similarly, instead of writing:

```
if (isEven == false) {
    ...
```

  you could just write this:

```
if (!isEven) {
    ...
```

Input Using a Sentinel

- Example problem: averaging an arbitrary number of grades.
- Instead of having the user tell us the number of grades in advance, we can let the user indicate that there are no more grades by entering a special *sentinal value*.
- When we encounter the sentinel, we break out of the loop
  - example interaction:

```
Enter grade (-1 to end): 10
Enter grade (-1 to end): 8
Enter grade (-1 to end): 9
Enter grade (-1 to end): 5
Enter grade (-1 to end): -1
The average is: 8.0
```

Input Using a Sentinel (cont.)

- Here's one way to do this:

```
Scanner console = new Scanner(System.in);
int total = 0;
int numGrades = 0;

System.out.print("Enter grade (or -1 to quit): ");
int grade = console.nextInt();
while (grade != -1) {
    total += grade;
    numGrades++;
    System.out.print("Enter grade (or -1 to quit): ");
    grade = console.nextInt();
}

if (numGrades > 0) {
    System.out.print("The average is ");
    System.out.println((double)total/numGrades);
}
```

Input Using a Sentinel and a Boolean Flag

- Here's another way, using what is known as a *boolean flag*, which is a variable that keeps track of some condition:

```
Scanner console = new Scanner(System.in);
int total = 0;
int numGrades = 0;
boolean done = false;

while (!done) {
    System.out.print("Enter grade (or -1 to quit): ");
    int grade = console.nextInt();
    if (grade == -1) {
        done = true;
    } else {
        total += grade;
        numGrades++;
    }
}

if (numGrades > 0) {
    ...
```

Input Using a Sentinel and a `break` Statement

- Here's another way, using what is known as a `break` statement, which "breaks out" of the loop:

```
Scanner console = new Scanner(System.in);
int total = 0;
int numGrades = 0;

while (true) {
    System.out.print("Enter grade (or -1 to quit): ");
    int grade = console.nextInt();
    if (grade == -1) {
        break;
    }
    total += grade;
    numGrades++;
}

// after the break statement, the flow of control
// resumes here...
if (numGrades > 0) {
    ...
```

Unit 4, Part 1

Arrays

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Collections of Data

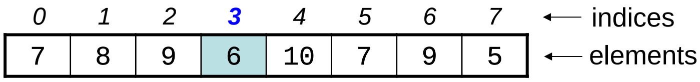
- Recall our program for averaging quiz grades:

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    int total = 0;
    int numGrades = 0;

    while (true) {
        System.out.print("Enter a grade (or -1 to quit): ");
        int grade = console.nextInt();
        if (grade == -1) {
            break;
        }
        total += grade;
        numGrades++;
    }

    if (numGrades > 0) {
        ...
}
```

- What if we wanted to store the individual grades?
    - an example of a *collection* of data

Arrays

- An *array* is a collection of data values of the same type.
- In the same way that we think of a variable as a single box, an array can be thought of as a sequence of boxes:



0	*1*	*2*	***3***	*4*	*5*	*6*	*7*	← indices
7	8	9	6	10	7	9	5	← elements

- Each box contains one of the data values in the collection
  - referred to as the *elements* of the array
- Each element has a numeric *index*
  - the first element has an index of 0,
    the second element has an index of 1,
    etc.
  - example: the value 6 above has an index of 3
  - like the index of a character in a `String`

Declaring and Creating an Array

- We use a variable to represent the array as a whole.
- Example of declaring an array variable:

```
int[] grades;
```

  - the `[]` indicates that it will represent an array
  - the `int` indicates that the elements will be `int`s
- Declaring the array variable does *not* create the array.
- Example of creating an array:

```
grades = new int[8];
```

the *length* of the array –
i.e., the number of elements

Declaring and Creating an Array (cont.)

- We often declare and create an array in the same statement:

```
int[] grades = new int[8];
```

- General syntax:

```
<type>[] <array> = new <type>[<length>];
```

  where

  <*type*> is the type of the individual elements
  <*array*> is the name of the variable used for the array
  <*length*> is the number of elements in the array

The Length of an Array

- The *length* of an array is the number of elements in the array.
- The length of an array can be obtained as follows:

```
<array>.length
```

  - example:

```
grades.length
```

  - note: it is *not* a method

    `grades.length()` won't work!

Auto-Initialization

- When you create an array in this way:

```
int[] grades = new int[8];
```

the runtime system gives the elements default values:

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

- The value used depends on the type of the elements:

`int`	`0`
`double`	`0.0`
`char`	`'\0'`
`boolean`	`false`
objects	`null`

Accessing an Array Element

- To access an array element, we use an expression of the form

<array>`[`*<index>*`]`

- Examples:

`grades[0]` accesses the first element
`grades[1]` accesses the second element
`grades[5]` accesses the sixth element

- Here's one way of setting up the array we showed earlier:

0	1	2	3	4	5	6	7
7	8	9	6	10	7	9	5

```
int[] grades = new int[8];
grades[0] = 7;  grades[1] = 8;  grades[2] = 9;
grades[3] = 6;  grades[4] = 10; grades[5] = 7;
grades[6] = 9;  grades[7] = 5;
```

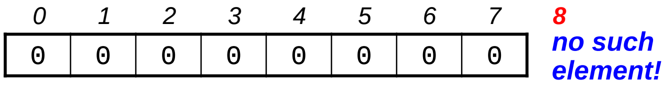
Accessing an Array Element (cont.)

- Acceptable index values:

  integers from 0 to <*array*>`.length – 1`

- If we specify an index outside that range, we'll get an `ArrayIndexOutOfBoundsException` at runtime.
  - example:

```
int[] grades = int[8];
grades[8] = 5;
```



0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	no such element!

Accessing an Array Element (cont.)

- The index can be any integer expression.
  - example:

```
int lastGrade = grades[grades.length – 1];
```

- We can operate on an array element in the same way that we operate on any other variable of that type.
  - example: applying a 10% late penalty to the grade at index `i`

```
grades[i] = (int)(grades[i] * 0.9);
```

  - example: adding 5 points of extra credit to the grade at index `i`

```
grades[i] += 5;
```

Another Way to Create an Array

- If we know that we want an array to contain specific values, we can specify them when create the array.

- Example: here's another way to create and initialize our `grades` array:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
```

- The list of values is known as an *initialization list*.
  - it can only be specified when the array is declared
  - we don't use the `new` operator in this case
  - we don't specify the length of the array – it is determined from the number of values in the initialization list

- Other examples:

```
double[] heights = {65.2, 72.0, 70.6, 67.9};
boolean[] isPassing = {true, true, false, true};
```

Storing Grades Entered by the User

- We need to know how big to make the array.
  - one way: ask the user for the maximum number of values

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);

    System.out.print("How many grades? ");
    int maxNumGrades = console.nextInt();
    int[] grades = new int[maxNumGrades];

    int total = 0;
    int numGrades = 0;

    while (numGrades < maxNumGrades) {
        System.out.print("Enter a grade (or -1 to quit): ");
        grades[numGrades] = console.nextInt();
        if (grades[numGrades] == -1) {
            break;
        }
        total += grades[numGrades];
        numGrades++;
    }
    ...
}
```

Processing the Values in an Array

- We often use a `for` loop to process the values in an array.
- Example: print out all of the grades

```
int[] grades = new int[maxNumGrades];
...
for (int i = 0; i < grades.length; i++) {
    System.out.println("grade " + i + ": " + grades[i]);
}
```

- General pattern:

```
for (int i = 0; i < <array>.length; i++) {
    do something with <array>[i];
}
```

- Processing array elements sequentially from first to last is known as *traversing* the array.
    - noun = *traversal*

Another Example of Traversing an Array

- Let's write code to find the highest quiz grade in the array:

```
int max = __________________;

for (_________; _________________; ______) {



}
```

Another Example of Traversing an Array (cont.)

grades array:

7	8	9	6	10	7	9	5

- Let's trace through our code:

```
int max = grades[0];
for (int i = 1; i < grades.length; i++) {
    if (grades[i] > max) {
        max = grades[i];
    }
}
```

```
i   grades[i]   max
                7
1   8           8
2   9           9
3   6           9
4   10          10
5   7           10
...
```

Review: What Is a Variable?

- We've seen that a variable is like a named "box" in memory that can be used to store a value.

```
int count = 10;            count | 10 |
```

- If a variable represents a primitive-type value, the value is stored in the variable itself, as shown above.

Reference Variables

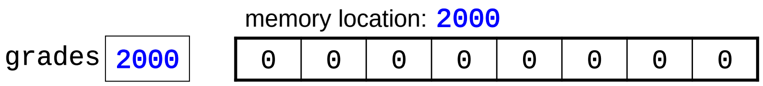
- If a variable represents an object, the object itself is *not* stored inside the variable.
- Rather, the object is located somewhere else in memory, and the variable holds the *memory address* of the object.
  - we say that the variable stores a *reference* to the object
  - such variables are called *reference variables*

Arrays and References

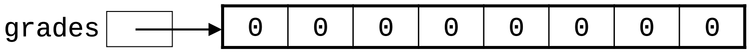
- An array is a type of object.
- Thus, an array variable is a reference variable.
  - it stores a reference to the array
- Example:

```
int[] grades = new int[8];
```

  might give the following picture:



- We usually use an arrow to represent a reference:



Printing an Array

- What is the output of the following lines?

```
    int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
    System.out.println(grades);
```

- To print the contents of the array, we can use a `for` loop as we showed earlier.

- We can also use the `Arrays.toString()` method, which is part of Java's built in `Arrays` class.

```
    int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
    System.out.println(Arrays.toString(grades));
```

  - doing so produces the following output:

```
    [7, 8, 9, 6, 10, 7, 9, 5]
```

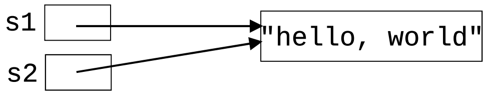
- To use this method, we need to import the `java.util` package.

Copying References

- When we assign the value of one reference variable to another, we copy the reference to the object.
  We do *not* copy the object itself.

- Example involving objects:

```
  String s1 = "hello, world";
  String s2 = s1;
```



s1

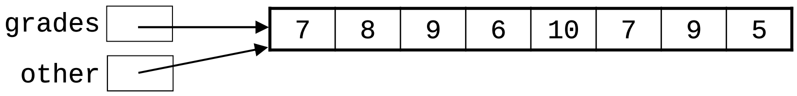
s2

"hello, world"

Copying References (cont.)

- An example involving an array:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
int[] other = grades;
```



grades

other

7	8	9	6	10	7	9	5

- Given the lines of code above, what will the lines below print?

```
other[2] = 4;
System.out.println(grades[2] + " " + other[2]);
```

Null References

- To indicate that a reference variable doesn't yet refer to any object, we can assign it a special value called `null`.

```
int[] grades = null;
String s = null;
```

grades null

s null

- Attempting to use a null reference to access an object produces a `NullPointerException`.
    - "pointer" is another name for reference
    - example:

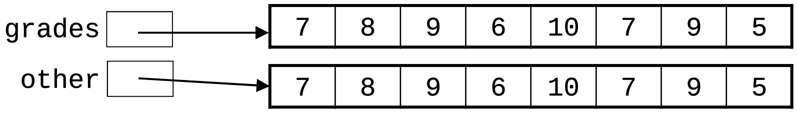
```
int[] grades = null;
grades[3] = 10;        // NullPointerException!
char ch = s.charAt(5); // NullPointerException!
```

Copying an Array

- To actually create a copy of an array, we can:
  - create a new array of the same length as the first
  - traverse the arrays and copy the individual elements

- Example:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
int[] other = new int[grades.length];
for (int i = 0; i < grades.length; i++) {
    other[i] = grades[i];
}
```



- What do the following lines print now?

```
other[2] = 4;
System.out.println(grades[2] + " " + other[2]);
```

Programming Style Point

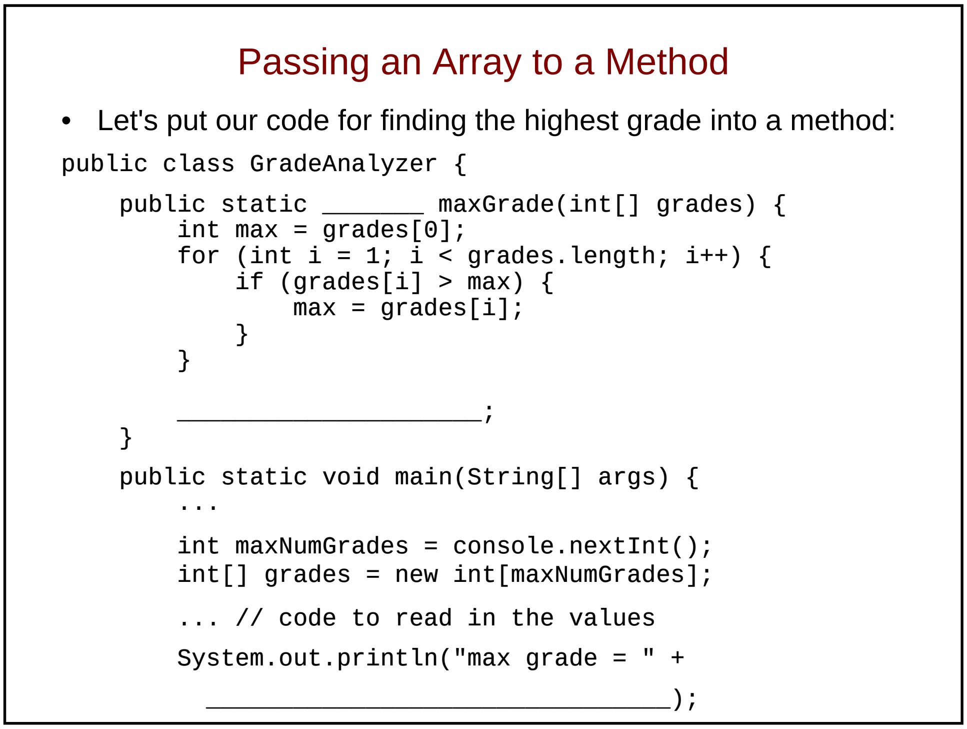
- Here's how we copied the array:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
int[] other = new int[grades.length];
for (int i = 0; i < grades.length; i++) {
    other[i] = grades[i];
}
```

- This would also work:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
int[] other = new int[8];
for (int i = 0; i < 8; i++) {
    other[i] = grades[i];
}
```

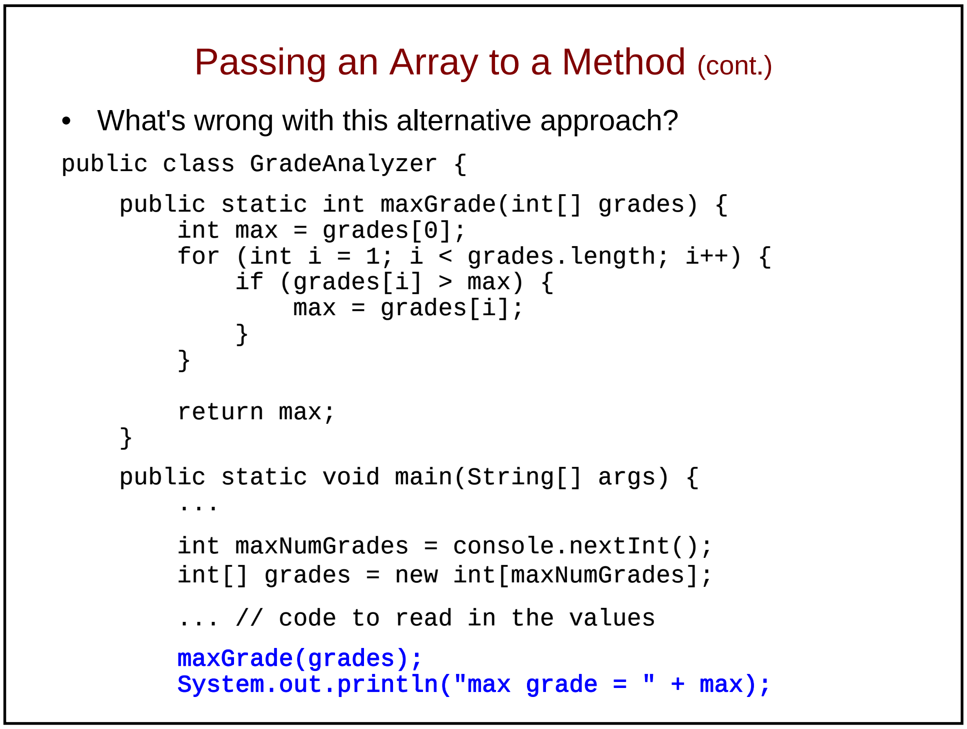
- Why is the first way better?



Passing an Array to a Method

- Let's put our code for finding the highest grade into a method:

```
public class GradeAnalyzer {
    public static _______ maxGrade(int[] grades) {
        int max = grades[0];
        for (int i = 1; i < grades.length; i++) {
            if (grades[i] > max) {
                max = grades[i];
            }
        }

        _____________________;
    }
    public static void main(String[] args) {
        ...
        int maxNumGrades = console.nextInt();
        int[] grades = new int[maxNumGrades];
        ... // code to read in the values
        System.out.println("max grade = " +
          ________________________________);
```



Passing an Array to a Method (cont.)

- What's wrong with this alternative approach?

```
public class GradeAnalyzer {
    public static int maxGrade(int[] grades) {
        int max = grades[0];
        for (int i = 1; i < grades.length; i++) {
            if (grades[i] > max) {
                max = grades[i];
            }
        }

        return max;
    }
    public static void main(String[] args) {
        ...
        int maxNumGrades = console.nextInt();
        int[] grades = new int[maxNumGrades];
        ... // code to read in the values
        maxGrade(grades);
        System.out.println("max grade = " + max);
```

Passing an Array to a Method (cont.)

- We could do this instead:

```
public class GradeAnalyzer {

    public static int maxGrade(int[] grades) {
        int max = grades[0];
        for (int i = 1; i < grades.length; i++) {
            if (grades[i] > max) {
                max = grades[i];
            }
        }

        return max;
    }

    public static void main(String[] args) {
        ...
        int maxNumGrades = console.nextInt();
        int[] grades = new int[maxNumGrades];

        ... // code to read in the values

        int max = maxGrade(grades);
        System.out.println("max grade = " + max);
```

Finding the Average Value in an Array

- Here's a method that computes the average grade:

```
public static double averageGrade(int[] grades) {
    int total = 0;
    for (int i = 0; i < grades.length; i++) {
        total += grades[i];
    }

    return (double)total / grades.length;
}
```

Testing If An Array Meets Some Condition

- Let's say that we need to be able to determine if there are any grades below a certain cutoff value.
  - e.g., to determine if a retest should be given
- Does this method work?

```
public static boolean
anyGradesBelow(int[] grades, int cutoff) {
    for (int i = 0; i < grades.length; i++) {
        if (grades[i] < cutoff) {
            return true;
        } else {
            return false;
        }
    }
}
```

Testing If An Array Meets Some Condition (cont.)

- We can return `true` as soon as we find a grade that is below the threshold.
- We can only return `false` if *none* of the grades is below.
- Here is a corrected version:

```
public static boolean
anyGradesBelow(int[] grades, int cutoff) {
    for (int i = 0; i < grades.length; i++) {
        if (grades[i] < cutoff) {
            return true;
        }
    }

    // if we get here, none of the grades is below.
    return false;
}
```

Testing If An Array Meets Some Condition (cont.)

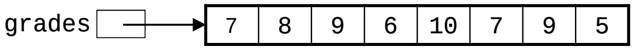
- Here's a similar problem: write a method that determines if all of the grades are perfect (assume perfect = 100).

```
public static boolean allPerfect(int[] grades) {









}
```

Using an Array to Count Things

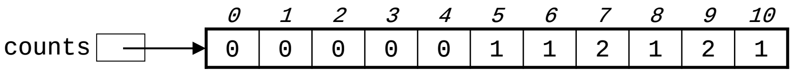
- Let's say that we want to count how many times each of the possible grade values appears in a collection of grades.

- We can use an array to store the counts.
  - `counts[i]` will store the number of times that the grade `i` appears
  - for this grades array

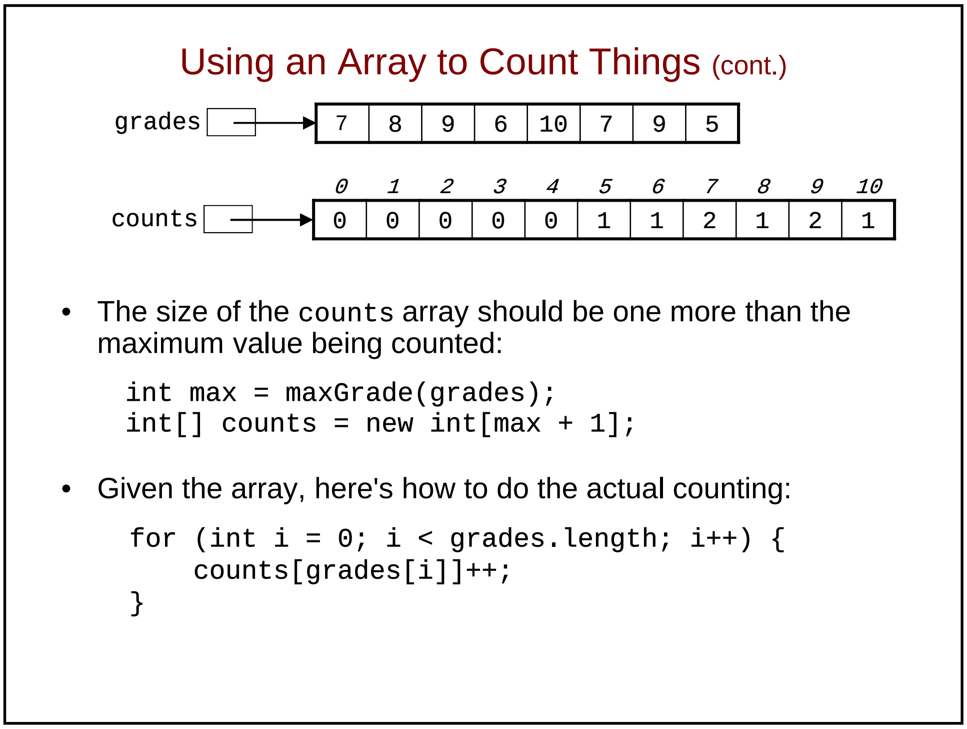


grades	7	8	9	6	10	7	9	5

we would have this array of counts:



	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
counts	0	0	0	0	0	1	1	2	1	2	1



Using an Array to Count Things (cont.)

grades →

7	8	9	6	10	7	9	5

counts →

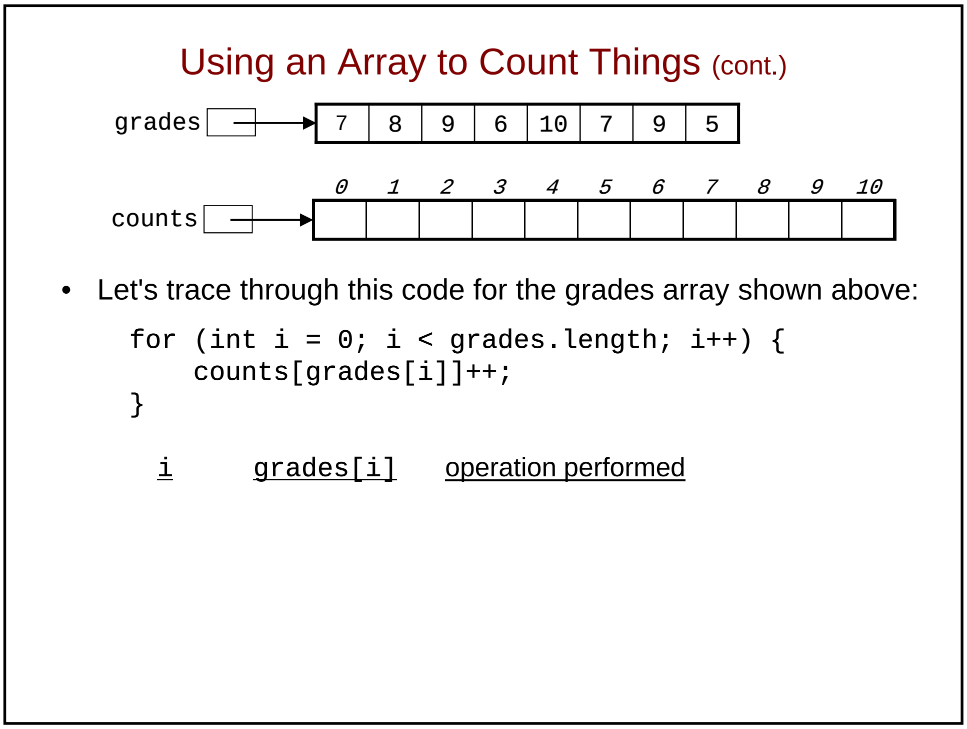
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
0	0	0	0	0	1	1	2	1	2	1

- The size of the `counts` array should be one more than the maximum value being counted:

```
int max = maxGrade(grades);
int[] counts = new int[max + 1];
```

- Given the array, here's how to do the actual counting:

```
for (int i = 0; i < grades.length; i++) {
    counts[grades[i]]++;
}
```



Using an Array to Count Things (cont.)

grades →

7	8	9	6	10	7	9	5

counts →

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

- Let's trace through this code for the grades array shown above:

```
for (int i = 0; i < grades.length; i++) {
    counts[grades[i]]++;
}
```

`i`	`grades[i]`	operation performed

A Method That Returns an Array

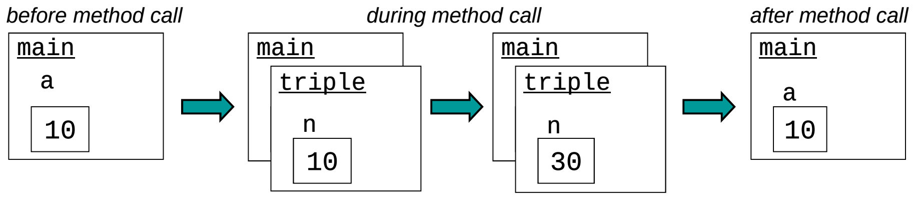
- We can write a method to create and return the array of counts:

```
public static int[] getCounts(int[] grades, int maxGrade) {
    int[] counts = new int[maxGrade + 1];
    for (int i = 0; i < grades.length; i++) {
        counts[grades[i]]++;
    }
    return counts;
}

public static void main(String[] args) {
    ... // main method begins as in the earlier versions
    int max = maxGrade(grades);
    int[] counts = getCounts(grades, max);
    ...
}
```

Review: Methods with Parameters

- A method cannot change its actual parameters, because the formal params are *copies* of the actual params.

```
public static void main(String[] args) {
    int a = 10;
    triple(a);
    System.out.println(a);
}
public static void triple(int n) {
    n *= 3;
}
```



Review: Methods with Parameters (cont.)

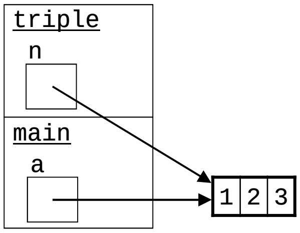
- In order for a method to change the value of an actual parameter, we need to do the following:
  - make the method return a value
  - assign the return value back to the variable used for the actual parameter

```
public static void main(String[] args) {
    int a = 10;
    a = triple(a);
    System.out.println(a);
}

public static int triple(int n) {
    n *= 3;
    return n;
}
```

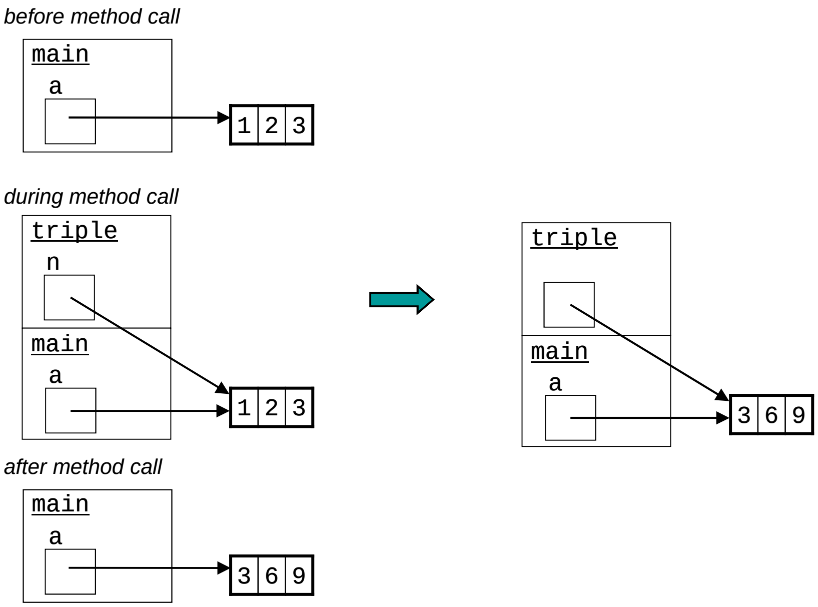
Using a Method to Change an Array

```
public static void main(String[] args) {
    int[] a = {1, 2, 3};
    triple(a);
    System.out.println(Arrays.toString(a));
}

public static void triple(int[] n) {
    for (int i = 0; i < n.length; i++) {
        n[i] = n[i] * 3;
    }
}
```

- When a method is passed an array as a parameter, it gets a reference to the same array.
- Thus, it can change the contents of the array.

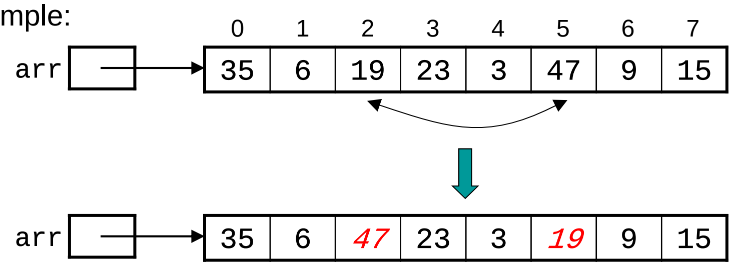


Using a Method to Change an Array (cont.)



Swapping Elements in an Array

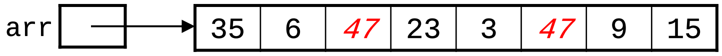
- We sometimes need to be able to swap two elements in an array.
- Example:



- What's wrong with this code for swapping the two values?

```
arr[2] = arr[5];
arr[5] = arr[2];
```

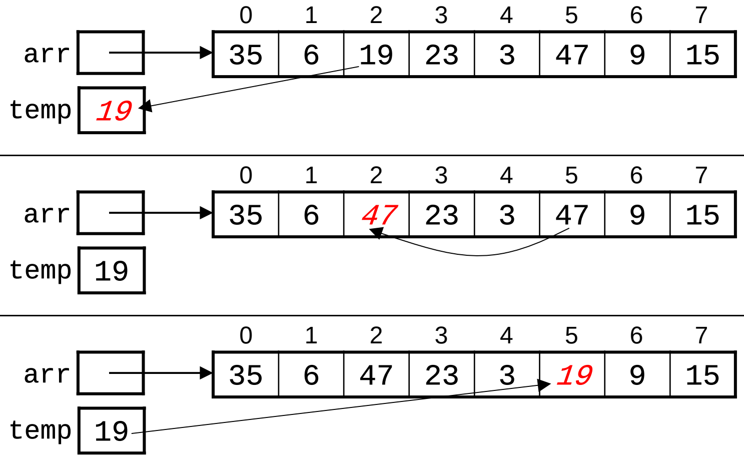
  - it gives this:



Swapping Elements in an Array (cont.)

- To perform a swap, we need to use a temporary variable:

```
int temp = arr[2];
arr[2] = arr[5];
arr[5] = temp;
```



arr

0	1	2	3	4	5	6	7
35	6	19	23	3	47	9	15

temp *19*

arr

0	1	2	3	4	5	6	7
35	6	*47*	23	3	47	9	15

temp 19

arr

0	1	2	3	4	5	6	7
35	6	47	23	3	*19*	9	15

temp 19

A Method for Swapping Elements

- Here's a method for swapping the elements at positions `i` and `j` in the array `arr`:

```
public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

- We don't need to return anything, because the method changes the array that is passed in.

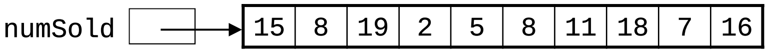
- Here's an example of how we would use it:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
swap(grades, 2, 5);
System.out.println(Arrays.toString(grades));
```

- What would the output be?

Shifting Values in an Array

- Let's say a small business is using an array to store the number of items sold over a 10-day period.

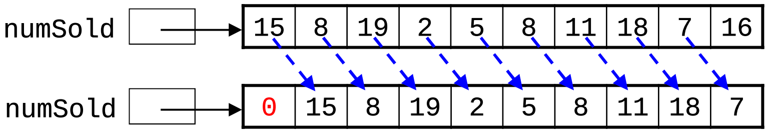


15	8	19	2	5	8	11	18	7	16

```
numSold[0]  gives the number of items sold today
numSold[1]  gives the number of items sold 1 day ago
numSold[2]  gives the number of items sold 2 days ago
…
numSold[9]  gives the number of items sold 9 days ago
```

Shifting Values in an Array (cont.)

- At the start of each day, it's necessary to shift the values over to make room for the new day's sales.



15	8	19	2	5	8	11	18	7	16
0	15	8	19	2	5	8	11	18	7

  - the last value is lost, since it's now 10 days old

- In order to shift the values over, we need to perform assignments like the following:

```
numSold[9] = numSold[8];
numSold[6] = numSold[5];
numSold[2] = numSold[1];
```

  - what is the general form (the pattern) of these assignments?

Shifting Values in an Array (cont.)

- Here's one attempt at code for shifting all of the elements:

```
for (int i = 0; i < numSold.length; i++) {
    numSold[i] = numSold[i - 1];
}
```

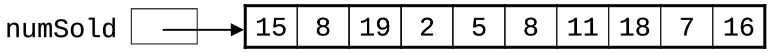
- If we run this, we get an `ArrayIndexOutOfBoundsException`. Why?

Shifting Values in an Array (cont.)

- This version of the code eliminates the exception:

```
for (int i = 1; i < numSold.length; i++) {
    numSold[i] = numSold[i – 1];
}
```

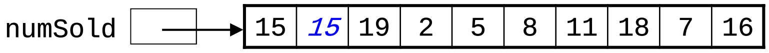
- Let's trace it to see what it does:



numSold

15	8	19	2	5	8	11	18	7	16

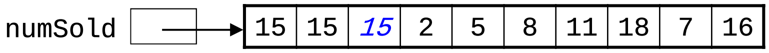
  - when `i == 1`, we perform `numSold[1] = numSold[0]` to get:



numSold

15	*15*	19	2	5	8	11	18	7	16

  - when `i == 2`, we perform `numSold[2] = numSold[1]` to get:



numSold

15	15	*15*	2	5	8	11	18	7	16

this obviously doesn't work!

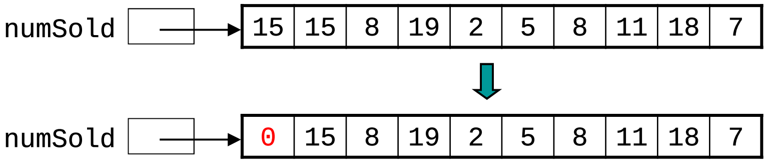
Shifting Values in an Array (cont.)

- How can we fix this code so that it does the right thing?

```
for (int i = 1; i < numSold.length; i++) {
    numSold[i] = numSold[i – 1];
}
```

```
for (              ;                ;         ) {



}
```

- After performing all of the shifts, we would do: `numSold[0] = 0;`



```
numSold  → | 15 | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 |

numSold  → | 0 | 15 | 8 | 19 | 2 | 5 | 8 | 11 | 18 | 7 |
```

"Growing" an Array

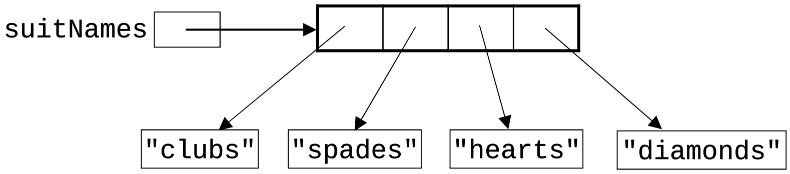
- Once we have created an array, we can't increase its size.
- Instead, we need to do the following:
  - create a new, larger array (use a temporary variable)
  - copy the contents of the original array into the new array
  - assign the new array to the original array variable
- Example for our `grades` array:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
...
int[] temp = new int[16];
for (int i = 0; i < grades.length; i++) {
     temp[i] = grades[i];
}
grades = temp;
```

Arrays of Objects

- We can use an array to represent a collection of objects.
- In such cases, the cells of the array store references to the objects.
- Example:

```
String[] suitNames = {"clubs", "spades",
  "hearts", "diamonds"};
```



suitNames

"clubs" "spades" "hearts" "diamonds"

Two-Dimensional Arrays

- Thus far, we've been looking at single-dimensional arrays
- We can also create *multi-dimensional* arrays.
- The most common type is a two-dimensional (2-D) array.
- We can visualize it as a matrix consisting of rows and columns:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	15	8	3	16	12	7	9	5
1	6	11	9	4	1	5	8	13
2	17	3	5	18	10	6	7	21
3	8	14	13	6	13	12	8	4
4	1	9	5	16	20	2	3	9

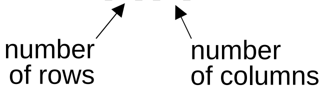
column indices

row indices

2-D Array Basics

- Example of declaring and creating a 2-D array:

```
int[][] scores = new int[5][8];
```



- To access an element, we use an expression of the form

  <array>`[`*<row>*`][`*<column>*`]`

  - example: `scores[3][4]` gives the score at row 3, column 4

	0	1	2	3	4	5	6	7
0	15	8	3	16	12	7	9	5
1	6	11	9	4	1	5	8	13
2	17	3	5	18	10	6	7	21
3	8	14	13	6	13	12	8	4
4	1	9	5	16	20	2	3	9

Example Application: Maintaining a Game Board

- For a Tic-Tac-Toe board, we could use a 2-D array to keep track of the state of the board:

```
char[][] board = new char[3][3];
```

- Alternatively, we could create *and* initialize it as follows:

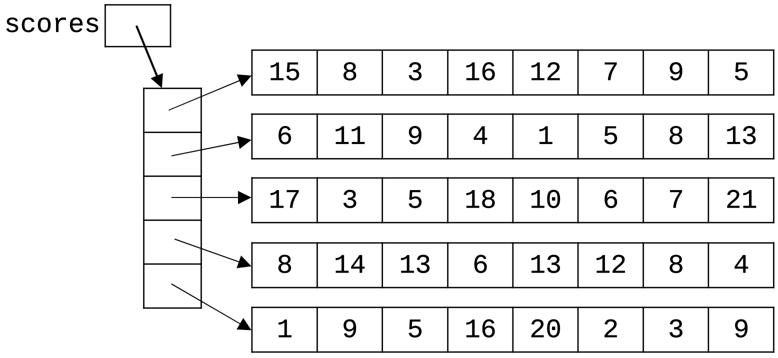
```
char[][] board = {{' ', ' ', ' '},
                  {' ', ' ', ' '},
                  {' ', ' ', ' '}};
```

- If a player puts an X in the middle square, we could record this fact by making the following assignment:

```
board[1][1] = 'X';
```

An Array of Arrays

- A 2-D array is really an array of arrays!



15	8	3	16	12	7	9	5
6	11	9	4	1	5	8	13
17	3	5	18	10	6	7	21
8	14	13	6	13	12	8	4
1	9	5	16	20	2	3	9

- `scores[0]` represents the entire first row
  `scores[1]` represents the entire second row, etc.
- <*array*>`.length` gives the number of rows
  <*array*>[<*row*>]`.length` gives the number of columns in that row

Processing All of the Elements in a 2-D Array

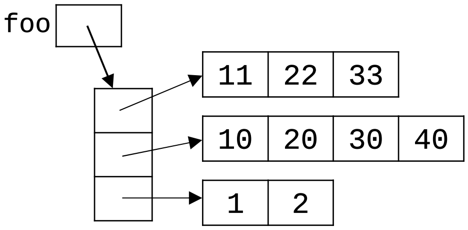
- To perform some operation on all of the elements in a 2-D array, we typically use a nested loop.
  - example: finding the maximum value in a 2-D array.

```
public static int maxValue(int[][] arr) {
    int max = arr[0][0];
    for (int r = 0; r < arr.length; r++) {
        for (int c = 0; c < arr[r].length; c++) {
            if (arr[r][c] > max) {
                max = arr[r][c];
            }
        }
    }

    return max;
}
```

Optional: Other Multi-Dimensional Arrays

- It's possible to have a "ragged" 2-D array in which different rows have different numbers of columns:

```
int[][] foo = {{11, 22, 33},
               {7, 20, 30, 40},
               {1, 2}};
```



foo

11 22 33

10 20 30 40

1 2

- We can also create arrays of higher dimensions.
  - example: a three-dimensional matrix:

```
double[][][] matrix = new double[2][5][4];
```

Unit 4, Part 2

File Processing

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

A Class for Representing a File

- The `File` class in Java is used to represent a file on disk.
- To use it, we need to import the `java.io` package:
```
    import java.io.*;
```
- Here's how we typically create a `File` object:
```
    File f = new File("<filename>");
```
- Here are some useful methods from this class:
```
   public boolean exists()
   public boolean canRead()
   public boolean canWrite()
   public boolean delete()
   public long length()
   public String getName()
   public String getPath()
```
  See the Java API documentation for more info.

Review: Scanner Objects

- We've been using a `Scanner` object to read from the console:

```
    Scanner console = new Scanner(System.in);
```

tells the constructor to
construct a `Scanner` object
that reads from the console

- `Scanner` methods:

```
    next()
    nextInt()
    nextDouble()
    nextLine()
```

Reading from a Text File

- We can also use a `Scanner` object to read from a text file:

```
 File f = new File("<filename>");
 Scanner input = new Scanner(f);
```

tells the constructor to
construct a `Scanner` object
that reads from the file

- We can combine the two lines above into a single line:

```
 Scanner input = new Scanner(new File("<filename>"));
```

- We use a different name for the `Scanner` (`input`),
to stress that we're reading from an input file.
- All of the same `Scanner` methods can be used.

Scanner Lookahead and Files

- When reading a file, we often don't know how big the file is.
- Solution: use an indefinite loop and a `Scanner` "lookahead" method.
- Basic structure:

```
Scanner input = new Scanner(new File(<filename>));
while (input.hasNextLine()) {
    String line = input.nextLine();
    // code to process the line goes here…
}
```

- `hasNextLine()` returns:
  - `true` if there's at least one more line of the file to be read
  - `false` if we've reached the end of the file

Sample Problem: Printing the Contents of a File

- Assume that we've already created a `Scanner` called `input` that is connected to a file.
- Here's the code for printing its contents:

```
while (input.hasNextLine()) {
    String line = input.nextLine();
    System.out.println(line);
}
```

File-Processing Exceptions

- Recall: An *exception* is an error that occurs at runtime as a result of some type of "exceptional" circumstance.

- We've seen several examples:

```
StringIndexOutOfBoundsException
IllegalArgumentException
TypeMismatchException
```

- When using a `Scanner` to process a file, we can get a `FileNotFoundException`
  - if the file that we specify isn't there
  - if the file is inaccessible for some reason

Checked vs. Unchecked Exceptions

- Most of the exceptions we've seen thus far have been *unchecked* exceptions.
  - we do *not* need to handle them
  - instead, we usually take steps to avoid them

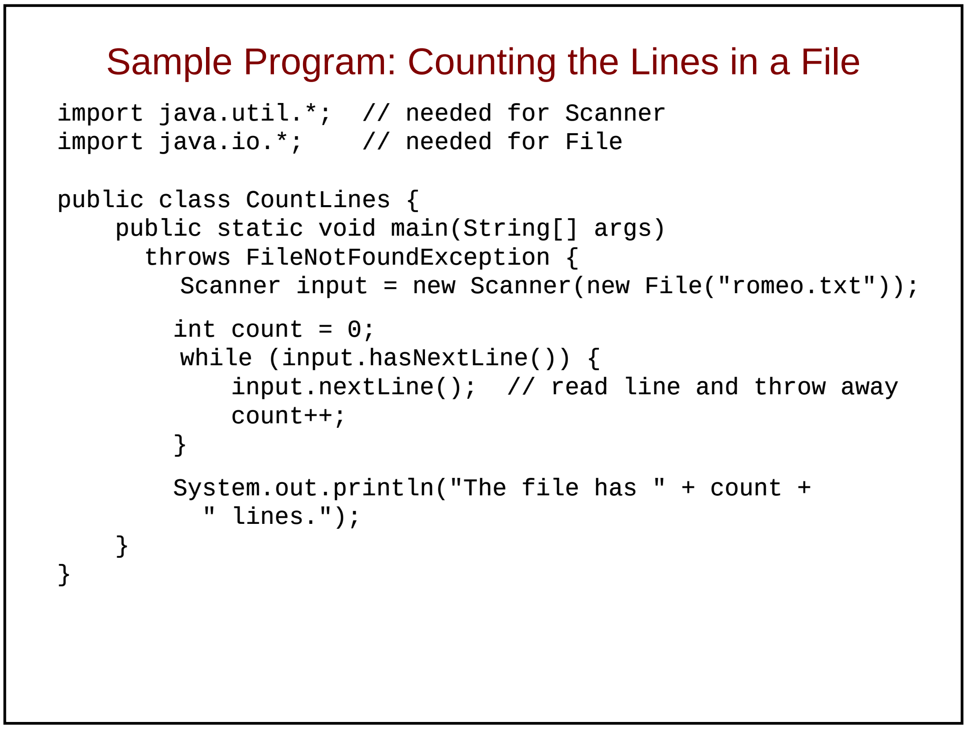
- `FileNotFoundException` is a *checked* exception. The compiler checks that we either:

  1) handle it

  2) declare that we don't handle it

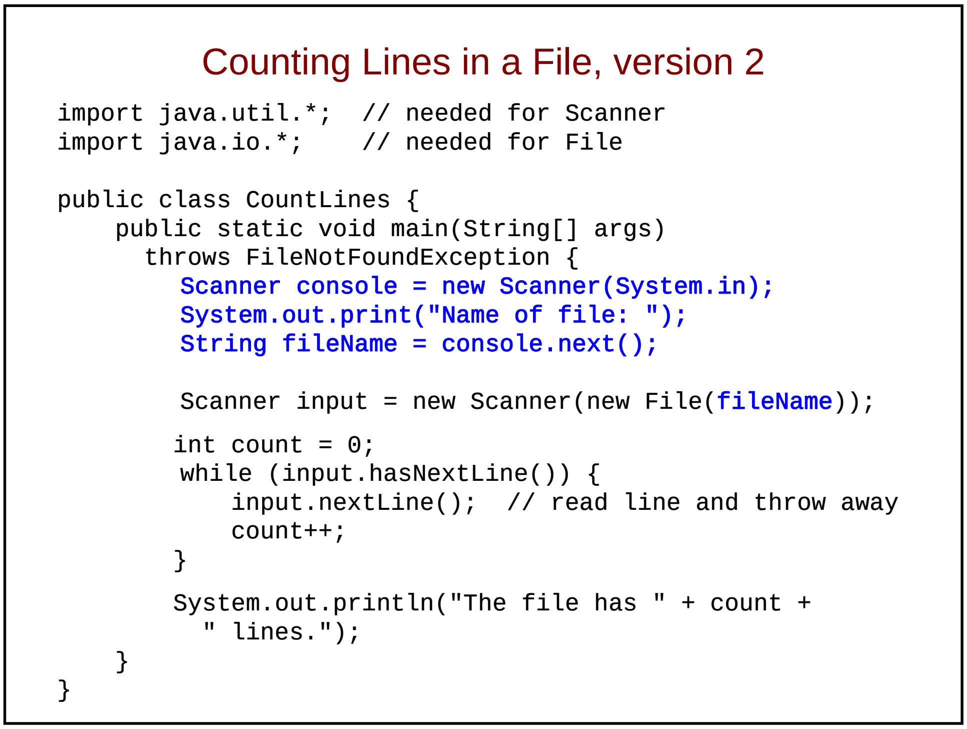
- For now, we'll take option 2. We do this by adding a `throws` clause to the header of any method in which a `Scanner` for a file is created:

```
public static void main(String[] args)
  throws FileNotFoundException {
```



Sample Program: Counting the Lines in a File

```
import java.util.*;  // needed for Scanner
import java.io.*;    // needed for File

public class CountLines {
    public static void main(String[] args)
      throws FileNotFoundException {
        Scanner input = new Scanner(new File("romeo.txt"));
        int count = 0;
        while (input.hasNextLine()) {
            input.nextLine();  // read line and throw away
            count++;
        }
        System.out.println("The file has " + count +
          " lines.");
    }
}
```



Counting Lines in a File, version 2

```
import java.util.*;  // needed for Scanner
import java.io.*;    // needed for File

public class CountLines {
    public static void main(String[] args)
      throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        System.out.print("Name of file: ");
        String fileName = console.next();

        Scanner input = new Scanner(new File(fileName));
        int count = 0;
        while (input.hasNextLine()) {
            input.nextLine();  // read line and throw away
            count++;
        }
        System.out.println("The file has " + count +
          " lines.");
    }
}
```

Counting Lines in a File, version 3

```
…
public static void main(String[] args)
  throws FileNotFoundException {
     Scanner console = new Scanner(System.in);
     System.out.print("Name of file: ");
     String fileName = console.next();
     System.out.println("The file has " +
        numLines(fileName) + " lines.");
}

public static int numLines(String fileName)
  throws FileNotFoundException {
     Scanner input = new Scanner(new File(fileName));
     int count = 0;
     while (input.hasNextLine()) {
         input.nextLine();  // read line and throw away
         count++;
     }
     return count;
}
```

- We put the counting code in a separate method (`numLines`).
- Both `numLines` and `main` need a `throws` clause.

Extracting Data from a File

- Collections of data are often stored in a text file.
- Example: the results of a track meet might be summarized in a text file that looks like this:

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Fran Flash,BU,800m,2:15:00
Tammy Turtle,UMass,800m,4:00:00
```

- Each line of the file represents a *record*.
- Each record is made up of multiple *fields*.
- In this case, the fields are separated by commas.
  - known as a CSV file – comma separated values
  - the commas serve as *delimiters*
  - could also use spaces or tabs (`'\t'`) instead of commas

Extracting Data from a File (cont.)

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Fran Flash,BU,800m,2:15:00
Tammy Turtle,UMass,800m,4:00:00
```

- We want a program that:
  - reads in a results file like the one above
  - extracts and prints only the results for a particular school
    - with the name of the school omitted
- Basic approach:
  - ask the user for the school of interest (the *target school)*
  - read one line at a time from the file
  - split the line into fields
  - if the field corresponding to the school name matches the target school, print out the other fields in that record

Splitting a String

- The `String` class includes a method named `split().`
  - breaks a string into component strings
  - takes a parameter indicating what delimiter should be used when performing the split
  - returns a `String` array containing the components
- Example:

```
> String sentence = "How now brown cow?";
> String[] words = sentence.split(" ");
> words[0]
"How"
> words[1]
"now"
> words[3]
"cow?"
> words.length
4
```

Extracting Data from a File (cont.)

```
import java.util.*;  // needed for Scanner
import java.io.*;    // needed for File

public class ExtractResults {
  public static void main(String[] args)
    throws FileNotFoundException {
      Scanner console = new Scanner(System.in);

      System.out.print("School to extract: ");
      String targetSchool = console.nextLine();

      Scanner input = new Scanner(new File("results.txt"));
      while (input.hasNextLine()) {
          String record = input.nextLine();
          String[] fields = record.split(",");

          if (fields[1].equals(targetSchool)) {
              System.out.print(fields[0] + ",");
              System.out.println(fields[2] + "," + fields[3]);
          }
      }
  }
}
```

- How can we modify it to print a message when no results are found for the target school?

Example Problem: Averaging Enrollments

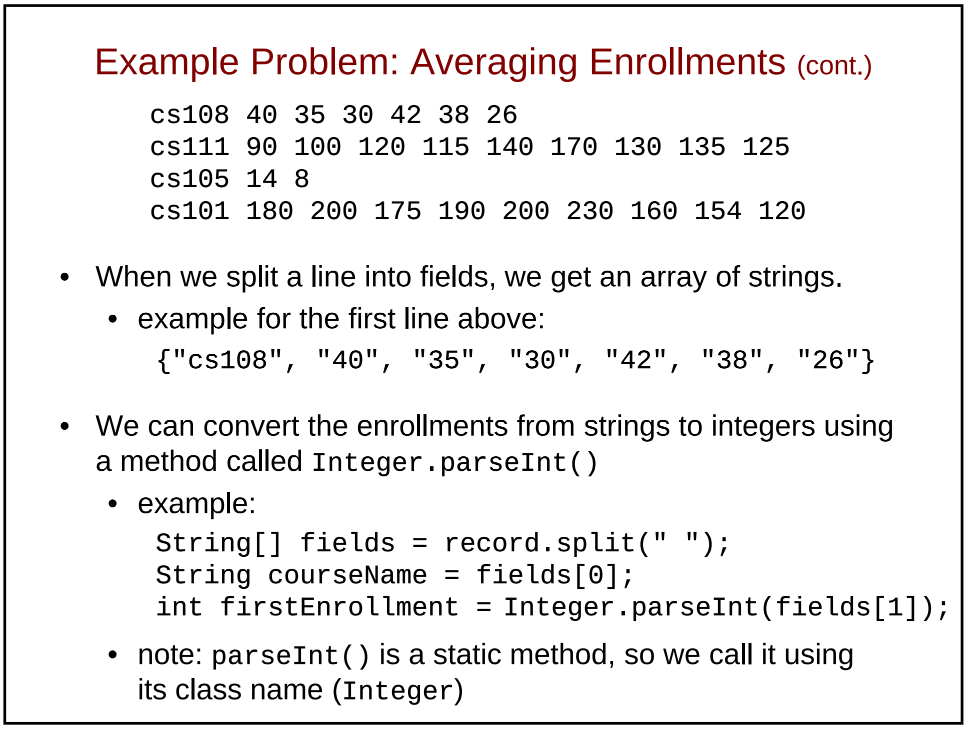
- Let's say that we have a file showing how course enrollments have changed over time:

```
cs111 90 100 120 115 140 170 130 135 125
cs105 14 8
cs108 40 35 30 42 38 26
cs101 180 200 175 190 200 230 160 154 120
```

- For each course, we want to compute the average enrollment.
    - different courses have different numbers of values

- Initial pseudocode:

```
while (there is another course in the file) {
    read the line corresponding to the course
    split it into an array of fields
    average the fields for the enrollments
    print the course name and average enrollment
}
```



Example Problem: Averaging Enrollments (cont.)

```
cs108 40 35 30 42 38 26
cs111 90 100 120 115 140 170 130 135 125
cs105 14 8
cs101 180 200 175 190 200 230 160 154 120
```

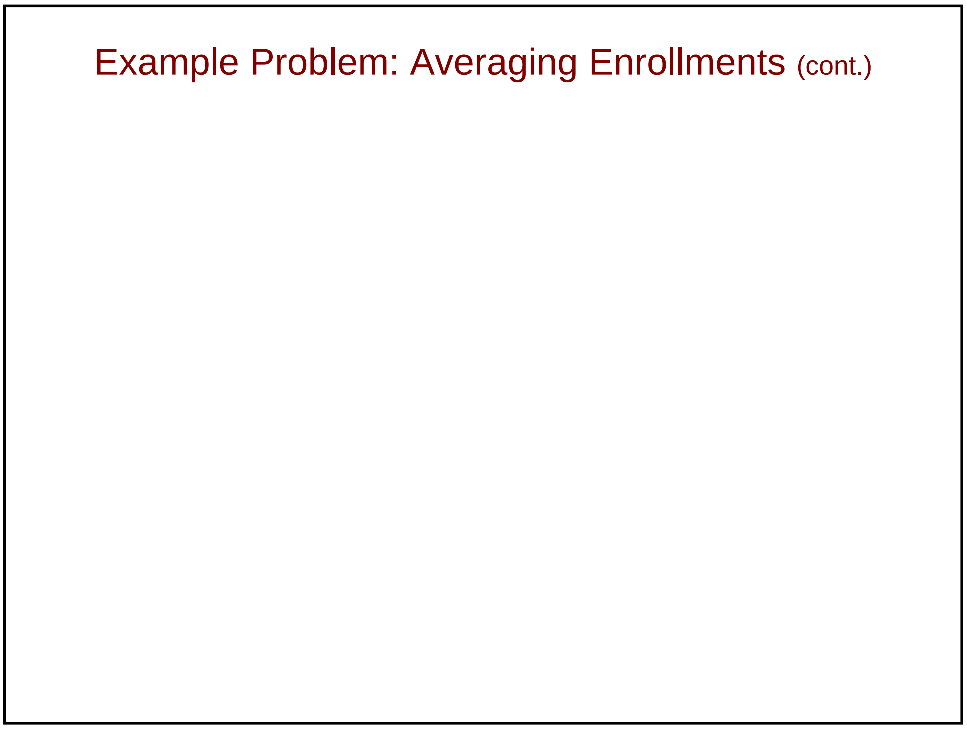
- When we split a line into fields, we get an array of strings.
  - example for the first line above:

    ```
    {"cs108", "40", "35", "30", "42", "38", "26"}
    ```

- We can convert the enrollments from strings to integers using a method called `Integer.parseInt()`
  - example:

    ```
    String[] fields = record.split(" ");
    String courseName = fields[0];
    int firstEnrollment = Integer.parseInt(fields[1]);
    ```

  - note: `parseInt()` is a static method, so we call it using its class name (`Integer`)



Example Problem: Averaging Enrollments (cont.)

Other Details About Reading Text Files

- Although we think of a text file as being two-dimensional (like a piece of paper), the computer treats it as a one-dimensional string of characters.
  - example: the file containing these lines

    ```
    Hello, world.
    How are you?
    I'm tired.
    ```

    is represented like this:

    ```
    Hello, world.\nHow are you?\nI'm tired.\n
    ```

- When reading a file using a `Scanner`, you are limited to *sequential* accesses in the forward direction.
  - you can't back up
  - you can't jump to an arbitrary location
  - to go back to the beginning of the file, you need to create a new `Scanner` object.

Optional Extra Topic: Writing to a Text File

- To write to a text file, we can use a `PrintStream` object, which has the same methods that we've used with `System.out`:
  - `print(), println()`

- Actually, `System.out` *is* a `PrintStream` that has been constructed to print to the console.

- To instantiate a `PrintStream` for a file:

  ```
  File f = new File("<filename>");
  PrintStream output = new PrintStream(f);
  ```

- We can also combine these two steps:

  ```
  PrintStream output = new PrintStream(
    new File("<filename>"));
  ```

- If there's an existing file with the same name, it will be overwritten.

Copying a Text File

```
import java.util.*;  // needed for Scanner
import java.io.*;    // needed for File

public class CopyFile {
    public static void main(String[] args)
      throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        System.out.print("Name of original file: ");
        String original = console.next();
        System.out.print("Name of copy: ");
        String copy = console.next();

        Scanner input = new Scanner(new File(original));
        PrintStream output = new PrintStream(new File(copy));

        while (input.hasNextLine()) {
            String line = input.nextLine();
            output.println(line);
        }
    }
}
```

- How could we combine the two lines in the body of the `while` loop?

Our Track-Meet Program Revisited

```
import java.util.*;  // needed for Scanner
import java.io.*;    // needed for File

public class ExtractResults {
  public static void main(String[] args)
    throws FileNotFoundException {
      Scanner console = new Scanner(System.in);

      System.out.print("School to extract: ");
      String targetSchool = console.nextLine();

      Scanner input = new Scanner(new File("results.txt"));
      while (input.hasNextLine()) {
          String record = input.nextLine();
          String[] fields = record.split(",");

          if (fields[1].equals(targetSchool)) {
              System.out.print(fields[0] + ",");
              System.out.println(fields[2] + "," + fields[3]);
          }
      }
  }
}
```

- How can we modify it to print the extracted results to a separate file?

Optional Extra Topic: Binary Files

- Not all files are text files.
- *Binary files* don't store the string representation of non-string values.
  - instead, they store their *binary* representation – the way they are stored in memory
- Example: 125
  - the text representation of 125 stores the string "125" – i.e., the characters for the individual digits in the number

'1'	'2'	'5'

↔

49	50	53

  - the binary representation of 125 stores the four-byte binary representation of the integer 125

0	0	0	125

Unit 4, Part 3

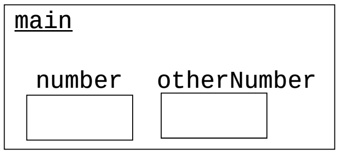
Recursion

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Method Frames

- When you make a method call, the Java runtime sets aside a block of memory known as the *frame* of that method call.

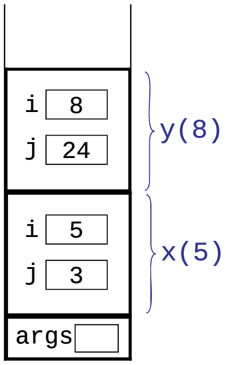


- The frame is used to store:
  - the formal parameters of the method
  - any *local variables* – variables declared within the method
- A given frame can only be accessed by statements that are part of the corresponding method call.

Frames and the Stack

- The frames we've been speaking about are stored in a region of memory known as *the stack*.
- For each method call, a new frame is added to the top of the stack.

```
public class Foo {
  public static int y(int i) {
      int j = i * 3;
      return j;
  }
  public static int x(int i) {
      int j = i - 2;
      return y(i + j);
  }
  public static void
    main(String[] args) {
      System.out.println(x(5));
  }
}
```



- When a method completes, its stack frame is removed.

Iteration

- Whenever we've encountered a problem that requires repetition, we've used *iteration* – i.e., some type of loop.
- Sample problem: printing the series of integers from n1 to n2, where n1 <= n2.
  - example: `printSeries(5, 10)` should print the following:

    `5, 6, 7, 8, 9, 10`
- Here's an iterative solution to this problem:

```
public static void printSeries(int n1, int n2) {
    for (int i = n1; i < n2; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(n2);
}
```

Recursion

- An alternative approach to problems that require repetition is to solve them using *a method that calls itself*.
- Applying this approach to the print-series problem gives:

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- A method that calls itself is a *recursive* method.
- This approach to problem-solving is known as *recursion*.

Tracing a Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.println(7);
            return
        return
    return
```

Recursive Problem-Solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.
- This simplest problem is known as the *base case*.

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {           // base case
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

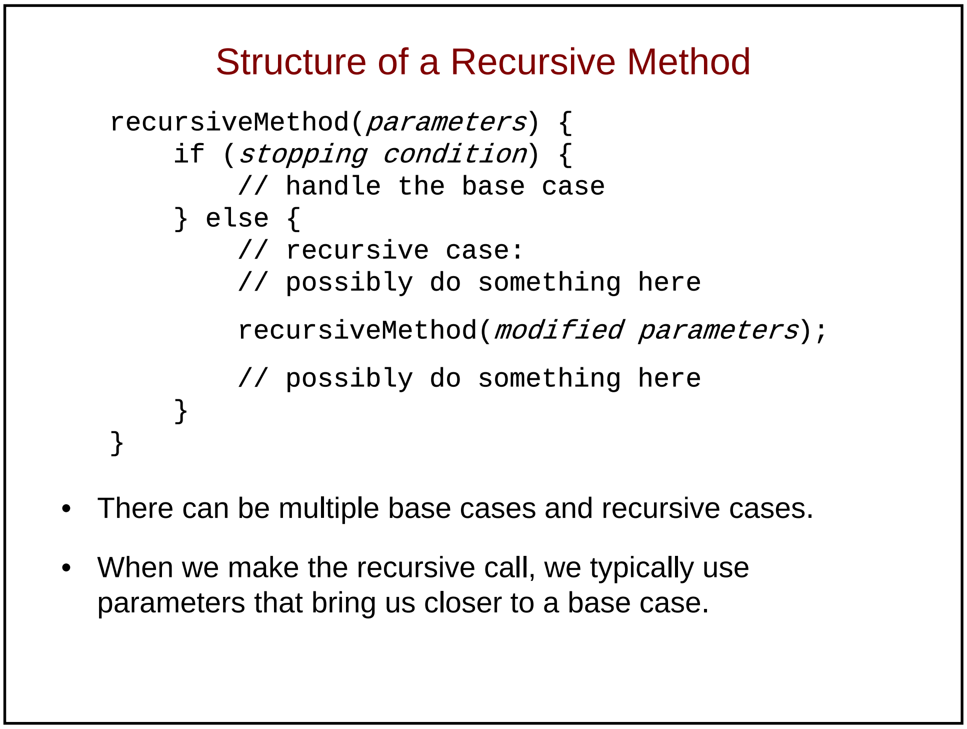
- The base case stops the recursion, because it doesn't make another call to the method.

Recursive Problem-Solving (cont.)

- If the base case hasn't been reached, we execute the *recursive case.*

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {           // base case
        System.out.println(n2);
    } else {                  // recursive case
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

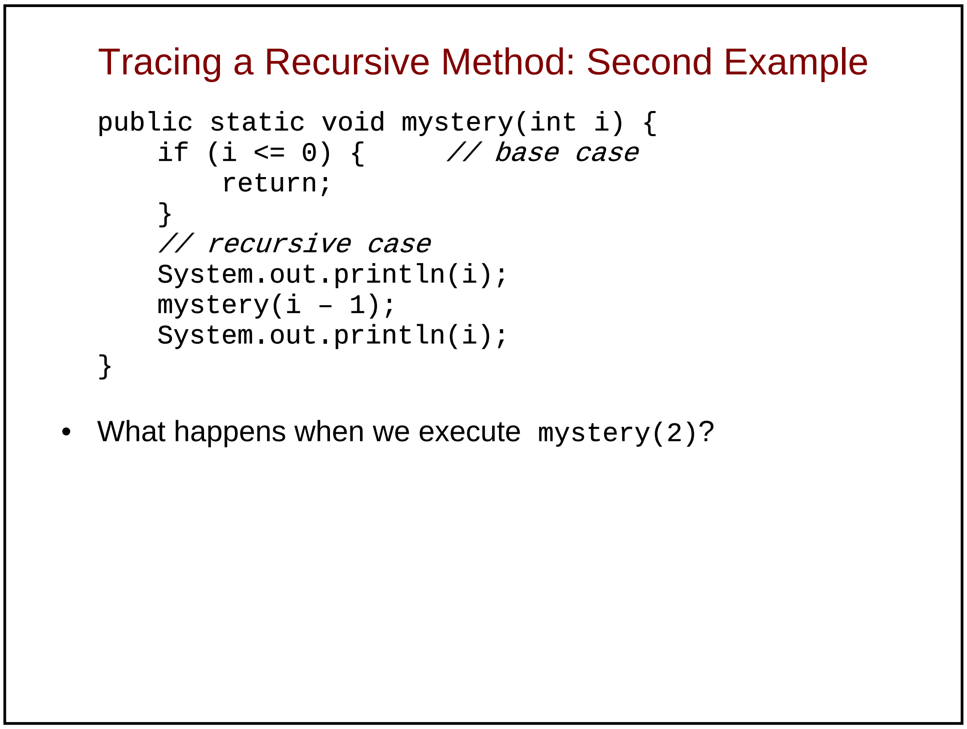
- The recursive case:
  - reduces the overall problem to one or more simpler problems of the same kind
  - makes recursive calls to solve the simpler problems



Structure of a Recursive Method

```
recursiveMethod(parameters) {
    if (stopping condition) {
        // handle the base case
    } else {
        // recursive case:
        // possibly do something here

        recursiveMethod(modified parameters);

        // possibly do something here
    }
}
```

- There can be multiple base cases and recursive cases.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.



Tracing a Recursive Method: Second Example

```
public static void mystery(int i) {
    if (i <= 0) {     // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i – 1);
    System.out.println(i);
}
```

- What happens when we execute `mystery(2)`?

Printing a File to the Console

- Here's a method that prints a file using iteration:

```
public static void print(Scanner input) {
    while (input.hasNextLine()) {
        System.out.println(input.nextLine());
    }
}
```

- Here's a method that uses recursion to do the same thing:

```
public static void printRecursive(Scanner input) {
    // base case
    if (!input.hasNextLine()) {
        return;
    }

    // recursive case
    System.out.println(input.nextLine());
    printRecursive(input);  // print the rest
}
```

Printing a File in Reverse Order

- What if we want to print the lines of a file in reverse order?
- It's not easy to do this using iteration. Why not?
- It's easy to do it using recursion!
- How could we modify our previous method to make it print the lines in reverse order?

```
public static void printRecursive(Scanner input) {
    if (!input.hasNextLine()) {   // base case
        return;
    }

    String line = input.nextLine();
    System.out.println(line);
    printRecursive(input);  // print the rest
}
```

Printing a File in Reverse Order (cont.)

- An iterative approach to reversing the file would need to:
  - read all of the lines in the file and store them in a temporary data structure (e.g., an array)
  - retrieve the lines from the data structure and print them in reverse order
- The recursive method doesn't need a separate data structure.
  - the lines are stored in the stack frames for the recursive method calls!

A Recursive Method That Returns a Value

- Simple example: summing the integers from 1 to n

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

- Example of this approach to computing the sum:

```
sum(6)  =  6 + sum(5)
        =  6 + 5 + sum(4)
        …
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

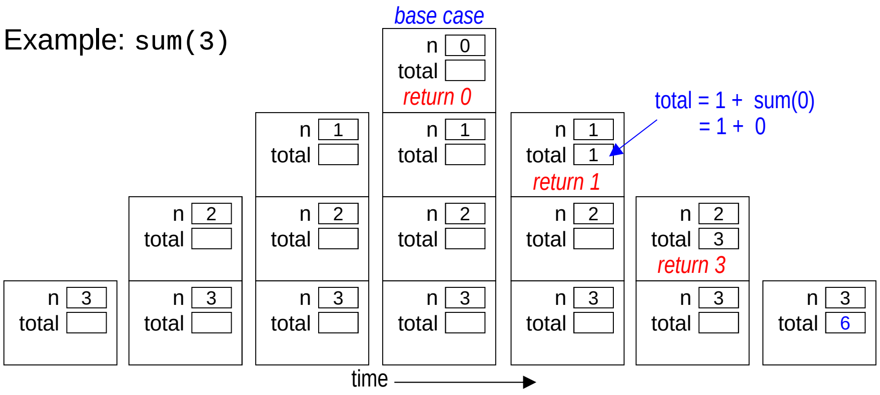
- What happens when we execute `int x = sum(3);` from inside the `main()` method?

```
main() calls sum(3)
    sum(3) calls sum(2)
        sum(2) calls sum(1)
            sum(1) calls sum(0)
                sum(0) returns 0
            sum(1) returns 1 + 0 or 1
        sum(2) returns 2 + 1 or 3
    sum(3) returns 3 + 3 or 6
main()
```

Tracing a Recursive Method on the Stack

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

Example: `sum(3)`



Alternative `sum()` Method

- We can rewrite it so that the recursive call is part of the `return` statement:

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

⬇

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    return n + sum(n – 1);
}
```

Another Option for Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    return n + sum(n – 1);
}
```

Infinite Recursion

- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get *infinite recursion*.
    - produces *stack overflow* – there's no room for more frames on the stack!
- Example: here's a version of our `sum()` method that uses a different test for the base case:

```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }
    int total = n + sum(n – 1);
    return total;
}
```

- what values of n would cause infinite recursion?

Thinking Recursively

- When solving a problem using recursion, ask yourself these questions:
    1. What are the base cases?
        - i.e., which instances of the problem are small enough to solve directly?
    2. How can we break the problem down into one or more smaller subproblems?
        - make recursive method calls to solve the subproblems
    3. What additional work does the current call of the method need to do?
        - how can we build the solution to the problem from the solutions to the subproblems?

Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string
- Thus, we can easily use recursion to process a string.
  - process one or two of the characters
  - make a recursive call to process the rest of the string
- Example: print a string vertically, one character per line:

```
public static void printVertical(String str) {
    if (str == null || str.equals("")) {
        return;
    }

    System.out.println(str.charAt(0)); // first char
    printVertical(str.substring(1));   // rest of string
}
```

Short-Circuited Evaluation

- The second operand of both the `&&` and `||` operators will not be evaluated if the result can be determined on the basis of the first operand alone.
- `expr1 || expr2`
  if `expr1` evaluates to `true`, `expr2` is not evaluated, because we already know that `expr1 || expr2` is `true`
  - example from the last slide:

```
if (str == null || str.equals("")) {
    return;
}
// if str is null, we won't check for empty string.
// This prevents a null pointer exception!
```

- `expr1 && expr2`
  if `expr1` evaluates to ________, `expr2` is not evaluated, because we already know that `expr1 && expr2` is ________.

Counting Occurrences of a Character in a String

- Let's design a recursive method called `numOccur()`.
- `numOccur(ch, str)` should return the number of times that the character `ch` appears in the string `str`
- Thinking recursively:

Counting Occurrences of a Character in a String (cont.)

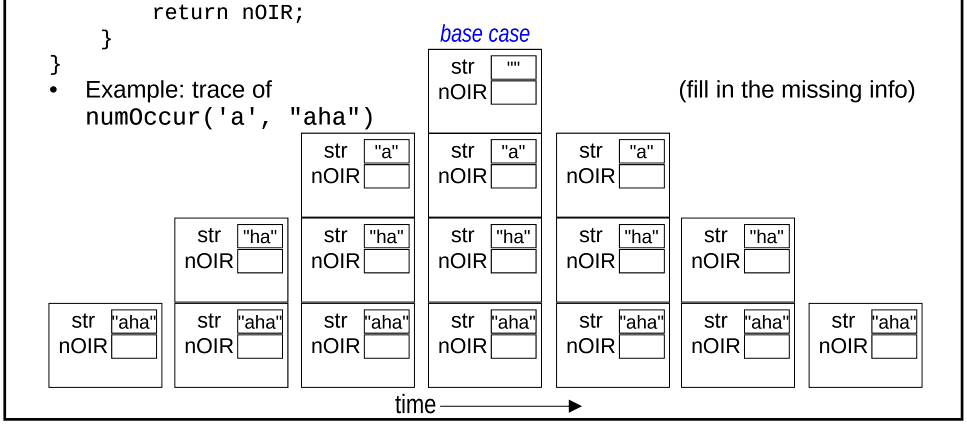
- Put the method definition here:

Tracing a Recursive Method on the Stack

```
public static int numOccur(char ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }
    int nOIR = numOccur(ch, str.substring(1));
    if (str.charAt(0) == ch) {
        return 1 + nOIR;
    } else {
        return nOIR;
    }
}
```

- Example: trace of `numOccur('a', "aha")`

(fill in the missing info)



base case

			str "" nOIR			
		str "a" nOIR	str "a" nOIR	str "a" nOIR		
	str "ha" nOIR	str "ha" nOIR	str "ha" nOIR	str "ha" nOIR	str "ha" nOIR	
str "aha" nOIR	str "aha" nOIR	str "aha" nOIR	str "aha" nOIR	str "aha" nOIR	str "aha" nOIR	str "aha" nOIR

time →

Common Mistake

- This version of the method does *not* work:

```
public static int numOccur(char ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }

    int count = 0;
    if (str.charAt(0) == ch) {
        count++;
    }

    numOccur(ch, str.substring(1));
    return count;
}
```

Another Faulty Approach

- Some people make `count` "global" to fix the prior version:

```
public static int count = 0;

public static int numOccur(char ch, String str) {
    if (str == null || str.equals("")) {
        return 0;
    }

    if (str.charAt(0) == ch) {
        count++;
    }

    numOccur(ch, str.substring(1));
    return count;
}
```

- Not recommended, and not allowed on the problem sets!
- Problems with this approach?

Testing for a Prefix

- Let's design a recursive method called `isPrefix()`.
- `isPrefix(str1, str2)` should return:
  - `true` if `str1` is a prefix of `str2`
  - `false` if `str1` is not a prefix of `str2`
- examples:

  `isPrefix("recur", "recurse")` should return `true`

  `isPrefix("record", "recurse")` should return `false`
- Special case: we will consider the empty string (`""`) to be the prefix of any string.
- Thinking recursively:

Testing for a Prefix (cont.)

- Put the method definition here:

Recursion vs. Iteration

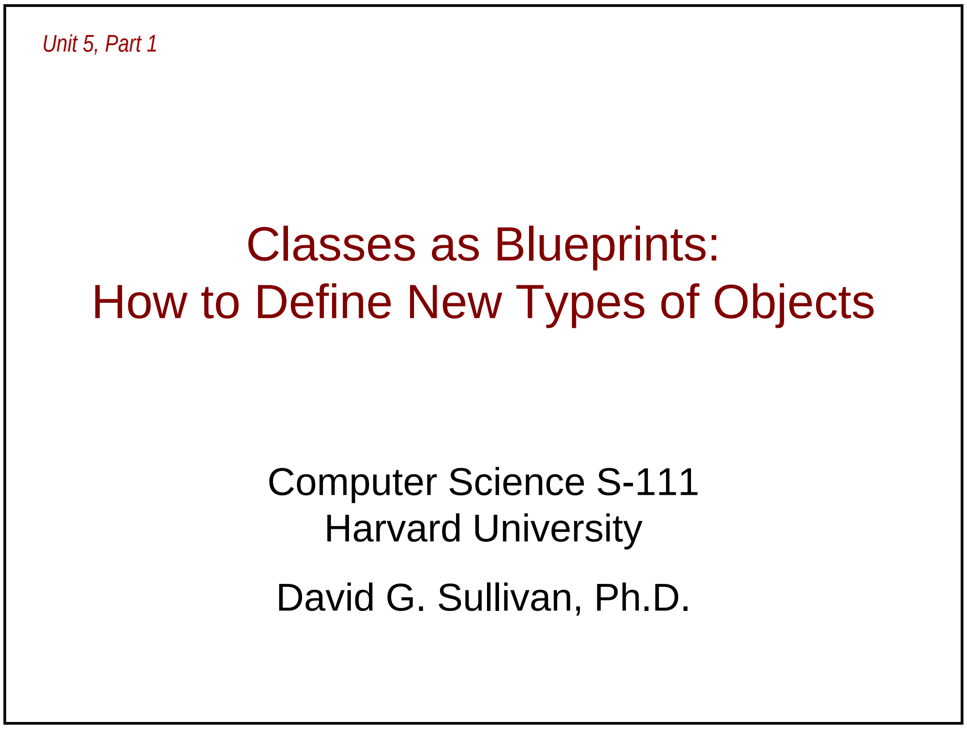
- Some problems are much easier to solve using recursion.
- Other problems are just as easy to solve using iteration.
- Recursion is a bit more costly because of the overhead involved in invoking a method.
  - also: in some cases, there may not be room on the stack
- Rule of thumb:
  - if it's easier to formulate a solution recursively, use recursion, unless the cost of doing so is too high
  - otherwise, use iteration

Extra Practice: A Palindrome Checker

- A *palindrome* is a word or phrase that reads the same forward and backward (ignoring spaces, punctuation, and case).
  - examples: "radar", "mom", "live not on evil"
- `isPalindrome(str)` should return `true` if `str` is a palindrome, and `false` otherwise.
- Thinking recursively:
  1. *What are the base cases?*
  2. *How can we break the problem down into one or more smaller subproblems?*
  3. *What additional work does the current call of the method need to do? How can we build the solution to the problem from the solutions to the subproblems?*

A Palindrome Checker (cont.)

- Put the method definition here:

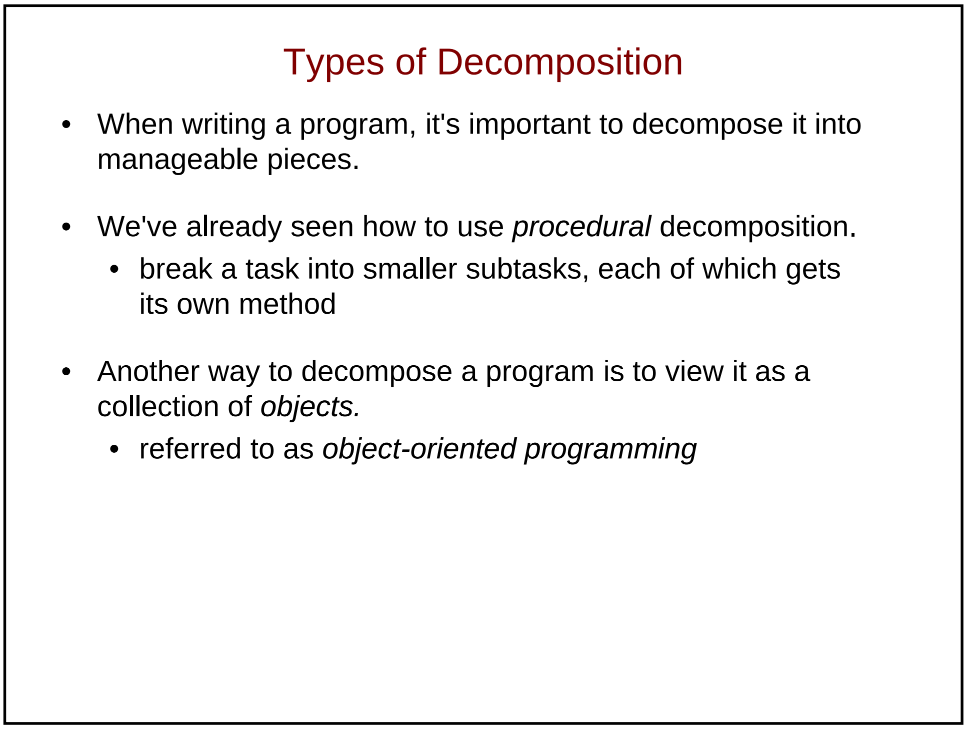


Unit 5, Part 1

Classes as Blueprints: How to Define New Types of Objects

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.



Types of Decomposition

- When writing a program, it's important to decompose it into manageable pieces.
- We've already seen how to use *procedural* decomposition.
  - break a task into smaller subtasks, each of which gets its own method
- Another way to decompose a program is to view it as a collection of *objects.*
  - referred to as *object-oriented programming*

Review: What is an Object?

- An object groups together:
  - one or more data values (the object's *fields*)
  - a set of operations that the object can perform (the object's *methods*)

Review: Using an Object's Methods

- An object's methods are different from the static methods that we've been writing thus far.
  - they're called *non-static* or *instance* methods
- When using an instance method, we specify the object to which the method belongs by using dot notation:

```
String firstName = "Perry";
int len = firstName.length();
```

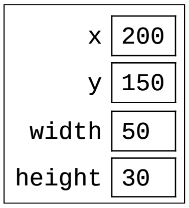
- Using an instance method is like sending a message to an object, asking it to perform an operation.
- We refer to the object on which the method is invoked as either:
  - the *called object*
  - the *current object*

Review: Classes as Blueprints

- We've been using classes as containers for our programs.
- A class can also serve as a blueprint – as the definition of a new type of object.
  - specifying the fields and methods that objects of that type will have
- The objects of a given class are built according to its blueprint.
- Objects of a class are referred to as *instances* of the class.

`Rectangle` Objects

- Java comes with a built-in `Rectangle` class.
  - in the `java.awt` package
- Each `Rectangle` object has the following fields:
  - `x` – the x coordinate of its upper left corner
  - `y` – the y coordinate of its upper left corner
  - `width`
  - `height`
- Here's an example of one:



x	200
y	150
width	50
height	30

`Rectangle` Methods

- A `Rectangle`'s methods include:

```
void grow(int h, int v)
void translate(int x, int y)
double getWidth()
double getHeight()
double getX()
double getY()
```

Writing a "Blueprint Class"

- To illustrate how to define a new type of object, let's write our own class for `Rectangle` objects.

```
public class Rectangle {
    ...
```

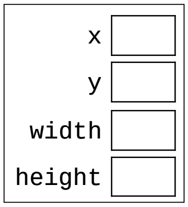
- As always, the class definition goes in an appropriately named text file.
    - in this case: `Rectangle.java`

Using Fields to Capture an Object's State

- Here's the first version of our `Rectangle` class:

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;
}
```

  - it declares four fields, each of which stores an `int`
  - each `Rectangle` object gets its own set of these fields



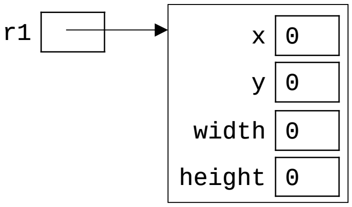
- Another name for a field is an *instance variable.*

Using Fields to Capture an Object's State (cont.)

- For now, we'll create `Rectangle` objects like this:

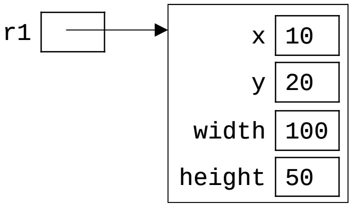
```
Rectangle r1 = new Rectangle();
```

- The fields are initially filled with the default values for their types.
  - just like array elements



- Fields can be accessed using dot notation:

```
r1.x = 10;
r1.y = 20;
r1.width = 100;
r1.height = 50;
```



Client Programs

- Our `Rectangle` class is *not* a program.
  - it has no `main` method
- Instead, it will be used by code defined in other classes.
  - referred to as *client programs* or *client code*
- More generally, when we define a new type of object, we create a building block that can be used in other code.
  - just like the objects from the built-in classes: `String`, `Scanner`, `File`, etc.
  - our programs have been clients of those classes

Initial Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.x = 10;       r1.y = 20;
        r1.width = 100;  r1.height = 50;

        Rectangle r2 = new Rectangle();
        r2.x = 50;       r2.y = 100;
        r2.width = 20;   r2.height = 80;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        int area1 = r1.width * r1.height;
        System.out.println("area = " + area1);

        System.out.println("r2: " + r2.width + " x " + r2.height);
        int area2 = r2.width * r2.height;
        System.out.println("area = " + area2);

        // grow both rectangles
        r1.width += 50;  r1.height += 10;
        r2.width += 5;   r2.height += 30;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Using Methods to Capture an Object's Behavior

- It would be useful to have a method for growing a `Rectangle`.
- One option would be to define a static method:

```
public static void grow(Rectangle r, int dWidth, int dHeight) {
    r.width += dWidth;
    r.height += dHeight;
}
```

- This would allow us to replace the statements

```
     r1.width += 50;
     r1.height += 10;
```

  with the method call

```
     Rectangle.grow(r1, 50, 10);
```

Using Methods to Capture an Object's Behavior

- It would be useful to have a method for growing a `Rectangle`.
- One option would be to define a static method in our `Rectangle` class:

```
public static void grow(Rectangle r, int dWidth, int dHeight) {
    r.width += dWidth;
    r.height += dHeight;
}
```

- This would allow us to replace these statements in the client

```
     r1.width += 50;
     r1.height += 10;
```

  with the method call

```
     Rectangle.grow(r1, 50, 10);
```

  (Note: We need to use the class name, because we're calling the method from outside the `Rectangle` class.)

Using Methods to Capture an Object's Behavior (cont.)

- A better approach is to give each `Rectangle` object the ability to grow itself.
- We do so by defining a non-static or instance method.
- We'll use dot notation to call the instance method:

  `r1.grow(50, 10);`

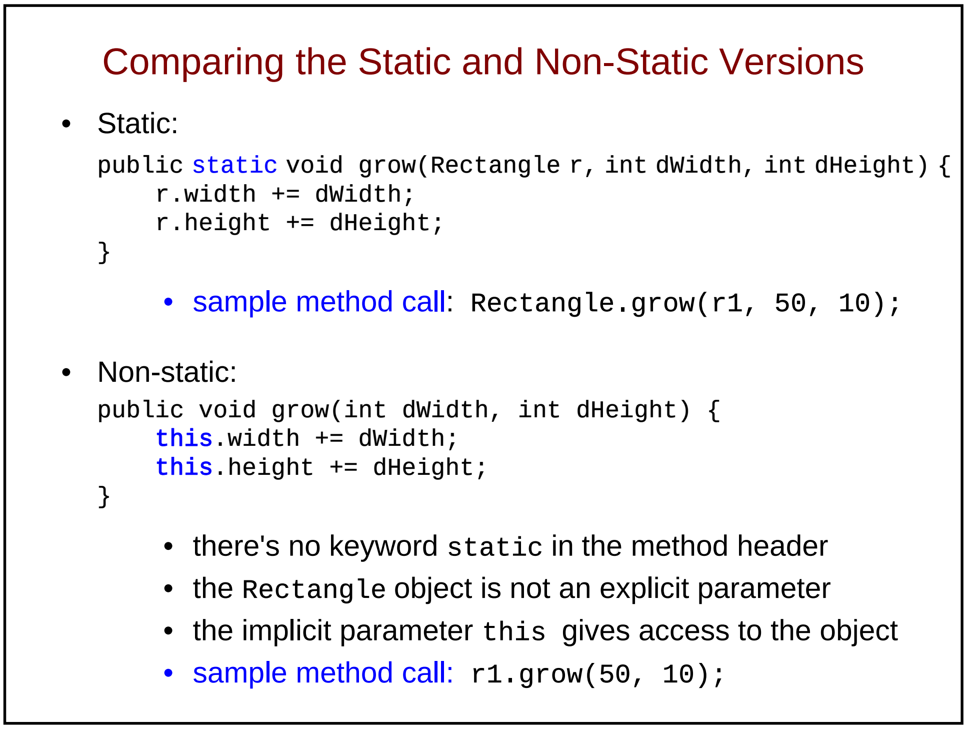
  instead of `Rectangle.grow(r1, 50, 10);`
- This is like sending a message to `r1`, asking it to grow itself.

Using Methods to Capture an Object's Behavior (cont.)

- Here's our `grow` instance method:

```
public void grow(int dWidth, int dHeight) { // no static
    this.width += dWidth;
    this.height += dHeight;
}
```

- We don't pass the `Rectangle` object as an explicit parameter.
- Instead, the Java keyword `this` gives us access to the called object.
  - every instance method has this special variable
  - referred to as the *implicit parameter*
- Example: `r1.grow(50, 10)`
  - `r1` is the called object
  - `this.width` gives us access to `r1`'s `width` field
  - `this.height` gives us access to `r1`'s `height` field



Comparing the Static and Non-Static Versions

- Static:

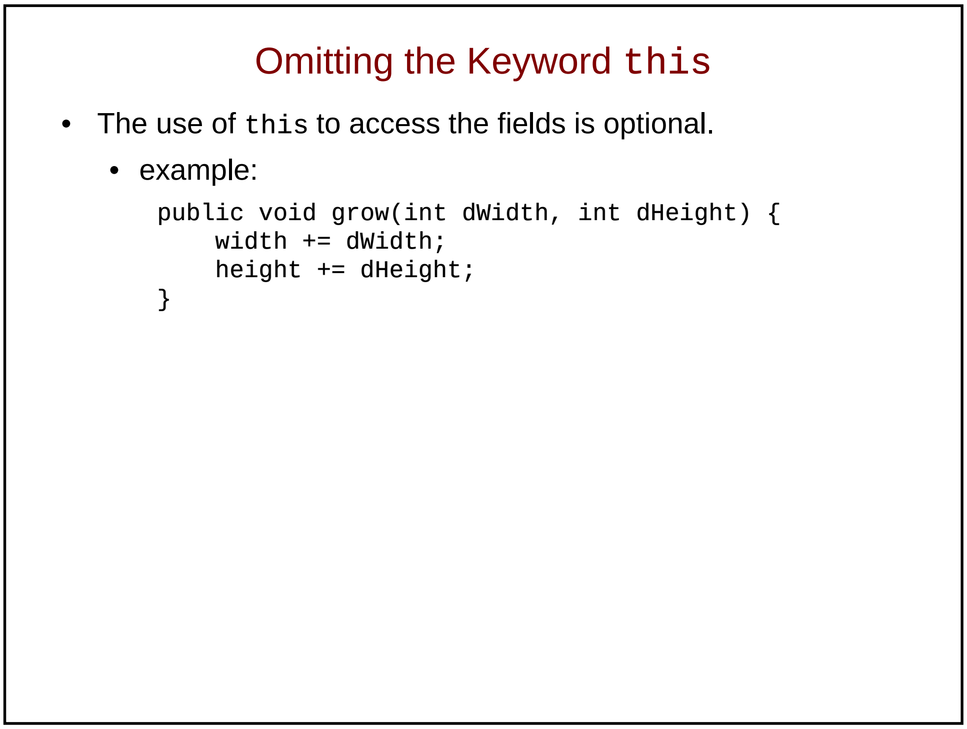
```
public static void grow(Rectangle r, int dWidth, int dHeight) {
    r.width += dWidth;
    r.height += dHeight;
}
```

  - sample method call: `Rectangle.grow(r1, 50, 10);`

- Non-static:

```
public void grow(int dWidth, int dHeight) {
    this.width += dWidth;
    this.height += dHeight;
}
```

  - there's no keyword `static` in the method header
  - the `Rectangle` object is not an explicit parameter
  - the implicit parameter `this` gives access to the object
  - sample method call: `r1.grow(50, 10);`



Omitting the Keyword `this`

- The use of `this` to access the fields is optional.
  - example:

```
public void grow(int dWidth, int dHeight) {
    width += dWidth;
    height += dHeight;
}
```

Another Example of an Instance Method

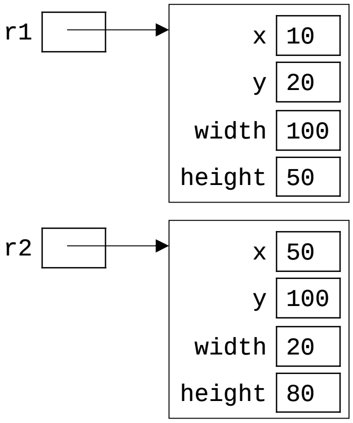
- Here's an instance method for getting the area of a `Rectangle`:

```
public int area() {
    return this.width * this.height;
}
```

- Sample method calls:

```
int area1 = r1.area();
int area2 = r2.area();
```

  - we're asking `r1` and r2 to give us their areas
  - no explicit parameters are needed because the necessary info. is in the objects' fields!



Types of Instance Methods

- There are two main types of instance methods:
  - *mutators* – methods that change an object's internal state
  - *accessors* – methods that retrieve information from an object without changing its state
- Examples of mutators:
  - `grow` in our `Rectangle` class
  - `nextLine` in the `Scanner` class – advances the `Scanner` past the line that was just read
- Examples of accessors:
  - `area` in our `Rectangle` class
  - `hasNextLine` in the `Scanner` class
  - `String` methods: `length`, `substring`, `charAt`, `indexOf`

```
}
```

Second Version of our Rectangle Class

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;

    public void grow(int dWidth, int dHeight) {
        this.width += dWidth;
        this.height += dHeight;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

Initial Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.x = 10;       r1.y = 20;
        r1.width = 100;  r1.height = 50;

        Rectangle r2 = new Rectangle();
        r2.x = 50;       r2.y = 100;
        r2.width = 20;   r2.height = 80;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        int area1 = r1.width * r1.height;
        System.out.println("area = " + area1);

        System.out.println("r2: " + r2.width + " x " + r2.height);
        int area2 = r2.width * r2.height;
        System.out.println("area = " + area2);

        // grow both rectangles
        r1.width += 50;  r1.height += 10;
        r2.width += 5;   r2.height += 30;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Revised Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.x = 10;        r1.y = 20;
        r1.width = 100;   r1.height = 50;

        Rectangle r2 = new Rectangle();
        r2.x = 50;        r2.y = 100;
        r2.width = 20;    r2.height = 80;

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.width + " x " + r2.height);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Practice Defining Instance Methods

- Add a mutator method that moves the rectangle to the right by a specified amount.

- Add an accessor method that gets the x coordinate of the right-hand side of the rectangle

Defining a Constructor

- Our current client program has to use several lines to initialize each `Rectangle` object:

```
Rectangle r1 = new Rectangle();
r1.x = 10;       r1.y = 20;
r1.width = 100;  r1.height = 50;
```

- We'd like to be able to do something like this instead:

```
Rectangle r1 = new Rectangle(10, 20, 100, 50);
```

- To do so, we need to define a *constructor*, a special method that initializes the state of an object when it is created.

Defining a Constructor (cont.)

- Here it is:

```
public Rectangle(int initialX, int initialY,
  int initialWidth, int initialHeight) {
    this.x = initialX;
    this.y = initialY;
    this.width = initialWidth;
    this.height = initialHeight;
}
```

- General syntax for a constructor:

```
public <class name>(<parameter list>) {
    body of the constructor
}
```

- Note that a constructor has no return type.

Third Version of our Rectangle Class

```
public class Rectangle {
    int x;
    int y;
    int width;
    int height;

    public Rectangle(int initialX, int initialY,
      int initialWidth, int initialHeight) {
        this.x = initialX;
        this.y = initialY;
        this.width = initialWidth;
        this.height = initialHeight;
    }

    public void grow(int dWidth, int dHeight) {
        this.width += dWidth;
        this.height += dHeight;
    }

    public int area() {
        return this.width * this.height;
    }
}
```

Revised Client Program

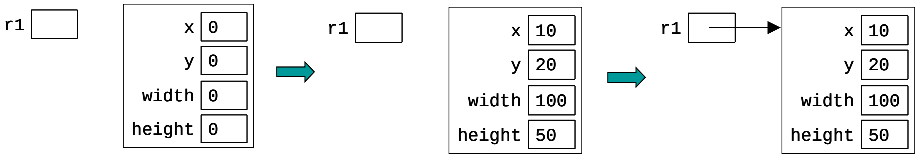
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.width + " x " + r2.height);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

A Closer Look at Creating an Object

- What happens when the following line is executed?

```
Rectangle r1 = new Rectangle(10, 20, 100, 50);
```

- Several different things actually happen:
  1) a new `Rectangle` object is created
     - initially, all fields have their default values
  2) the constructor is then called to assign values to the fields
  3) a reference to the new object is stored in the variable r1



Limiting Access to Fields

- The current version of our `Rectangle` class allows clients to directly access a `Rectangle` object's fields:

```
r1.width = 100;
r1.height += 20;
```

- This means that clients can make inappropriate changes:

```
r1.width = -100;
```

- To prevent this, we can declare the fields to be *private*:

```
public class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;
    ...
}
```

- This indicates that these fields can only be accessed or modified by methods that are part of the `Rectangle` class.

Limiting Access to Fields (cont.)

- Now that the fields are private, our client program won't compile:

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.width + " x " + r2.height);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.width + " x " + r1.height);
        System.out.println("r2: " + r2.width + " x " + r2.height);
    }
}
```

Adding Accessor Methods for the Fields

```
public class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;
    ...
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
    public int getWidth() {
        return this.width;
    }
    public int getHeight() {
        return this.height;
    }
}
```

- These methods are *public*, which indicates that they can be used by code that is outside the `Rectangle` class.

Revised Client Program

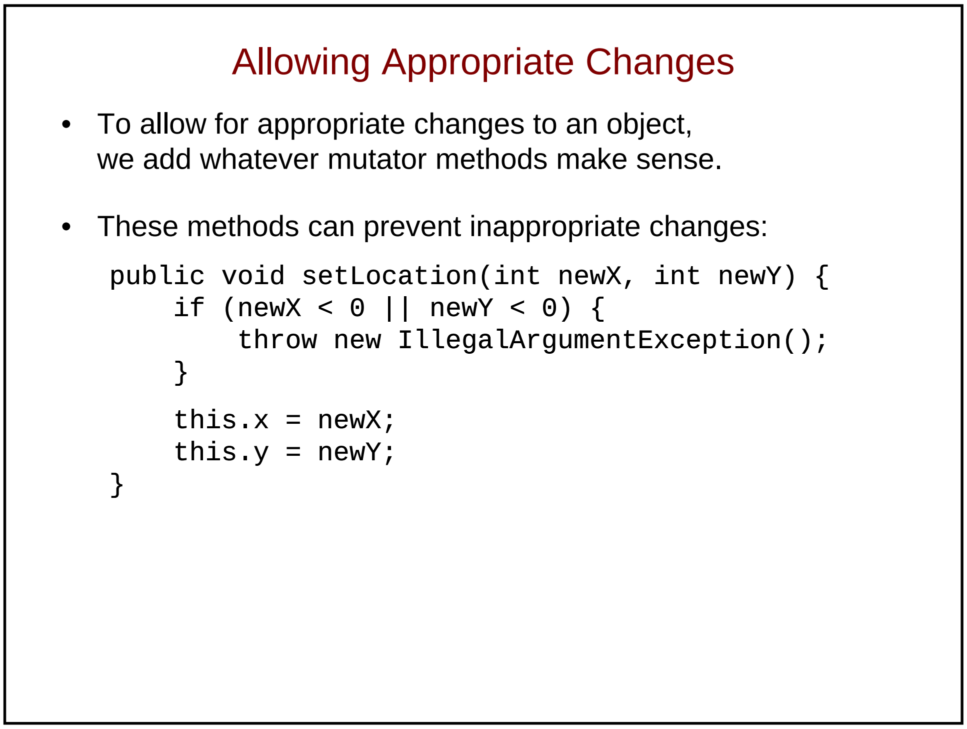
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1.getWidth() + " x " +
          r1.getHeight());
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2.getWidth() + " x " +
          r2.getHeight());
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1.getWidth() + " x " +
          r1.getHeight());
        System.out.println("r2: " + r2.getWidth() + " x " +
          r2.getHeight());
    }
}
```

Access Modifiers

- `public` and `private` are known as *access modifiers*.
  - they specify where a class, field, or method can be used
- A class is usually declared to be `public`:

  ```
  public class Rectangle {
  ```

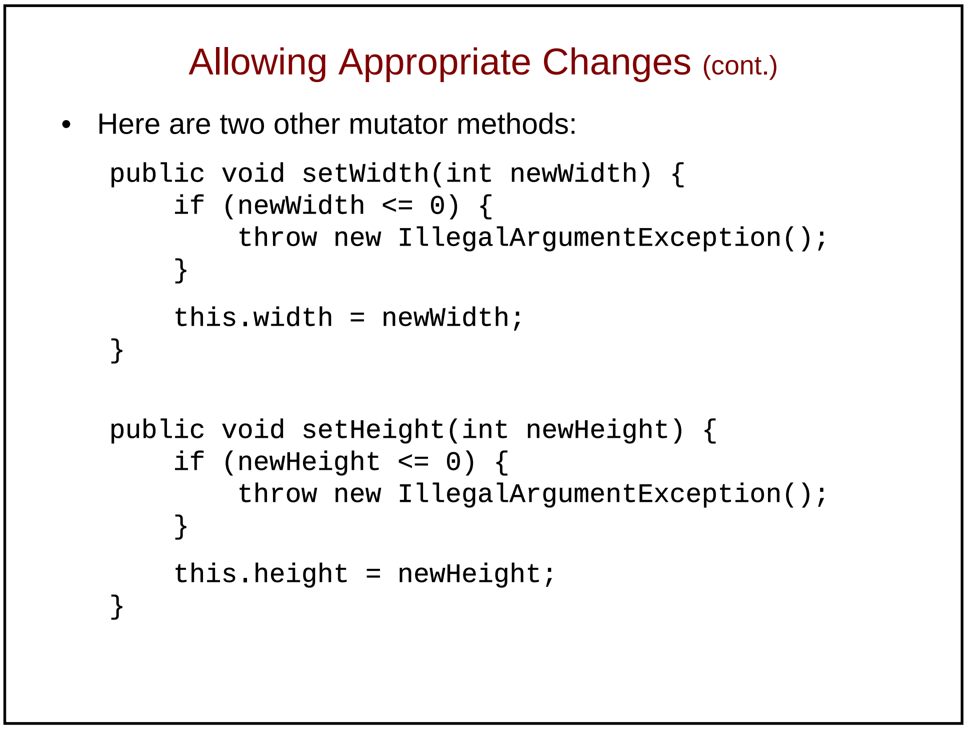
  - indicates that objects of the class can be used anywhere, including in other classes
- Fields are usually declared to be `private`.
- Methods are usually declared to be `public`.
- We occasionally define `private` methods.
  - serve as *helper methods* for the `public` methods
  - cannot be invoked by code that is outside the class



Allowing Appropriate Changes

- To allow for appropriate changes to an object, we add whatever mutator methods make sense.
- These methods can prevent inappropriate changes:

```
public void setLocation(int newX, int newY) {
    if (newX < 0 || newY < 0) {
        throw new IllegalArgumentException();
    }

    this.x = newX;
    this.y = newY;
}
```



Allowing Appropriate Changes (cont.)

- Here are two other mutator methods:

```
public void setWidth(int newWidth) {
    if (newWidth <= 0) {
        throw new IllegalArgumentException();
    }

    this.width = newWidth;
}


public void setHeight(int newHeight) {
    if (newHeight <= 0) {
        throw new IllegalArgumentException();
    }

    this.height = newHeight;
}
```

Instance Methods Calling Other Instance Methods

- Here's another mutator method that we already had:

```
public void grow(int dWidth, int dHeight) {
    this.width += dWidth;
    this.height += dHeight;
}
```

- However, it doesn't prevent inappropriate changes.
- Rather than adding error-checking to it, we can have it call the new mutator methods:

```
public void grow(int dWidth, int dHeight) {
    this.setWidth(this.width + dWidth);
    this.setHeight(this.height + dHeight);
}
```

Revised Constructor

- To prevent invalid values in the fields of a `Rectangle` object, we also need to modify our constructor.
- Here again, we take advantage of the error-checking code that's already present in the mutator methods:

```
public Rectangle(int initialX, int initialY,
  int initialWidth, int initialHeight)
{
    this.setLocation(initialX, initialY);
    this.setWidth(initialWidth);
    this.setHeight(initialHeight);
}
```

  - `setLocation`, `setWidth`, and `setHeight` operate on the newly created `Rectangle` object

Encapsulation

- *Encapsulation* is one of the key principles of object-oriented programming.
  - another name for it is *information hiding*
- It refers to the practice of “hiding” the implementation of a class from users of the class.
  - prevent *direct* access to the internals of an object
    - making the fields private
  - provide *limited*, *indirect* access through a set of methods
    - making them public
- In addition to preventing inappropriate changes, encapsulation allows us to change the implementation of a class without breaking the client code that uses it.

Abstraction

- *Abstraction* involves focusing on the essential properties of something, rather than its inner or low-level details.
  - an important concept in computer science
- Encapsulation leads to abstraction.
  - example: rather than treating a `Rectangle` as four `int`s, we treat it as an object that's capable of growing itself, changing its location, etc.

Practice Defining Instance Methods

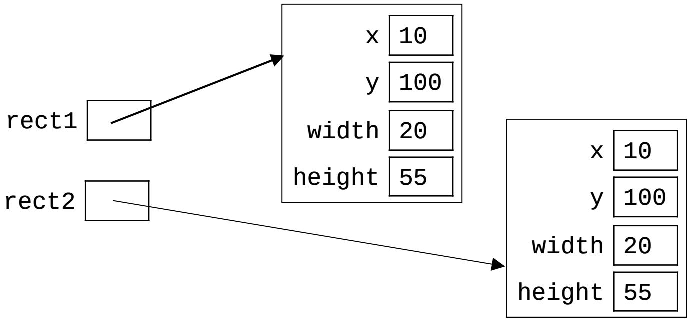
- Add a mutator method that scales the dimensions of a `Rectangle` object by a specified factor.
  - make the factor a `double`, to allow for fractional values
  - take advantage of existing mutator methods
  - use a type cast to turn the result back into an integer

- Add an accessor method that gets the perimeter of a `Rectangle` object.

Testing for Equivalent Objects

- Let's say that we have two different `Rectangle` objects, both of which represent the same rectangle:

```
Rectangle rect1 = new Rectangle(10, 100, 20, 55);
Rectangle rect2 = new Rectangle(10, 100, 20, 55);
```



- What is the value of the following condition?

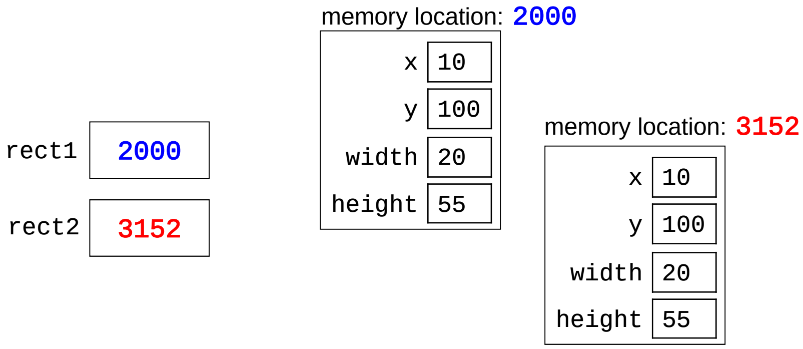
```
rect1 == rect2
```

Testing for Equivalent Objects (cont.)

- The condition

```
rect1 == rect2
```

compares the *references* stored in `rect1` and `rect2`.



memory location: 2000

x	10
y	100
width	20
height	55

memory location: 3152

x	10
y	100
width	20
height	55

rect1	2000
rect2	3152

- It doesn't compare the objects themselves.

Testing for Equivalent Objects (cont.)

- Recall: to test for equivalent objects, we need to use the `equals` method:

```
rect1.equals(rect2)
```

- Java's built-in classes have `equals` methods that:
  - return `true` if the two objects are equivalent to each other
  - return `false` otherwise

Default `equals()` Method

- If we don't write an `equals()` method for a class, objects of that class get a default version of this method.
- The default `equals()` just tests if the memory addresses of the two objects are the same.
  - the same as what == does!
- To ensure that we're able to test for equivalent objects, we need to write our own `equals()` method.

`equals()` Method for Our `Rectangle` Class

```
public boolean equals(Rectangle other) {
    if (other == null) {
        return false;
    } else if (this.x != other.x) {
        return false;
    } else if (this.y != other.y) {
        return false;
    } else if (this.width != other.width) {
        return false;
    } else if (this.height != other.height) {
        return false;
    } else {
        return true;
    }
}
```

- **Note:** The method is able to access the fields in `other` directly (without using accessor methods).
- Instance methods can access the private fields of *any* object from the same class as the method.

equals() Method for Our Rectangle Class (cont.)

- Here's an alternative version:

```
public boolean equals(Rectangle other) {
    return (other != null
         && this.x == other.x
         && this.y == other.y
         && this.width == other.width
         && this.height == other.height);
}
```

Converting an Object to a String

- The `toString()` method allows objects to be displayed in a human-readable format.
    - it returns a string representation of the object
- This method is called implicitly when you attempt to print an object or when you perform string concatenation:

```
Rectangle r1 = new Rectangle(10, 20, 100, 80);
System.out.println(r1);

// the second line above is equivalent to:
System.out.println(r1.toString());
```

- If we don't write a `toString()` method for a class, objects of that class get a default version of this method.
    - here again, it usually makes sense to write our own version

toString() Method for Our Rectangle Class

```
public String toString() {
    return this.width + " x " + this.height;
}
```

- Note: the method does not do any printing.
- It returns a String that can then be printed.

Revised Client Program

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        System.out.println("r1: " + r1);
        System.out.println("area = " + r1.area());

        System.out.println("r2: " + r2);
        System.out.println("area = " + r2.area());

        // grow both rectangles
        r1.grow(50, 10);
        r2.grow(5, 30);

        System.out.println("r1: " + r1);
        System.out.println("r2: " + r2);
    }
}
```

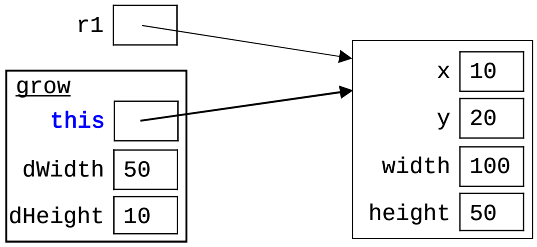
Conventions for Accessors and Mutators

- Accessors:
  - usually have no parameters
    - all of the necessary info. is inside the called object
  - have a non-`void` return type
  - often have a name that begins with "get" or "is"
    - examples: `getWidth()`, `isSquare()`
    - but not always: `area()`, `perimeter()`

- Mutators:
  - usually have one or more parameter
  - usually have a `void` return type
  - often have a name that begins with "set"
    - examples: `setLocation()`, `setWidth()`
    - but not always: `grow()`, `scale()`

The Implicit Parameter and Method Frames

- When we call an instance method, the implicit parameter is included in its method frame.
  - example: `r1.grow(50, 10)`



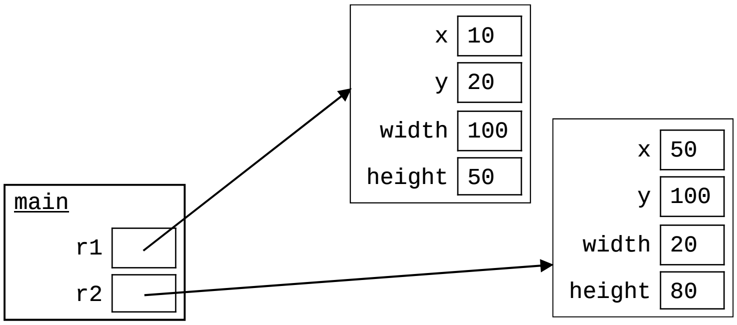
- The method uses `this` to access the fields in the called object.
  - even if the code doesn't explicitly use it

```
width += dWidth;         →    this.width += dWidth;
height += dHeight;            this.height += dHeight;
```

Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

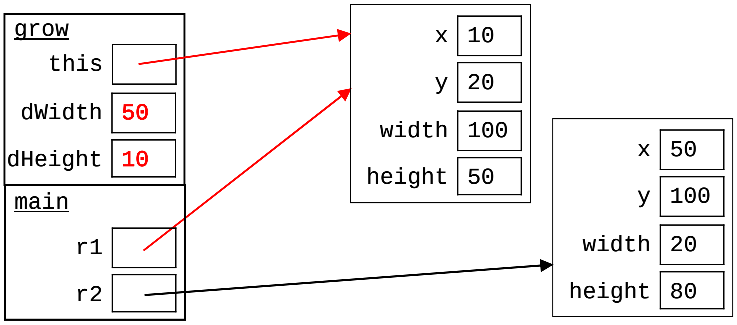
- After the objects are created:



Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

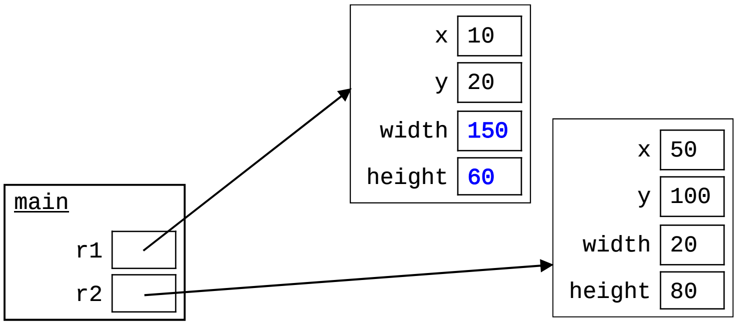
- During the method call `r1.grow(50, 10)`:



Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

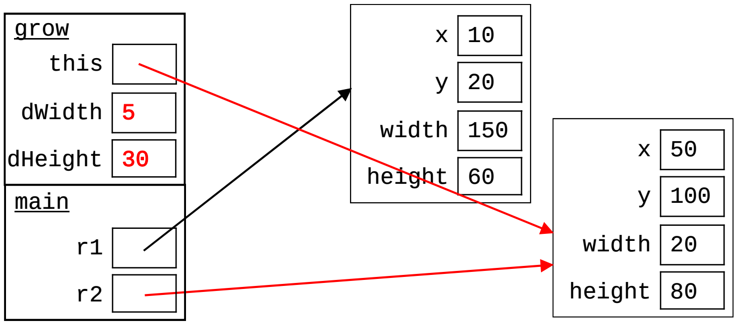
- After the method call `r1.grow(50, 10)`:



Example: Method Frames for Instance Methods

```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

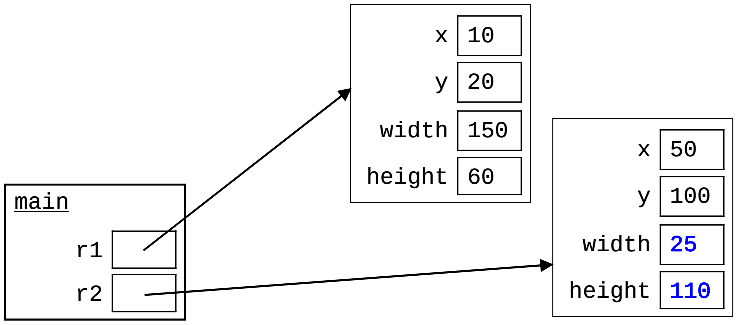
- During the method call `r2.grow(5, 30)`:



Example: Method Frames for Instance Methods

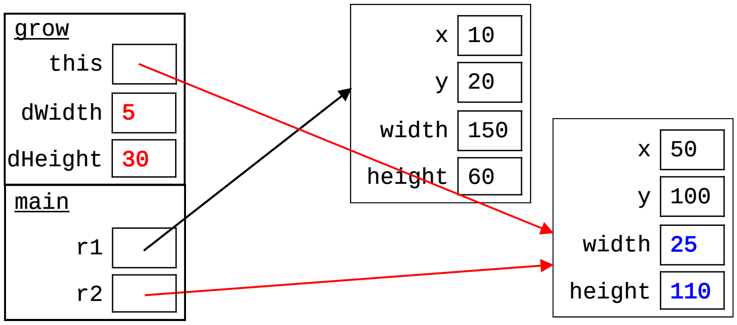
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

- After the method call `r2.grow(5, 30)`:



Why Mutators Don't Need to Return Anything

- A mutator operates directly on the called object, so any changes it makes will be there after the method returns.
  - example: the call `r2.grow(5, 30)` from the last slide



  - during this call, `grow` gets a copy of the reference in `r2`, so it changes the object to which `r2` refers

Variable Scope: Static vs. Non-Static Methods

```
public class Foo {
    private int a;

    public static int bar(int b, int c, Foo f) {
        c = c + this.a;         // would not compile
        return 3*b + f.a;       // would compile
    }

    public int boo(int d, Foo f) {
        d = d + this.a + f.a;   // would compile
        return 2 * d;
    }
}
```

- Static methods (like `bar` above) do *NOT* have a called object, so they can't access its fields.
- Instance/non-static methods (like `boo` above) *do* have a called object, so they *can* access its fields.
- *Any* method of a class can access fields in an object of that class that is passed in as a parameter (like the parameter `f` above).

A Common Use of the Implicit Parameter

- Here's our `setLocation` method:

```
public void setLocation(int newX, int newY) {
    if (newX < 0 || newY < 0) {
        throw new IllegalArgumentException();
    }
    this.x = newX;
    this.y = newY;
}
```

- Here's an equivalent version:

```
public void setLocation(int x, int y) {
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException();
    }
    this.x = x;
    this.y = y;
}
```

  - When the parameters have the same names as the fields, we *must* use `this` to access the fields.

Defining a Second Constructor

- Here's our `Rectangle` constructor:

```
public Rectangle(int initialX, int initialY,
  int initialWidth, int initialHeight) {
    this.setLocation(initialX, initialY);
    this.setWidth(initialWidth);
    this.setHeight(initialHeight);
}
```

- It requires four parameters:

```
Rectangle r1 = new Rectangle(10, 20, 100, 50);
```

- A class can have an arbitrary number of constructors, provided that each of them has a distinct parameter list.

Defining a Second Constructor (cont.)

- Here's a constructor that only takes values for width and height:

```
public Rectangle(int width, int height) {
    this.setWidth(width);
    this.setHeight(height);
    this.x = 0;
    this.y = 0;
}
```

  - it puts the rectangle at the location (0, 0)

- Equivalently, we can call the original constructor, and let it perform the actual assignments:

```
public Rectangle(int width, int height) {
    this(0, 0, width, height);  // call other constr.
}
```

  - we use the keyword `this` instead of `Rectangle`
  - this is the way that one constructor calls another

Practice Exercise: Writing Client Code

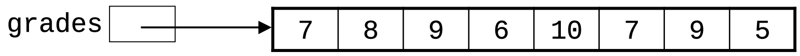
- Write a static method called `processRectangle()` that:
  - takes a `Rectangle` object (call it `rect`) and an integer (call it `amount`) as parameters
  - prints the existing dimensions and area of the `Rectangle` (*hint:* take advantage of the `toString()` method)
  - increases both of the `Rectangle`'s dimensions by the specified amount
  - prints the new dimensions and area
- You should assume that the method a true client – i.e., it is *not* part of the `Rectangle` class.

Collections of Data

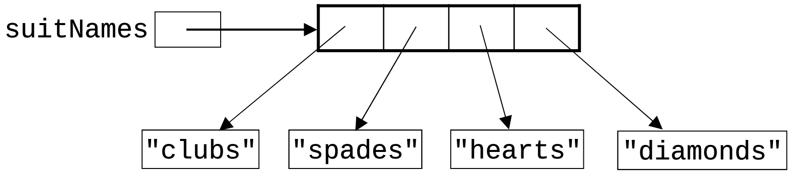
- There are many situations in which we need a program to maintain a collection of data.
- Examples include:
  - all of the grades on a given assignment/exam
  - a simple database of song info (e.g., in a music player)

Using an Array for a Collection

- We've used an array to maintain a collection of primitive data values.



- It's also possible to have an array of objects:



A Class for a Collection

- Rather than just using an array, it's often helpful to create a blueprint class for the collection.
- Example: a `GradeSet` class for a collection of grades from a single assignment or exam
  - possible field definitions:

```
public class GradeSet {
    private String name;
    private int possiblePoints;
    private double[] grades;
    private int gradeCount;
```

- The array of values is "inside" the collection object, along with other relevant information associated with the collection.
- In addition, we would add methods for maintaining and processing the collection.

A Blueprint Class for Grade Objects

- Rather than just representing the grades as `int`s or `double`s, we'll use a separate blueprint class for a single grade:

```
public class Grade {
    private double rawScore;
    private int latePenalty;  // as a percent
```

- This allows us to store both the raw score and the late penalty (if any).

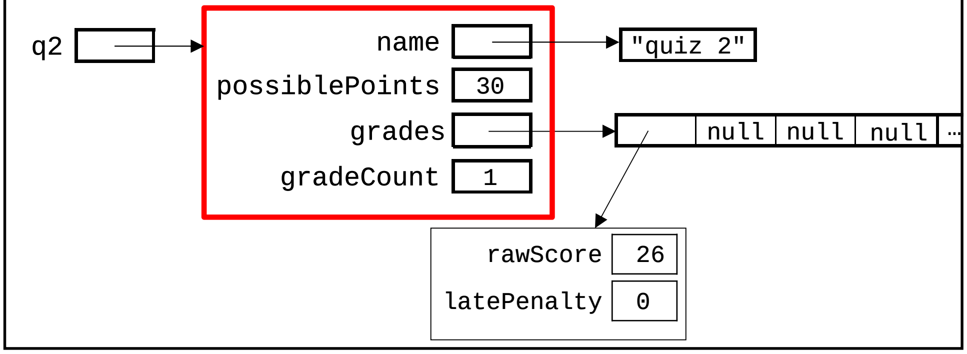
- Constructors and methods include:

```
Grade(double raw, int late)
Grade(double raw)
getRawScore()
getLatePenalty()
setRawScore(double newScore)
setLatePenalty(int newPenalty)
getAdjustedScore()   // with late penalty
```

Revised `GradeSet` Class

```
public class GradeSet {
    private String name;
    private int possiblePoints;
    private Grade[] grades;
    private int gradeCount;
```

- Here's what one of these objects would look like in memory:



GradeSet Constructor/Methods

- Constructor:
  ```
  GradeSet(String name, int possPts, int numGrades)
  ```

- Accessor methods:
  ```
  String getName()
  int getPossiblePoints()
  int getGradeCount()
  Grade getGrade(int i) // get grade at position i
  double averageGrade(boolean includePenalty)
  ```

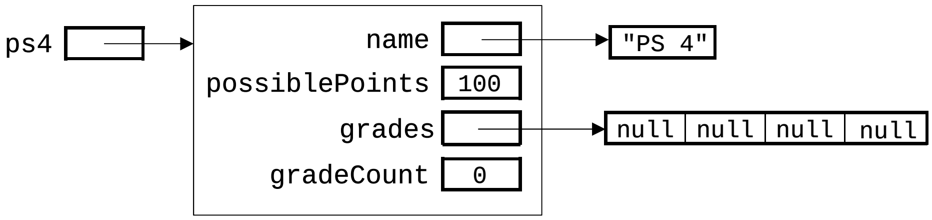
- Mutator methods:
  ```
  void setName(String name)
  void setPossiblePoints(int possPoints)
  void addGrade(Grade g)
  Grade removeGrade(int i) // remove grade at posn i
  ```

- Let's review the code for these, and write some of them together.

GradeSet Constructor/Methods

GradeSet Constructor/Methods

GradeSet: Adding a Grade



ps4

name

possiblePoints 100

grades

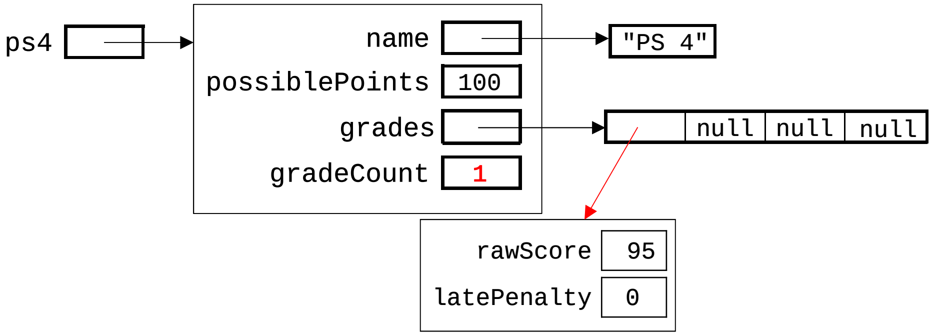
gradeCount 0

"PS 4"

null null null null

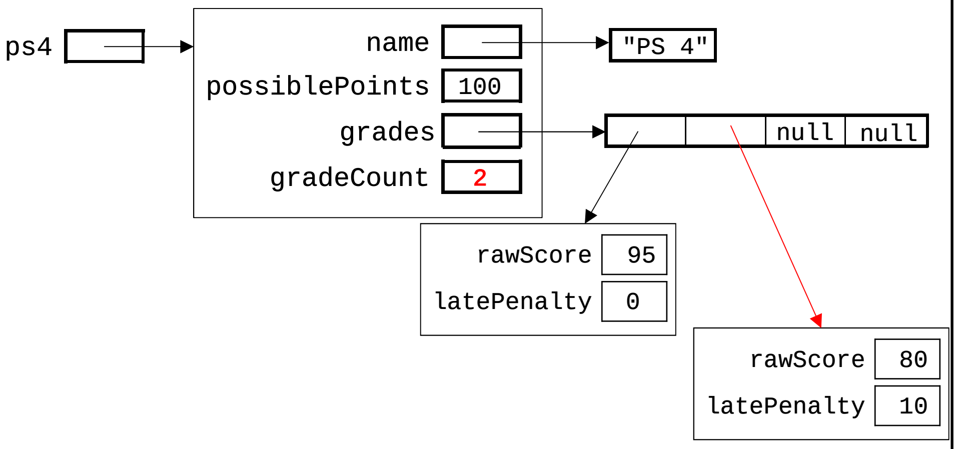
```
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));
```

GradeSet: Adding a Grade



```
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));
```

GradeSet: Adding a Grade



```
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));
```

Unit 5, Part 2

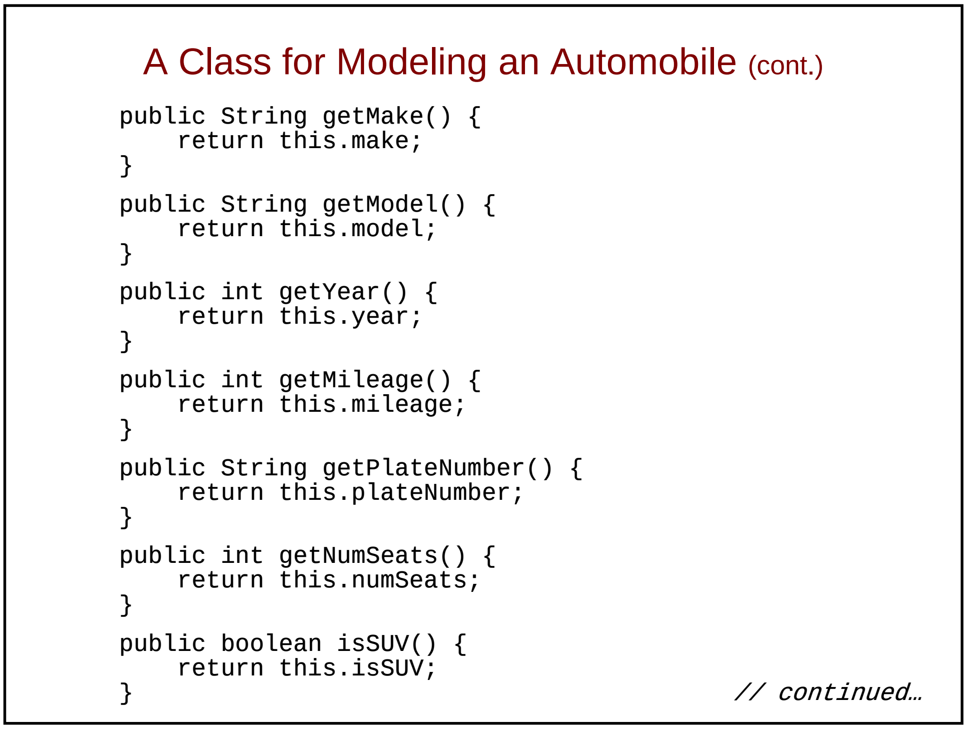
Inheritance and Polymorphism

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

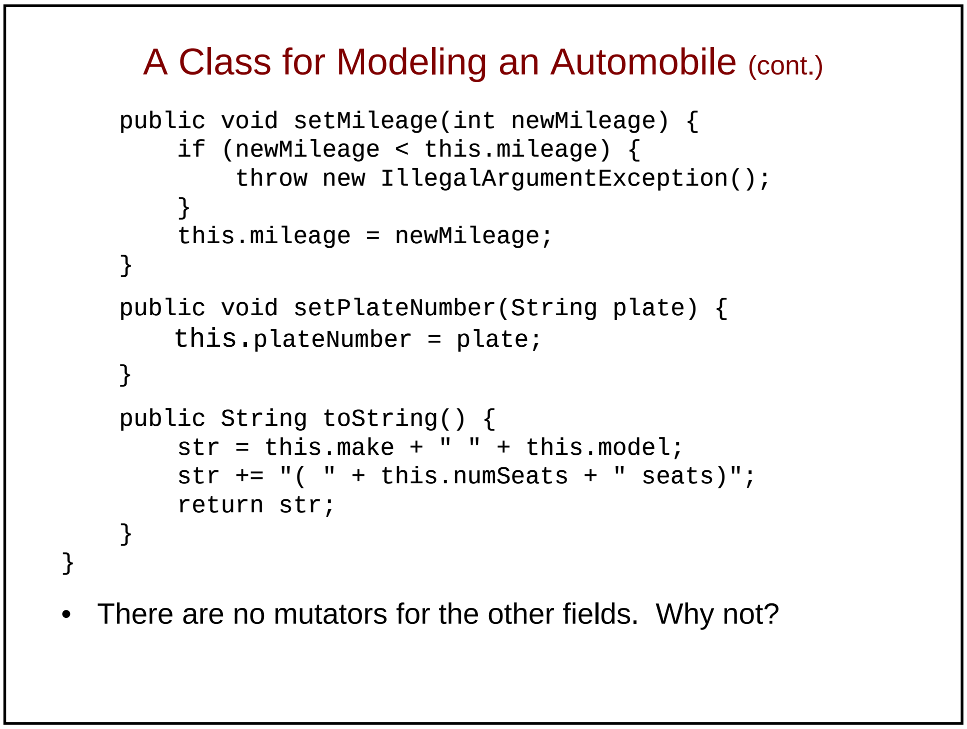
A Class for Modeling an Automobile

```
public class Automobile {
    private String make;
    private String model;
    private int year;
    private int mileage;
    private String plateNumber;
    private int numSeats;
    private boolean isSUV;

    public Automobile(String make, String model, int year,
      int numSeats, boolean isSUV) {
        this.make = make;
        this.model = model;
        if (year < 1900) {
           throw new IllegalArgumentException();
        }
        this.year = year;
        this.numSeats = numSeats;
        this.isSUV = isSUV;
        this.mileage = 0;
        this.plateNumber = "unknown";
    }

    public Automobile(String make, String model, int year) {
        this(make, model, year, 5, false);
    }                                           // continued…
```



A Class for Modeling an Automobile (cont.)

```
    public String getMake() {
        return this.make;
    }
    public String getModel() {
        return this.model;
    }
    public int getYear() {
        return this.year;
    }
    public int getMileage() {
        return this.mileage;
    }
    public String getPlateNumber() {
        return this.plateNumber;
    }
    public int getNumSeats() {
        return this.numSeats;
    }
    public boolean isSUV() {
        return this.isSUV;
    }                                                  // continued…
```



A Class for Modeling an Automobile (cont.)

```
    public void setMileage(int newMileage) {
        if (newMileage < this.mileage) {
            throw new IllegalArgumentException();
        }
        this.mileage = newMileage;
    }
    public void setPlateNumber(String plate) {
        this.plateNumber = plate;
    }
    public String toString() {
        str = this.make + " " + this.model;
        str += "( " + this.numSeats + " seats)";
        return str;
    }
}
```

- There are no mutators for the other fields. Why not?

Modeling a Related Class

- What if we now want to write a class to represent a taxi?
- The `Taxi` class will have the same fields and methods as the `Automobile` class.
- It will also have its own fields and methods:

`taxiID`	`getID, setID`
`fareTotal`	`getFareTotal, addFare`

`numFares`	`getNumFares, getAverageFare`
	`resetFareInfo`

- We may also want the `Taxi` versions of some of the `Automobile` methods to behave differently. Examples:
  - we may want the `toString` method to include values from different fields
  - we may want the `getNumSeats` method to return only the number of seats available for passengers

Inheritance

- To avoid redefining all of the `Automobile` fields and methods, we specify that the `Taxi` class *extends* the `Automobile` class:

```
public class Taxi extends Automobile {
```

- The `Taxi` class will *inherit* the fields and methods of the `Automobile` class.
  - it doesn't have to redefine them

A Class for Modeling a Taxi

```
public class Taxi extends Automobile {
    // We don't need to include the fields
    // from Automobile!
    private String taxiID;
    private double fareTotal;
    private int numFares;

    // constructor goes here...

    // We don't need to include the methods
    // from Automobile!

    public String getID() {
        return this.taxiID;
    }

    public void addFare(double fare) {
        if (fare < 0) {
            throw new IllegalArgumentException();
        }
        this.fareTotal += fare;
        this.numFares++;
    }
    ...
```

Using Inherited Methods

- Because `Taxi` extends `Automobile`, we can invoke a method defined in the `Automobile` class on a `Taxi` object.
    - example:

```
Taxi t = new Taxi(…);
t.setMileage(25000);
```

- This works even though there is no `setMileage` method defined in the `Taxi` class!
    - `Taxi` inherits it from `Automobile`

Overriding an Inherited Method

- A subclass can *override* an inherited method, replacing it with its own version.
- To override a method, the new method must have the same:
  - return type
  - name
  - number and types of parameters
- Example: our `Taxi` class can define its own `toString` method:

```
public String toString() {
    return "Taxi (id = " + this.taxiID + ")";
}
```

  - it overrides the `toString` method inherited from `Automobile`

Rethinking Our Design

- What if we also want to be able to capture information about other types of vehicles?
  - motorcycles
  - trucks
- The classes for these other vehicles should *not* inherit from `Automobile`. Why not?
- Solution: define a `Vehicle` class
  - fields and methods common to all vehicles are defined there
  - leave automobile-specific state and behavior in `Automobile`
    - everything else is inherited from `Vehicle`
  - define `Motorcycle` and `Truck` classes that also inherit from `Vehicle`

A Class for Modeling a Vehicle

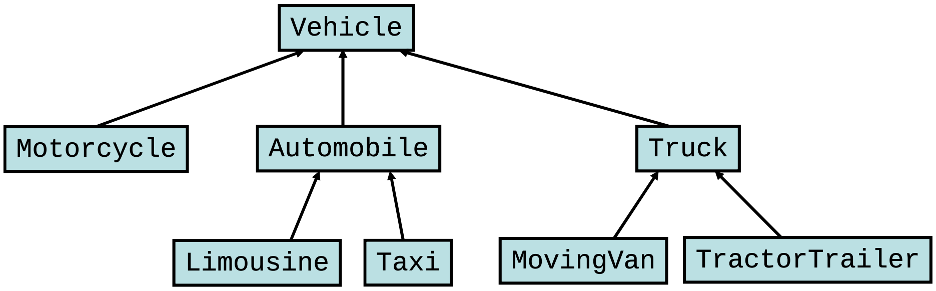
```
public class Vehicle {
    private String make;
    private String model;
    private int year;
    private int mileage;
    private String plateNumber;
    private int numWheels;  // this was not in Automobile

    public Vehicle(String make, String model, int year,
      int numWheels) {
        this.make = make;
        this.model = model;
        if (year < 1900) {
           throw new IllegalArgumentException();
        }
        this.year = year;
        this.numWheels = numWheels;
        this.mileage = 0;
        this.plateNumber = "unknown";
    }

    public String getMake() {
        return this.make;
    }

    // etc.
```

Revised Automobile Class

```
public class Automobile extends Vehicle {
    // make, model, etc. are now inherited from Vehicle

    // The following are specific to automobiles,
    // so we leave them here.
    private int numSeats;
    private boolean isSUV;

    // constructor goes here...

    // getMake(), etc. are now inherited from Vehicle

    // The following are specific to automobiles,
    // so we leave them here.
    public int getNumSeats() {
        return this.numSeats;
    }

    public boolean isSUV() {
        return this.isSUV;
    }
    ...
}
```

Inheritance Hierarchy

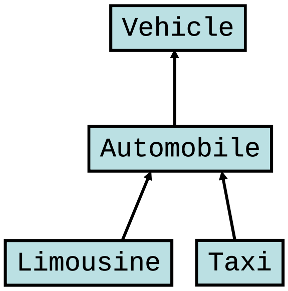
- Inheritance leads classes to be organized in a *hierarchy*:



- A class in Java inherits *directly* from at most one class.
- However, a class can inherit *indirectly* from a class higher up in the hierarchy.
  - example: `Taxi` inherits indirectly from `Vehicle`

Terminology

- When class C extends class D (directly or indirectly):
  - class D is known as a *superclass* or *base class* of C
    - super – comes *above* it in the hierarchy
  - class C is known as a *subclass* or *derived class* of D
    - sub – comes *below* it in the hierarchy
- Examples:
  - `Automobile` is a superclass of `Taxi and Limosine`
  - `Taxi` is a subclass of `Automobile and Vehicle`



Deciding Where to Define a Method

- Assume we only care about the number of axles in truck vehicles.
- Thus, we define the `getNumAxles` method in the `Truck` class, rather than in the `Vehicle` class.

```
public int getNumAxles() {
    return this.getNumWheels() / 2;
}
```

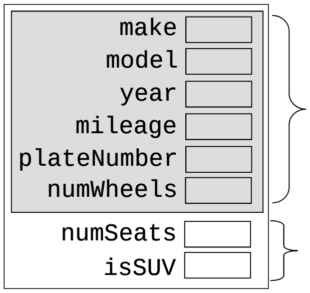
  - it will be inherited by subclasses of `Truck`
  - it won't be available to non-truck subclasses of `Vehicle`

- We override this method in the `TractorTrailer` class, because tractor trailers have four wheels on all but the front axle:

```
public int getNumAxles() {
    int numBackWheels = this.getNumWheels() – 2;
    return 1 + numBackWheels/4;
}
```

What is Accessible From a Superclass?

- A subclass has direct access to the *public* fields and methods of a superclass.
- A subclass does <u>not</u> have direct access to the *private* fields and methods of a superclass.
- Example: we can think of an `Automobile` object as follows:



private fields inherited from `Vehicle`. They *cannot* be accessed directly by methods in `Automobile`.

fields defined in `Automobile`. They *can* be accessed directly by methods in `Automobile`.

What is Accessible From a Superclass? (cont.)

- Example: now that `make` and `model` are defined in `Vehicle`, we're no longer able to access them directly in the `Automobile` version of `toString`:

won't compile

```
public String toString() {
    String str = this.make + " " + this.model;
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

- Instead, we need to make method calls to access the inherited fields:

```
public String toString() {
    String str = this.getMake() + " " +
      this.getModel();
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

What is Accessible From a Superclass? (cont.)

- Faulty approach: redefine the inherited fields in the subclass

```
public class Vehicle {
    private String make;
    private String model;
    …
}

public class Automobile extends Vehicle {
    private String make;   // NOT a good idea!
    private String model;
    …
}
```

- You should NOT do this!

Writing a Constructor for a Subclass

- Another example of illegally accessing inherited private fields:

```
public Automobile(String make, String model, int year,
  int numSeats, boolean isSUV) {
     this.make = make;
     this.model = model;
     ...
}
```

- To initialize inherited fields, a constructor should invoke a constructor from the superclass.

```
public Automobile(String make, String model, int year,
  int numSeats, boolean isSUV) {
    super(make, model, year, 4);   // 4 is for numWheels
    this.numSeats = numSeats;
    this.isSUV = isSUV;
}
```

  - use the keyword `super` followed by appropriate parameters for the superclass constructor
  - must be done as the very first line of the constructor

Writing a Constructor for a Subclass (cont.)

- If a subclass constructor doesn't explicitly invoke a superclass constructor, the compiler tries to insert a call to the superclass constructor with no parameters.

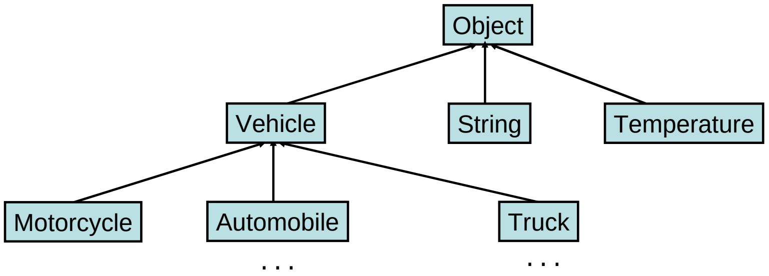
- If there isn't such a constructor, we get a compile-time error.
  - example: this constructor won't compile:

```
public Taxi(String make, String model, int year, String ID)
{
    this.taxiID = ID;
}
```

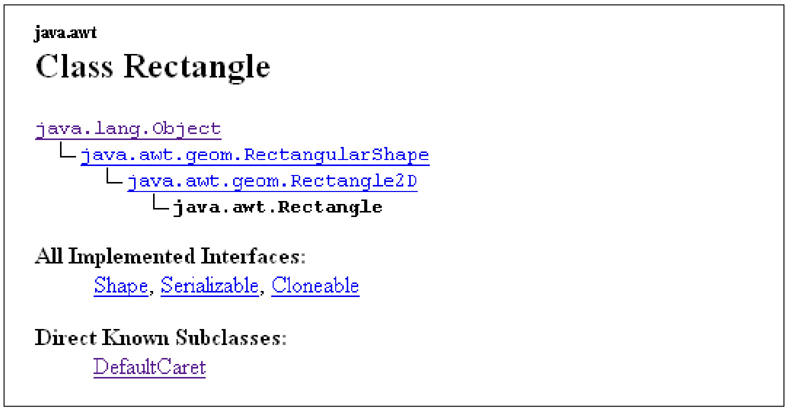
  - the compiler attempts to insert the following call:
    `super();`
  - there isn't an `Automobile` constructor with no parameters

The `Object` Class

- If a class doesn't explicitly extend another class, it implicitly extends a special class called `Object`.
- Thus, the `Object` class is at the top of the class hierarchy.
  - *all* classes are subclasses of this class
  - the default `toString` and `equals` methods are defined in this class



Inheritance in the Java API



java.awt

Class Rectangle

```
java.lang.Object
  └java.awt.geom.RectangularShape
      └java.awt.geom.Rectangle2D
          └java.awt.Rectangle
```

All Implemented Interfaces:

Shape, Serializable, Cloneable

Direct Known Subclasses:

DefaultCaret

More Examples of Method Overriding

- `Vehicle` inherits the fields and methods of `Object`.
- The inherited `toString` method isn't very helpful.
- We define a `Vehicle` version that overrides the inherited one:

```
public String toString() {   // Vehicle version
    String str = this.make + " " + this.model;
    return str;
}
```

- When `toString` is invoked on a `Vehicle` object,
  the `Vehicle` version is executed:

```
Vehicle v = new Vehicle("Radio Flyer",
  "Classic Tricycle", 2002, 3);
System.out.println(v);
```

outputs: `Radio Flyer Classic Tricycle`

More Examples of Method Overriding (cont.)

- The `Automobile` class inherits the `Vehicle` version of `toString`.
- If we didn't define a `toString()` method in `Automobile`, the inherited version would be used.
- The `Automobile` version overrides the `Vehicle` version so that the number of seats can be included in the string:

```
public String toString() {
    String str = this.getMake() + " " +
      this.getModel();
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

Invoking an Overriden Method

- When a subclass overrides an inherited method, we can invoke the inherited version by using the keyword `super`.

- Example: the `Automobile` version of `toString()` begins with the same fields as the `Vehicle` version:

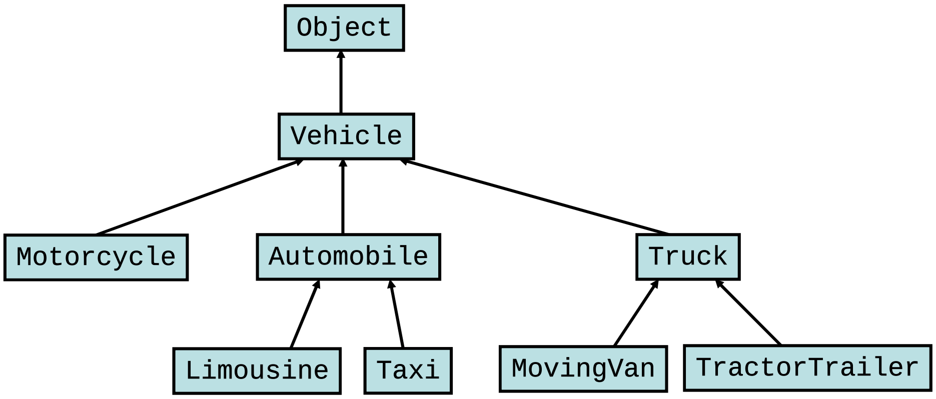
```
public String toString() {
    String str = this.getMake() + " " +
      this.getModel();
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

  - instead of calling the accessor methods, we can do this:

```
public String toString() {
    String str = super.toString();
    str += " ( " + this.numSeats + " seats)";
    return str;
}
```

is-a Relationships

- We use inheritance to capture *is-a* relationships.
  - an automobile *is a* vehicle
  - a taxi *is an* automobile
  - a tractor trailer *is a* truck



has-a Relationships

- Another type of relationship is a *has-a* relationship.
  - one type of object "owns" another type of object
  - example: a driver *has a* vehicle
- Inheritance should not be used to capture has-a relationships.
  - it does not make sense to make the `Driver` class a subclass of `Vehicle`
- Instead, we give the "owner" object a field that refers to the "owned" object:

```
public class Driver {
    String name;
    String ID;
    Vehicle v;
    ...
```

Polymorphism

- We've been using reference variables like this:

```
Automobile a = new Automobile("Ford", "Model T", …);
```

  - variable `a` is declared to be of type `Automobile`
  - it holds a reference to an `Automobile` object
- In addition, a reference variable of type T can hold a reference to an object from a *subclass* of T:

```
Automobile a = new Taxi("Ford", "Tempo", …);
```

  - this works because `Taxi` is a subclass of `Automobile`
  - a taxi is an automobile!
- The name for this feature of Java is *polymorphism*.
  - from the Greek for “many forms”
  - the same code can be used with objects of different types!

Polymorphism and Collections of Objects

- Polymorphism is useful when we have a collection of objects of different but related types.

- Example:
  - let's say that a company has a collection of vehicles of different types
  - we can store all of them in an array of type `Vehicle`:

```
Vehicle[] fleet = new Vehicle[5];
fleet[0] = new Automobile("Honda", "Civic", …);
fleet[1] = new Motorcycle("Harley", ...);
fleet[2] = new TractorTrailer("Mack", ...);
fleet[3] = new Taxi("Ford", …);
fleet[4] = new Truck("Dodge", …);
```

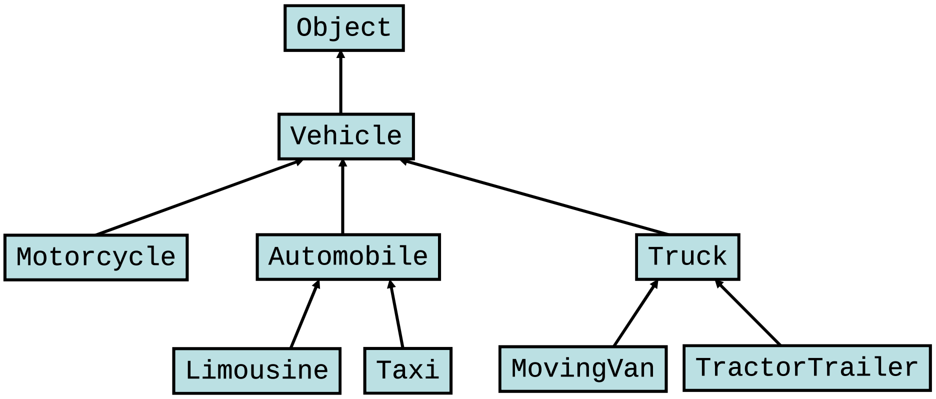
Processing a Collection of Objects

- We can determine the average age of the vehicles in the company's fleet by doing the following:

```
int totalAge = 0;
for (int i = 0; i < fleet.length; i++) {
    int age = CURRENT_YEAR – fleet[i].getYear();
    totalAge += age;
}
double averageAge = (double)totalAge / fleet.length;
```

- We can invoke `getYear()` on each object in the array, regardless of its type.
  - their classes are all subclasses of `Vehicle`
  - thus, they must all have a `getYear()` method

Practice with Polymorphism


Object
Vehicle
Motorcycle
Automobile
Truck
Limousine
Taxi
MovingVan
TractorTrailer

- Which of these assignments would be allowed?

```
Vehicle v1 = new Motorcycle(…);
TractorTrailer t1 = new Truck(…);
Truck t2 = new MovingVan(…);
Taxi t3 = new Automobile(…);
Vehicle v2 = new TractorTrailer(…);
MovingVan m1 = new TractorTrailer(…);
```

Declared Type vs. Actual Type

- An object's declared type may not match its actual type:
  - declared type: type specified when declaring a variable
  - actual type: type specified when creating an object
- Recall this client code:

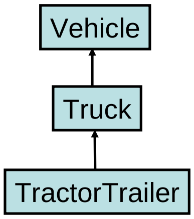
```
Vehicle[] fleet = new Vehicle[5];
fleet[0] = new Automobile("Honda", "Civic", 2005);
fleet[1] = new Motorcycle("Harley", …);
fleet[2] = new TractorTrailer("Mack", …);
```

- Here are the types:

object	declared type	actual type
fleet[0]	Vehicle	Automobile
fleet[1]	Vehicle	Motorcycle
fleet[2]	Vehicle	TractorTrailer

Determining if a Method Call is Valid

- The compiler uses the *declared* type of an object to determine if a method call is valid.
  - starts at the declared type, and goes up the inheritance hierarchy as needed looking for a version of the method
  - if it can't find a version, the method call will *not* compile



- Example: the following would *not* work:

```
Vehicle[] fleet = new Vehicle[5];
...
fleet[2] = new TractorTrailer("Mack",…);
...
System.out.println(fleet[2].getNumAxles());
```

  - the declared type of `fleet[2]` is `Vehicle`
  - there's no `getNumAxles()` method defined in or inherited by `Vehicle`

Determining if a Method Call is Valid (cont.)

- In such cases, we can use casting to create a reference with the necessary declared type:

```
Vehicle[] fleet = new Vehicle[5];
...
fleet[2] = new TractorTrailer("Mack", …);
...
TractorTrailer t = (TractorTrailer)fleet[2];
```

- The following *will* work:

```
System.out.println(t.getNumAxles());
```

  - the declared type of `t` is `TractorTrailer`
  - there is a `getNumAxles()` method defined in `TractorTrailer`, so the compiler is happy

Determining Which Method to Execute

- `Truck` also has a `getNumAxles` method, so this would be another way to handle the previous problem:

```
Vehicle[] fleet = new Vehicle[5];
...
fleet[2] = new TractorTrailer("Mack", …);
...
Truck t2 = (Truck)fleet[2];
System.out.println(t2.getNumAxles());
```

- The object represented by `t2` has:
  - a declared type of ________________
  - an actual type of _________________
- Both `Truck` and `TractorTrailer` have a getNumAxles. Which version will be executed?
- More generally, how does the interpreter decide which version of a method should be used?

Dynamic Binding

- At runtime, the Java interpreter selects the version of a method that is appropriate to the *actual* type of the object.
  - starts at the actual type, and goes up the inheritance hierarchy as needed until it finds a version of the method
  - known as *dynamic binding*
- Given the code from the previous slide

```
Vehicle[] fleet = new Vehicle[5]
...
fleet[2] = new TractorTrailer("Mack", …);
...
Truck t2 = (Truck)fleet[2];
System.out.println(t2.getNumAxles());
```

the `TractorTrailer` version of `getNumAxles` would be run

  - `TractorTrailer` is the actual type of `t2`, and that class has its own version of `getNumAxles`

Dynamic Binding (cont.)

- Another example:

```
public static void printFleet(Vehicle[] fleet) {
    for (int i = 0; i < fleet.length; i++) {
        System.out.println(fleet[i]);
    }
}
```

  - the `toString()` method is implicitly invoked on each element of the array when we go to print it.
  - the appropriate version is selected by dynamic binding
  - note: the selection must happen at runtime, because the actual types of the objects may not be known when the code is compiled

Dynamic Binding (cont.)

- Recall our initialization of the array:

```
 Vehicle[] fleet = new Vehicle[5];
fleet[0] = new Automobile("Honda", "Civic", …);
fleet[1] = new Motorcycle("Harley", …);
fleet[2] = new TractorTrailer("Mack", …);
...
```

- `System.out.println(fleet[0]);` will invoke the `Automobile` version of the `toString()` method.
- `Motorcycle` does not define its own `toString()` method, so `System.out.println(fleet[1]);` will invoke the `Vehicle` version of `toString()`, which is inherited by `Motorcycle`.
- `TractorTrailer` does not define its own `toString()` but `Truck does`, so `System.out.println(fleet[2]);` will invoke the `Truck` version of `toString()`, which is inherited by `TractorTrailer`.

Dynamic Binding (cont.)

- Dynamic binding also applies to method calls on the called object that occur within other methods.
- Example: the `Truck` class has the following `toString` method:

```
public String toString() {
    String str = this.getMake() + " " +
       this.getModel();
    str = str + ", capacity = " + this.capacity;
    str = str + ", " + this.getNumAxles() + " axles";
    return str;
}
```

- The `TractorTrailer` class inherits it and does *not* override it.
- When `toString` is called on a `TractorTrailer` object:
  - this `Truck` version of `toString()` will run
  - the *`TractorTrailer`* version of `getNumAxles()` will run when the code above is executed

The Power of Polymorphism

- Recall our `printFleet` method:

```
public static void printFleet(Vehicle[] fleet) {
    for (int i = 0; i < fleet.length; i++) {
        System.out.println(fleet[i]);
    }
}
```

  - polymorphism allows this method to use a single println statement to print the appropriate info. for *any* kind of vehicle.

- Without polymorphism, we would need a large if-else-if:

```
if (fleet[i] is an Automobile) {
    print the appropriate info for Automobiles
} else if (fleet[i] is a Truck) {
    print the appropriate info for Trucks
} else if ...
```

- Polymorphism allows us to easily write code that works for more than one type of object.

Polymorphism and the `Object` Class

- The `Object` class is a superclass of every other class.
- Thus, we can use an `Object` variable to store a reference to *any* object.

```
Object o1 = "Hello World";
Object o2 = new Temperature(20, 'C');
Object o3 = new Taxi("Ford", "Tempo", 2000, "T253");
```

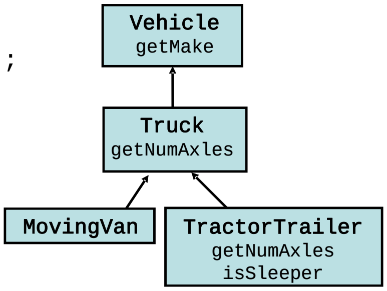
Summary and Extra Practice

- To determine if a method call is valid:
  - start at the *declared* type
  - go up the hierarchy as needed to see if you can find the specified method in the declared type *or* a superclass
  - if you don't find it, the method call is *not* valid
- Given the following:

```
TractorTrailer t1 = new TractorTrailer(…);
Vehicle v = new Truck(…);
MovingVan m = new MovingVan(…);
Truck t2 = new TractorTrailer(…);
```

- Which of the following are valid?

```
v.getNumAxles()
m.getNumAxles()
t1.getMake()
t1.isSleeper()
t2.isSleeper()
```



Summary and Extra Practice (cont.)

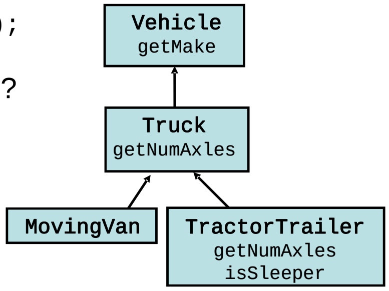
- To determine which version of a method will run (dynamic binding):
  - start at the *actual* type
  - go up the hierarchy as needed until you find the method
  - the first version you encounter is the one that will run

- Given the following:

```
TractorTrailer t1 = new TractorTrailer(…);
Vehicle v = new Truck(…);
MovingVan m = new MovingVan(…);
Truck t2 = new TractorTrailer(…);
```

- Which version of the method will run?

```
m.getNumAxles()
t1.getNumAxles()
t2.getNumAxles()
v.getMake()
t2.getMake()
```



More Practice

```
public class E extends G {
    public void method2() {
        System.out.print("E 2 ");
        this.method1();
    }
    public void method3() {
        System.out.print("E 3 ");
        this.method1();
    }
}
public class F extends G {
    public void method2() {
        System.out.print("F 2 ");
    }
}
public class G {
    public void method1() {
        System.out.print("G 1 ");
    }
    public void method2() {
        System.out.print("G 2 ");
    }
}
public class H extends E {
    public void method1() {
        System.out.print("H 1 ");
    }
}
```

More Practice (cont.)

- Which of these would compile and which would not?

```
E e1 = new E();
E e2 = new H();
E e3 = new G();
E e4 = new F();
G g1 = new H();
G g2 = new F();
H h1 = new H();
```

- To answer these questions, draw the inheritance hierarchy:

Here are the classes again…

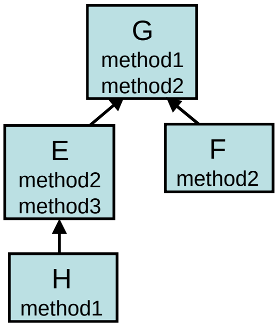
```
public class E extends G {
    public void method2() {
        System.out.print("E 2 ");
        this.method1();
    }
    public void method3() {
        System.out.print("E 3 ");
        this.method1();
    }
}
public class F extends G {
    public void method2() {
        System.out.print("F 2 ");
    }
}
public class G {
    public void method1() {
        System.out.print("G 1 ");
    }
    public void method2() {
        System.out.print("G 2 ");
    }
}
public class H extends E {
    public void method1() {
        System.out.print("H 1 ");
    }
}
```

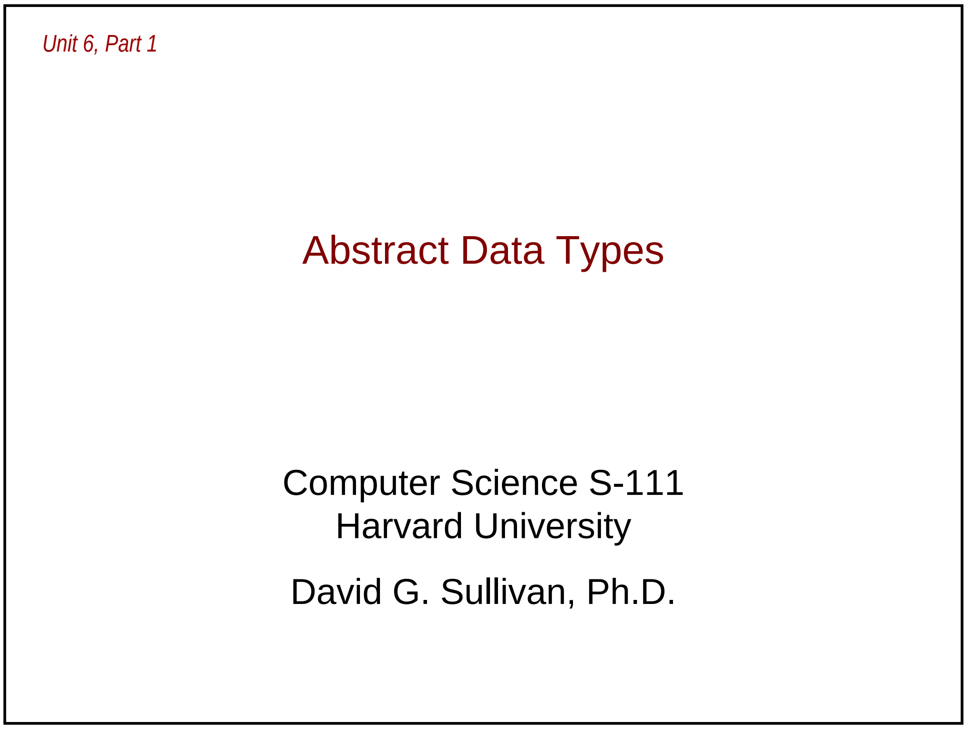
More Practice (cont.)

```
E e1 = new E();
G g1 = new H();
G g2 = new F();
```

- Which of the following would compile and which would not? For the ones that would compile, what is the output?

```
e1.method1();

e1.method2();

e1.method3();

g1.method1();

g1.method2();

g1.method3();

g2.method1();

g2.method2();

g2.method3();
```



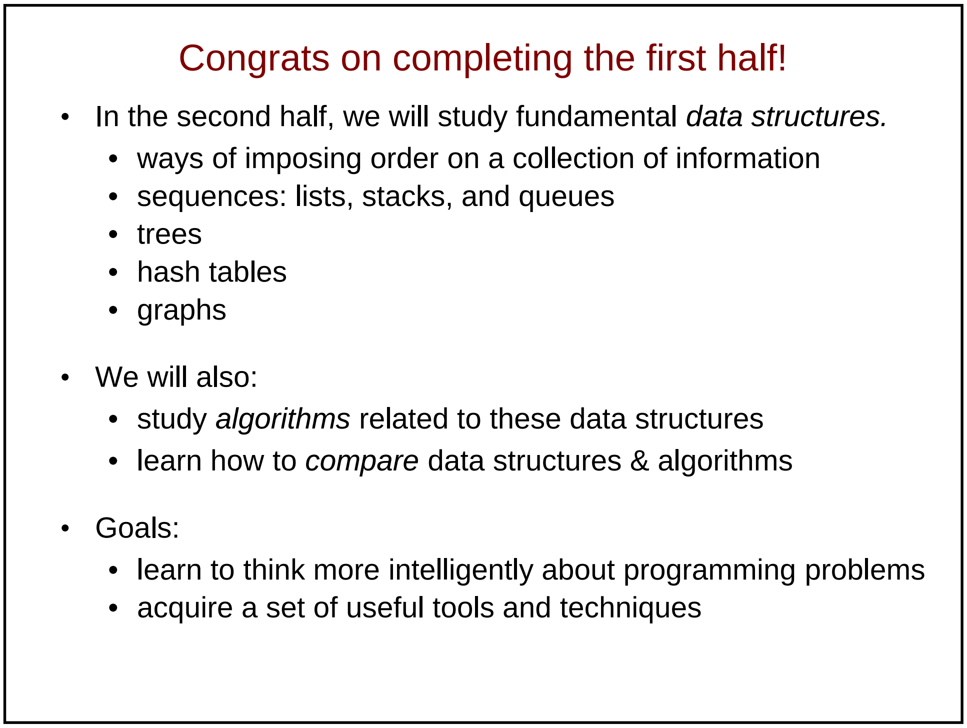


Unit 6, Part 1

Abstract Data Types

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

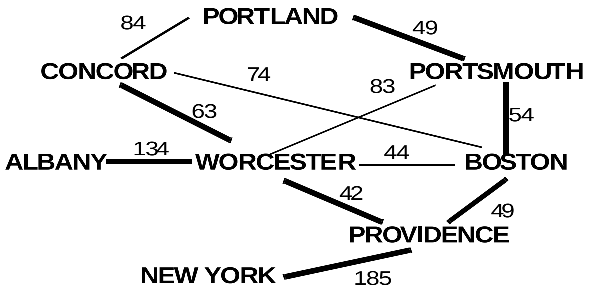


Congrats on completing the first half!

- In the second half, we will study fundamental *data structures.*
  - ways of imposing order on a collection of information
  - sequences: lists, stacks, and queues
  - trees
  - hash tables
  - graphs
- We will also:
  - study *algorithms* related to these data structures
  - learn how to *compare* data structures & algorithms
- Goals:
  - learn to think more intelligently about programming problems
  - acquire a set of useful tools and techniques

Sample Problem I: Finding Shortest Paths

- Given a set of routes between pairs of cities, determine the shortest path from city A to city B.



Sample Problem II: A Data "Dictionary"

- Given a large collection of data, how can we arrange it so that we can efficiently:
  - add a new item
  - search for an existing item
- Some data structures provide better performance than others for this application.
- More generally, we'll learn how to characterize the *efficiency* of different data structures and their associated algorithms.

Example of Comparing Algorithms

- Consider the problem of finding a phone number in a phonebook.
- Let’s informally compare the time efficiency of two algorithms for this problem.

Algorithm 1 for Finding a Phone Number

```
findNumber(person) {
    for (p = number of first page; p <= number of the last page; p++) {
        if person is found on page p {
            return the person's phone number
        }
    }
    return NOT_FOUND
}
```

- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
- What if there were 1,000,000 pages?

Algorithm 2 for Finding a Phone Number

```
findNumber(person) {
    min = the number of the first page
    max = the number of the last page
    while (min <= max) {
          mid = (min + max) / 2       // page number of the middle page
          if person is found on page mid {
               return the person's number
          } else if the person's name comes earlier in the book {
               max = mid – 1
          } else {
               min = mid + 1
          }
    }
    return NOT_FOUND
}
```

- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
- What if there were 1,000,000 pages?

Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a record in a database
- Algorithm 1 is known as *sequential search*.
  - also called *linear search*
- Algorithm 2 is known as *binary search*.
  - only works if the items in the data collection are sorted

Review: What is an Object?

- An *object* groups together:
  - one or more data values (the object's *fields* – also known as *instance variables*)
  - a set of operations that the object can perform (the object's *methods*)
- In Java, we use a *class* to define a new type of object.
  - serves as a "blueprint" for objects of that type
  - simple example:

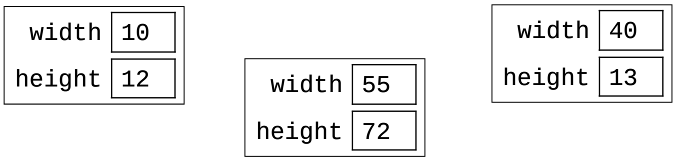
```
public class Rectangle {
    // fields
    private int width;
    private int height;

    // methods
    public int area() {
        return width * height;
    }
    …
```

Class vs. Object

- The `Rectangle` class is a blueprint:

```
public class Rectangle {
    // fields
    private int width;
    private int height;

    // methods
    ...
}
```

- `Rectangle` objects are built according to that blueprint:



(You can also think of the methods as being inside the object, but we won't show them in our diagrams.)

Creating and Using an Object

- We create an object by using the `new` operator and a special method known as a *constructor*:

  ```
  Rectangle r1 = new Rectangle(10, 30);
  ```

- Once an object is created, we can call one of its methods by using *dot notation*:

  ```
  int a1 = r1.area();
  ```

- The object on which the method is invoked is known as the *called object* or the *current object.*

Two Types of Methods

- Methods that belong to an object are referred to as *instance methods* or *non-static methods.*
  - they are invoked on an object

    ```
    int a1 = r1.area();
    ```

  - they have access to the fields of the called object

- *Static* methods do *not* belong to an object – they belong to the class as a whole.
  - they have the keyword static in their header:

    ```
    public static int max(int num1, int num2) {
        …
    ```

  - they do *not* have access to the fields of the class
  - outside the class, they are invoked using the class name:

    ```
    int result = Math.max(5, 10);
    ```

Abstract Data Types

- An *abstract data type* (ADT) is a model of a data structure that specifies:
  - the characteristics of the collection of data
  - the operations that can be performed on the collection
- It's *abstract* because it doesn't specify *how* the ADT will be implemented.
- A given ADT can have multiple implementations.

A Simple ADT: A Bag

- A bag is just a container for a group of data items.
  - analogy: a bag of candy
- The positions of the data items don't matter (unlike a list).
  - {3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
- The items do *not* need to be unique (unlike a set).
  - {7, 2, 10, 7, 5} isn't a set, but it is a bag

A Simple ADT: A Bag (cont.)

- The operations supported by our Bag ADT:
  - `add(item)`: add `item` to the `Bag`
  - `remove(item)`: remove one occurrence of `item` (if any) from the `Bag`
  - `contains(item)`: check if `item` is in the `Bag`
  - `numItems()`: get the number of items in the `Bag`
  - `grab()`: get an item at random, without removing it
    - reflects the fact that the items don't have a position (and thus we can't say "get the 5th item in the Bag")
  - `toArray()`: get an array containing the current contents of the bag
- Note that we *don't* specify *how* the bag will be implemented.

Specifying an ADT Using an Interface

- In Java, we can use an interface to specify an ADT:

```
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}
```

- An interface specifies a set of methods.
  - includes only the method headers
  - *cannot* include the actual method definitions

Implementing an ADT Using a Class

- To implement an ADT, we define a class:

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        …
}
```

- When a class header includes an `implements` clause, the class must define all of the methods in the interface.

Encapsulation

- Our implementation provides proper *encapsulation*.
- We prevent direct access to the internals of an object by making its fields *private.*

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
```

- We provide limited *indirect* access through methods that are labeled *public.*

```
    public boolean add(Object item) {
        …
```

All Interface Methods Are Public

- Methods specified in an interface *must* be `public`, so we don't need to use the keyword `public` in the interface definition.

- For example:

```
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}
```

- However, when we actually implement one of these methods in a class, we *do* need to explicitly use the keyword `public`:

```
public class ArrayBag implements Bag {
    …
    public boolean add(Object item) {
        …
```

Inheritance

- We can define a class that explicitly *extends* another class:

```
public class Animal {
    private String name;
    …
    public String getName() {
        return name;
    }
    …
}

public class Dog extends Animal {
    …
```

- We say that `Dog` is a *subclass* of `Animal`, and `Animal` is a *superclass* of `Dog`.

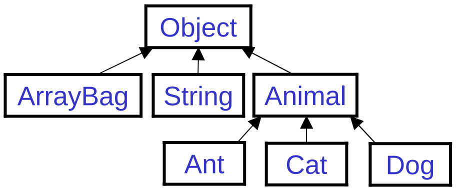
- A class *inherits* the instance variables and methods of the class that it extends.

The `Object` Class

- If a class does not explicitly extend another class, it implicitly extends Java’s `Object` class.

- The `Object` class includes methods that all classes must possess. For example:
  - `toString()`: returns a string representation of the object
  - `equals()`: is this object equal to another object?

- The process of extending classes forms a hierarchy of classes, with the `Object` class at the top of the hierarchy:



Object

ArrayBag String Animal

Ant Cat Dog

Polymorphism

- An object can be used wherever an object of one of its superclasses is called for.

- For example:

```
Animal a = new Dog();

Animal[] zoo = new Animal[100];
zoo[0] = new Ant();
zoo[1] = new Cat();
…
```

Storing Items in an `ArrayBag`

- We store the items in an array of type `Object`.

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
}
```

- This allows us to store *any* type of object in the `items` array, thanks to the power of polymorphism:

```
ArrayBag bag = new ArrayBag();
bag.add("hello");
bag.add(new Double(3.1416));
```

Another Example of Polymorphism

- An interface name can be used as the type of a variable.

```
Bag b;
```

- Variables that have an interface type can hold references to objects of any class that implements the interface.

```
Bag b = new ArrayBag();
```

- Using a variable that has the interface as its type allows us to write code that works with any implementation of an ADT.

```
public void processBag(Bag b) {
    for (int i = 0; i < b.numItems(); i++) {
      …
}
```

  - the param can be an instance of *any* `Bag` implementation
  - we must use method calls to access the object's internals, because we can't know for certain what the field names are

Memory Management: Looking Under the Hood

- In order to understand the implementation of the data structures we’ll cover in this course, you’ll need to have a good understanding of how memory is managed.
- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory.

Memory Management, Type I: Static Storage

- Static storage is used in Java for *class variables*, which are declared using the keyword `static`:

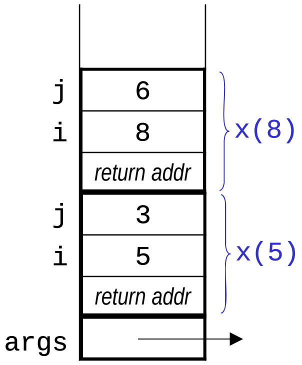
```
public static final PI = 3.1495;
public static int numCompares;
```

- There is only one copy of each class variable; it is shared by all instances (i.e., all objects) of the class.
- The Java runtime system allocates memory for class variables when the class is first encountered.
    - this memory stays fixed for the duration of the program

Memory Management, Type II: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

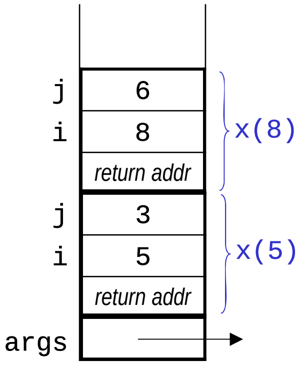
```
public class Foo {
  static void x(int i) {
      int j = i - 2;
      if (i >= 6) return;
      x(i + j);
  }
  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed. The values stored there are *not* preserved.

Stack Storage (cont.)

- Memory allocation on the stack is very efficient, because there are only two simple operations:
  - add a stack frame to the top of the stack
  - remove a stack frame from the top of the stack
- Limitations of stack storage:
  It can't be used if
  - the amount of memory needed isn't known in advance
  - we need the memory to persist after the method completes
- Because of these limitations, Java never stores arrays or objects on the stack.

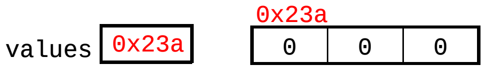


Memory Management, Type III: Heap Storage

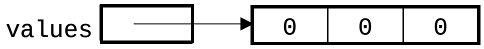
- Arrays and objects in Java are stored in a region of memory known as *the heap*.
- Memory on the heap is allocated using the `new` operator:

```
    int[] values = new int[3];
    ArrayBag b = new ArrayBag();
```

- `new` returns the memory address of the start of the array or object on the heap.
- This memory address – which is referred to as a *reference* in Java – is stored in the variable that represents the array/object:



- We will often use an arrow to represent a reference:



Heap Storage (cont.)

- In Java, an object or array persists until there are no remaining references to it.
- You can explicitly drop a reference by setting the variable equal to `null`. For example:

```
        int[] values = {5, 23, 61, 10};
        System.out.println(mean(values, 4));
        values = null;
```

- Unused objects/arrays are *automatically* reclaimed by a process known as garbage collection.
  - makes their memory available for other objects or arrays

Constructors for the `ArrayBag` Class

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        items = new Object[DEFAULT_MAX_SIZE];
        numItems = 0;
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0)
            throw new IllegalArgumentException(
              "maxSize must be > 0");
        items = new Object[maxSize];
        numItems = 0;
    }
    …
}
```

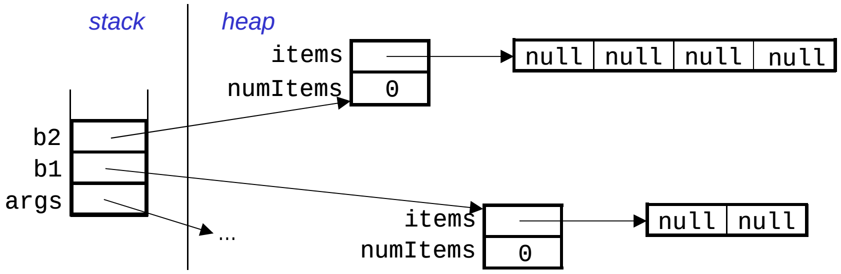
Note: In the remainder of the lecture notes, I will *not* use the implicit parameter (`this`) unless it's necessary to do so.

- If the user inputs an invalid value for `maxSize`, we throw an exception.

Example: Creating Two `ArrayBag` Objects

```
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    …
}
```

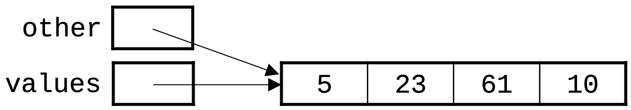
- After the objects have been created, here's what we have:



Copying References

- A variable that represents an array or object is known as a *reference variable*.
- Assigning the value of one reference variable to another reference variable copies the reference to the array or object. It does *not* copy the array or object itself.

```
int[] values = {5, 23, 61, 10};
int[] other = values;
```



- Given the lines above, what will the lines below output?

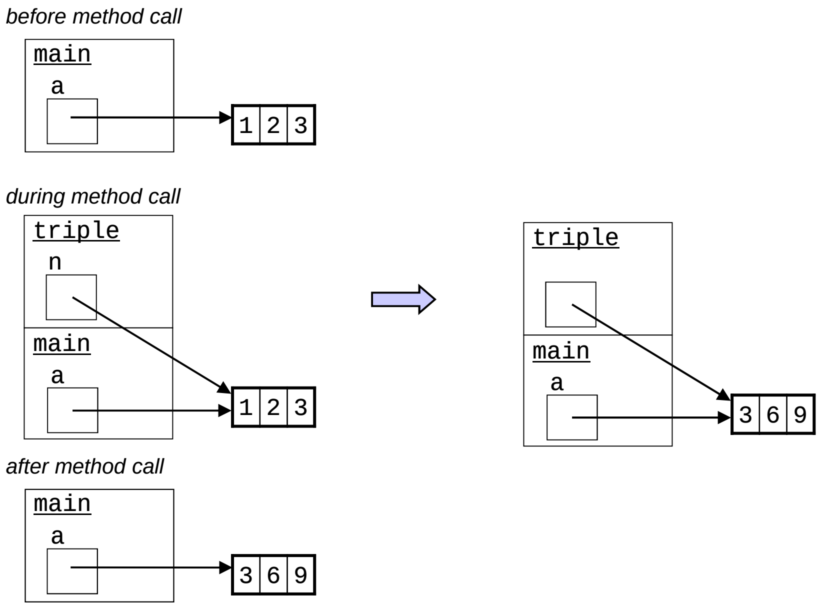
```
other[2] = 17;
System.out.println(values[2] + " " + other[2]);
```

Passing an Object/Array to a Method

- When a method is passed an object or array as a parameter, the method gets a copy of the *reference* to the object or array, *not* a copy of the object or array itself.
- Thus, any changes that the method makes to the object/array will still be there when the method returns.
- Consider the following:

```
public static void main(String[] args) {
    int[] a = {1, 2, 3};
    triple(a);
    System.out.println(Arrays.toString(a));
}

public static void triple(int[] n) {
    for (int i = 0; i < n.length; i++) {
        n[i] = n[i] * 3;
    }
}
```

Passing an Object/Array to a Method (cont.)



before method call

during method call

after method call

A Method for Adding an Item to a Bag

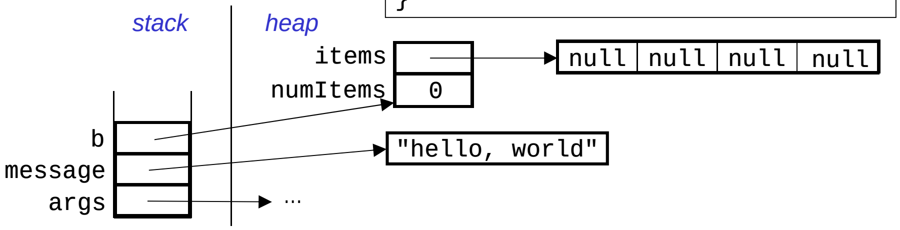
```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        if (item == null)
            throw new IllegalArgumentException();
        if (numItems == items.length)
            return false;  // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

- `add()` is an instance method (a.k.a. a non-static method), so it has access to the fields of the current object.

Example: Adding an Item

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

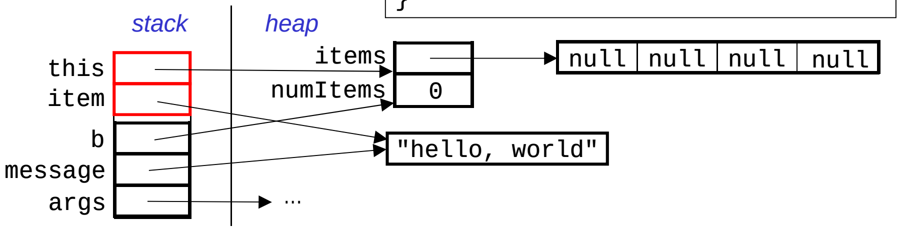
```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```



Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

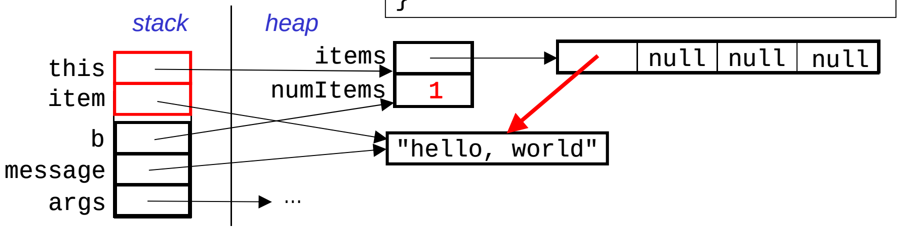


- add's stack frame includes:
  - `item`, which stores a copy of the reference passed as a param.
  - `this`, which stores a reference to the called/current object

Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

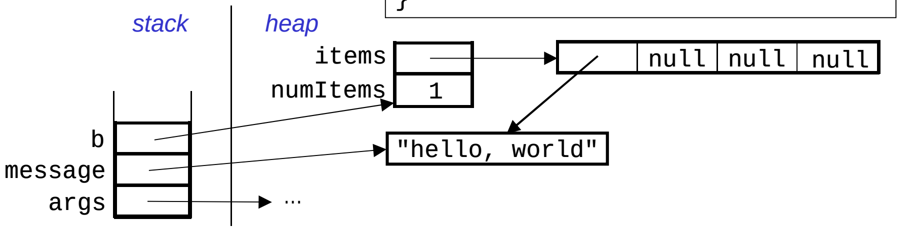


- The method modifies the `items` array and `numItems`.
  - note that the array holds a copy of the *reference* to the item, not a copy of the item itself.

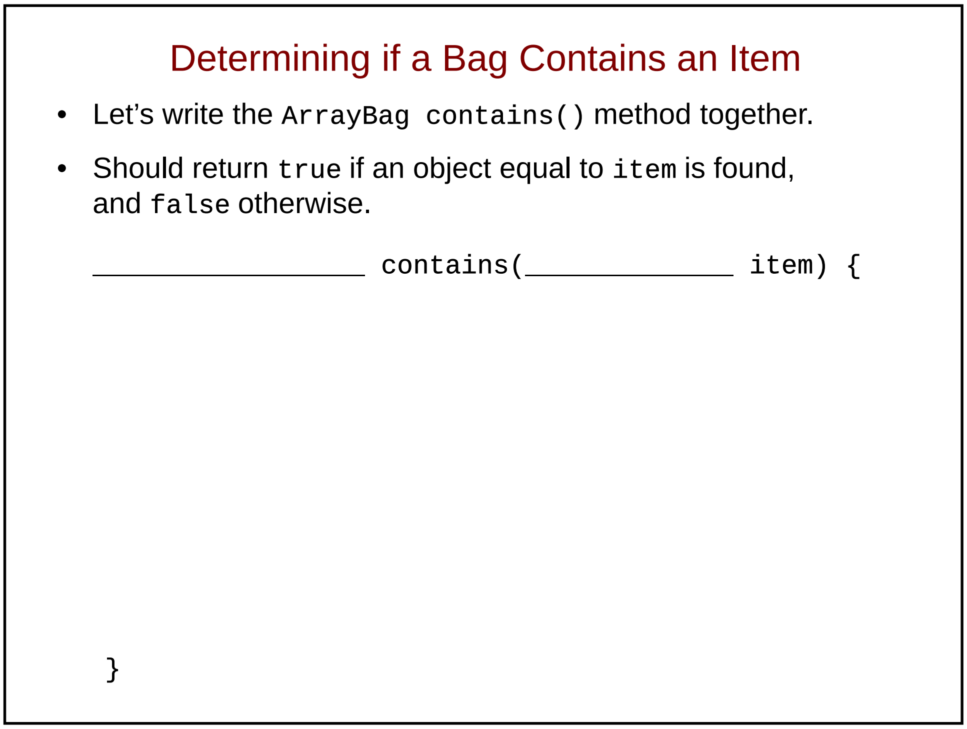
Example: Adding an Item (cont.)

```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```



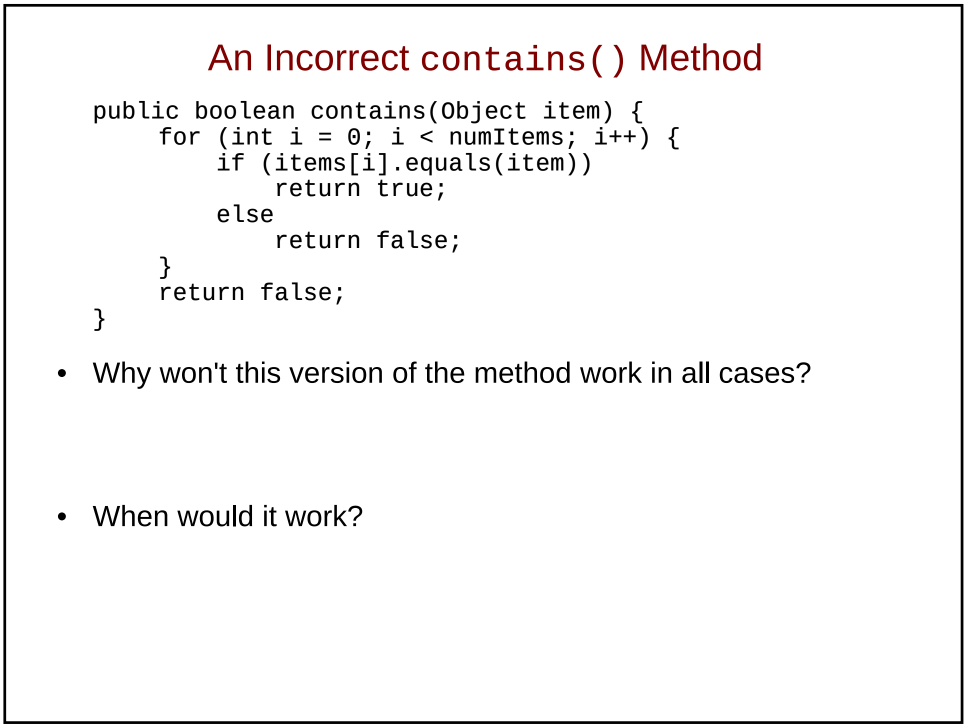
- After the method call returns, `add`'s stack frame is removed from the stack.



Determining if a Bag Contains an Item

- Let’s write the `ArrayBag contains()` method together.
- Should return `true` if an object equal to `item` is found, and `false` otherwise.

```
___________________ contains(______________ item) {










 }
```



An Incorrect `contains()` Method

```
public boolean contains(Object item) {
    for (int i = 0; i < numItems; i++) {
        if (items[i].equals(item))
            return true;
        else
            return false;
    }
    return false;
}
```

- Why won't this version of the method work in all cases?
- When would it work?

A Method That Takes a Bag as a Parameter

```
public boolean containsAll(Bag otherBag) {
    if (otherBag == null || otherBag.numItems() == 0)
        return false;

    Object[] otherItems = otherBag.toArray();
    for (int i = 0; i < otherItems.length; i++) {
        if (!contains(otherItems[i]))
            return false;
    }

    return true;
}
```

- We use `Bag` instead of `ArrayBag` as the type of the parameter.
  - allows this method to be part of the `Bag` interface
  - allows us to pass in *any* object that implements `Bag`
- Because the parameter may not be an `ArrayBag`, we can't assume it has `items` and `numItems` fields.
  - instead, we use `toArray()` and `numItems()`

A Need for Casting

- Let's say that we want to store a collection of `String` objects in an `ArrayBag`.
- `String` is a subclass of `Object`, so we can store `String` objects in the bag without doing anything special:

```
ArrayBag stringBag = new ArrayBag();
stringBag.add("hello");
stringBag.add("world");
```

- `Object` isn't a subclass of `String`, so this will not work:

```
String str = stringBag.grab();    // compiler error
```

- Instead, we need to use casting:

```
String str = (String)stringBag.grab();
```

Extra: Thinking About a Method's Efficiency

- For a bag with 1000 items, how many items will `contains()` look at:
  - in the best case?
  - in the worst case?
  - in the average case?

- Could we make it more efficient?

- If so, what changes would be needed to do so, and what would be the impact of those changes?

Extra: Understanding Memory Management

- Our Bag ADT has a method `toArray()`, which returns an array containing the current contents of the bag
  - allows users of the ADT to iterate over the items

- When implementing `toArray()` in our `ArrayBag` class, can we just return a reference to the `items` array? Why or why not?

Unit 6, Part 2

Recursion Revisited; Recursive Backtracking

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Recursive Problem-Solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.
- This simplest problem is known as the *base case*.

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {           // base case
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The base case stops the recursion, because it doesn't make another call to the method.

Review: Recursive Problem-Solving (cont.)

- If the base case hasn't been reached, we execute the *recursive case.*

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {            // base case
        System.out.println(n2);
    } else {                   // recursive case
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The recursive case:
  - reduces the overall problem to one or more simpler problems of the same kind
  - makes recursive calls to solve the simpler problems

Raising a Number to a Power

- We want to write a recursive method to compute

$$x^n = \underbrace{x*x*x*\ldots*x}_{n \text{ of them}}$$

  where *x* and *n* are both integers and *n* >= 0.

- Examples:
  - 2^{10} = 2*2*2*2*2*2*2*2*2*2 = 1024
  - 10^{5} = 10*10*10*10*10 = 100000

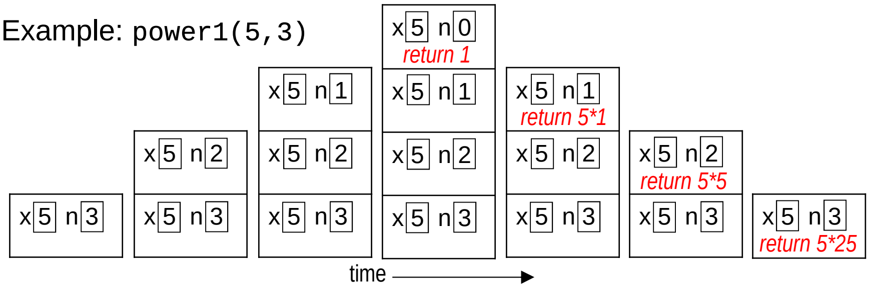
- Computing a power recursively: $2^{10} = 2*2^9$
  $= 2*(2 * 2^8)$
  $= \ldots$

- Recursive definition: $x^n = x * x^{n-1}$ when $n > 0$
  $x^0 = 1$

Power Method: First Try

```
public class Power {
    public static int power1(int x, int n) {
        if (n < 0)
            throw new IllegalArgumentException(
              “n must be >= 0”);
        if (n == 0)
            return 1;
        else
            return x * power1(x, n-1);
    }
}
```

Example: `power1(5,3)`



x 5 n 3

x 5 n 2
x 5 n 3

x 5 n 1
x 5 n 2
x 5 n 3

x 5 n 0 *return 1*
x 5 n 1
x 5 n 2
x 5 n 3

x 5 n 1 *return 5*1*
x 5 n 2
x 5 n 3

x 5 n 2 *return 5*5*
x 5 n 3

x 5 n 3 *return 5*25*

time →

Power Method: Second Try

- There’s a better way to break these problems into subproblems. For example: $2^{10} = (2*2*2*2*2)*(2*2*2*2*2)$
  $= (2^5) * (2^5) = (2^5)^2$
- A more efficient recursive definition of x^n (when $n > 0$):
  $x^n = (x^{n/2})^2$ when n is even
  $x^n = x * (x^{n/2})^2$ when n is odd (using integer division for n/2)
- Let's write the corresponding method together:

```
public static int power2(int x, int n) {





}
```

Analyzing power2

- How many method calls would it take to compute 2^{1000} ?

```
power2(2, 1000)
  power2(2, 500)
    power2(2, 250)
      power2(2, 125)
        power2(2, 62)
          power2(2, 31)
            power2(2, 15)
              power2(2, 7)
                power2(2, 3)
                  power2(2, 1)
                    power2(2, 0)
```

- Much more efficient than power1() for large n.
- It can be shown that it takes approx. $\log_2 n$ method calls.

An Inefficient Version of power2

- What's wrong with the following version of `power2()`?

```
public static int power2Bad(int x, int n) {
    // code to handle n < 0 goes here...
    if (n == 0)
        return 1;
    if ((n % 2) == 0)
        return power2(x, n/2) * power2(x, n/2);
    else
        return x * power2(x, n/2) * power2(x, n/2);
}
```

Review: Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string
- Thus, we can easily use recursion to process a string.
  - process one or two of the characters
  - make a recursive call to process the rest of the string
- Example: print a string vertically, one character per line:

```
public static void printVertical(String str) {
    if (str == null || str.equals("")) {
        return;
    }

    System.out.println(str.charAt(0)); // first char
    printVertical(str.substring(1));   // rest of string
}
```

Removing Vowels from a String

- Let's design a recursive method called `removeVowels()`.
- `removeVowels(str)` should return a string in which all of the vowels in the string `str` have been removed.
  - example:

    `removeVowels("recurse")`

    should return

    `"rcrs"`
- Thinking recursively:

Removing Vowels from a String (cont.)

- Put the method definition here:

Recursive Backtracking: the n-Queens Problem

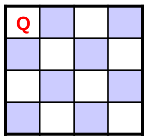
- Find all possible ways of placing n queens on an n x n chessboard so that no two queens occupy the same row, column, or diagonal.
- Sample solution for n = 8:

Q							
				Q			
							Q
					Q		
		Q					
						Q	
	Q						
			Q				

- This is a classic example of a problem that can be solved using a technique called *recursive backtracking*.

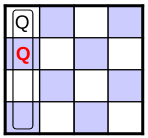
Recursive Strategy for n-Queens

- Consider one row at a time. Within the row, consider one column at a time, looking for a “safe” column to place a queen.
- If we find one, place the queen, and *make a recursive call* to place a queen on the next row.
- If we can’t find one, *backtrack* by returning from the recursive call, and try to find another safe column in the previous row.
- Example for n = 4:
  - row 0:

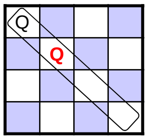


col 0: safe

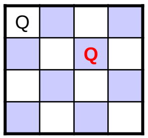
  - row 1:



col 0: same col



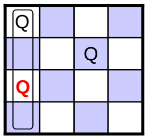
col 1: same diag



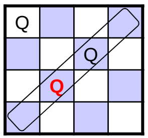
col 2: safe

4-Queens Example (cont.)

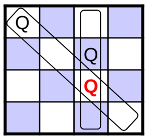
- row 2:



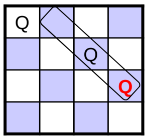
col 0: same col



col 1: same diag

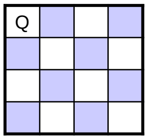


col 2: same col/diag

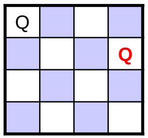


col 3: same diag

- We’ve run out of columns in row 2!
- *Backtrack* to row 1 by returning from the recursive call.
  - pick up where we left off
  - we had already tried columns 0-2, so now we try column 3:



we left off in col 2

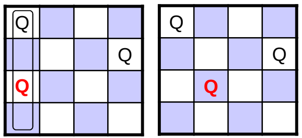


try col 3: safe

- Continue the recursion as before.

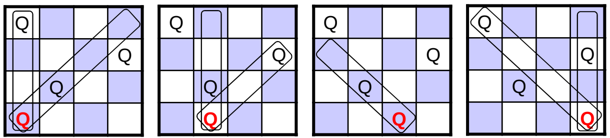
4-Queens Example (cont.)

- row 2:



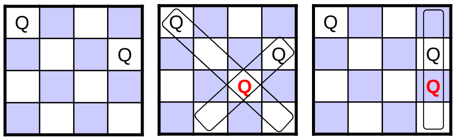
col 0: same col   col 1: safe

- row 3:



col 0: same col/diag   col 1: same col/diag   col 2: same diag   col 3: same col/diag

- Backtrack to row 2:

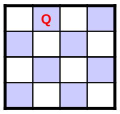


we left off in col 1   col 2: same diag   col 3: same col

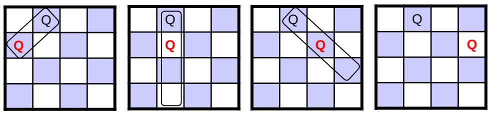
- Backtrack to row 1. No columns left, so backtrack to row 0!

4-Queens Example (cont.)

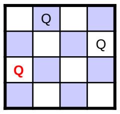
- row 0:



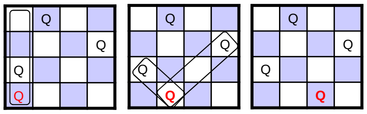
- row 1:



- row 2:



- row 3:



A solution!

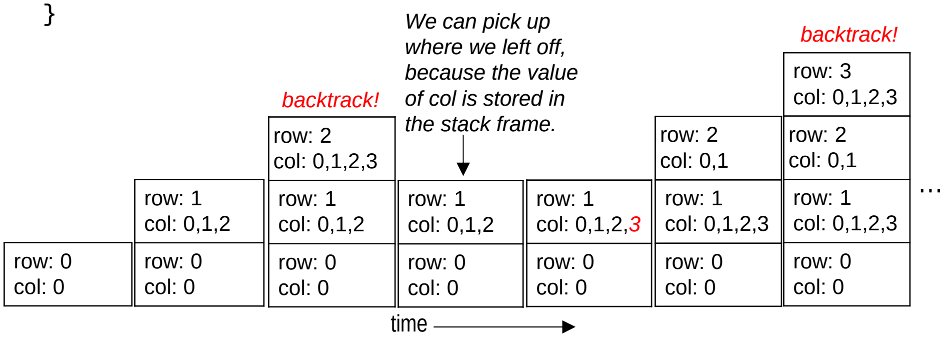
findSafeColumn() Method

```
public void findSafeColumn(int row) {
    if (row == boardSize) {  // base case: a solution!
        solutionsFound++;
        displayBoard();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (int col = 0; col < boardSize; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);

            // Move onto the next row.
            findSafeColumn(row + 1);

            // If we get here, we've backtracked.
            removeQueen(row, col);
        }
    }
}
```

Note: neither row++ nor ++row will work here.

Tracing findSafeColumn()

```
public void findSafeColumn(int row) {
    if (row == boardSize) {
        // code to process a solution goes here...
    }
    for (int col = 0; col < BOARD_SIZE; col++) {
        if (isSafe(row, col)) {
            placeQueen(row, col);
            findSafeColumn(row + 1);
            removeQueen(row, col);
        }
    }
}
```



row: 0
col: 0

row: 1
col: 0,1,2
row: 0
col: 0

backtrack!
row: 2
col: 0,1,2,3
row: 1
col: 0,1,2
row: 0
col: 0

We can pick up where we left off, because the value of col is stored in the stack frame.
row: 1
col: 0,1,2
row: 0
col: 0

row: 1
col: 0,1,2,3
row: 0
col: 0

row: 2
col: 0,1
row: 1
col: 0,1,2,3
row: 0
col: 0

backtrack!
row: 3
col: 0,1,2,3
row: 2
col: 0,1
row: 1
col: 0,1,2,3
row: 0
col: 0

…

time

Template for Recursive Backtracking

```
void findSolutions(n, other params) {
    if (found a solution) {
        solutionsFound++;
        displaySolution();
        if (solutionsFound >= solutionTarget)
            System.exit(0);
        return;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            findSolutions(n + 1, other params);
            removeValue(val, n);
        }
    }
}
```

Template for Finding a Single Solution

```
boolean findSolutions(n, other params) {
    if (found a solution) {
        displaySolution();
        return true;
    }

    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            if (findSolutions(n + 1, other params))
                return true;
            removeValue(val, n);
        }
    }

    return false;
}
```

Data Structures for n-Queens

- Three key operations:
  - `isSafe(row, col)`: check to see if a position is safe
  - `placeQueen(row, col)`
  - `removeQueen(row, col)`
- A two-dim. array of booleans would be sufficient:

```
public class Queens {
    private boolean[][] queenOnSquare;
```

- Advantage: easy to place or remove a queen:

```
        public void placeQueen(int row, int col) {
            queenOnSquare[row][col] = true;
        }
        public void removeQueen(int row, int col) {
            queenOnSquare[row][col] = false;
        }
    …
```

- Problem: `isSafe()` takes a lot of steps. What matters more?

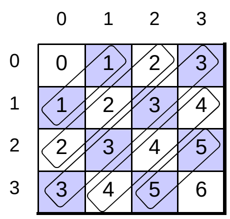
Additional Data Structures for n-Queens

- To facilitate `isSafe()`, add three arrays of booleans:

```
        private boolean[] colEmpty;
        private boolean[] upDiagEmpty;
        private boolean[] downDiagEmpty;
```

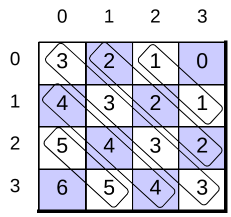
- An entry in one of these arrays is:
  – `true` if there are no queens in the column or diagonal
  – `false` otherwise
- Numbering diagonals to get the indices into the arrays:

upDiag = row + col



	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

downDiag =
(boardSize – 1) + row – col



	0	1	2	3
0	3	2	1	0
1	4	3	2	1
2	5	4	3	2
3	6	5	4	3

Using the Additional Arrays

- Placing and removing a queen now involve updating four arrays instead of just one. For example:

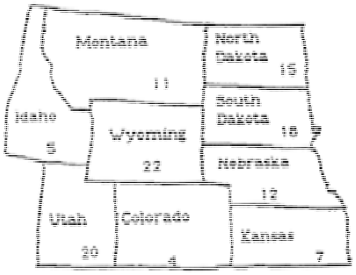
```
public void placeQueen(int row, int col) {
    queenOnSquare[row][col] = true;
    colEmpty[col] = false;
    upDiagEmpty[row + col] = false;
    downDiagEmpty[(boardSize - 1) + row - col] = false;
}
```

- However, checking if a square is safe is now more efficient:

```
public boolean isSafe(int row, int col) {
    return (colEmpty[col]
      && upDiagEmpty[row + col]
      && downDiagEmpty[(boardSize - 1) + row - col]);
}
```

Recursive Backtracking II: Map Coloring

- Using just four colors (e.g., red, orange, green, and blue), we want color a map so that no two bordering states or countries have the same color.
- Sample map (numbers show alphabetical order in full list of state names):



- This is another example of a problem that can be solved using recursive backtracking.

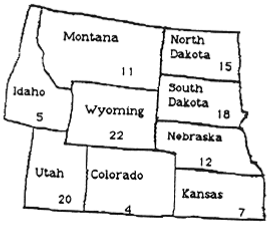
Applying the Template to Map Coloring

```
boolean findSolutions(n, other params) {
    if (found a solution) {
        displaySolution();
        return true;
    }
    for (val = first to last) {
        if (isValid(val, n)) {
            applyValue(val, n);
            if (findSolutions(n + 1, other params))
                return true;
            removeValue(val, n);
        }
    }
    return false;
}
```

template element	***meaning in map coloring***
n	
found a solution	
val	
isValid(val, n)	
applyValue(val, n)	
removeValue(val, n)	

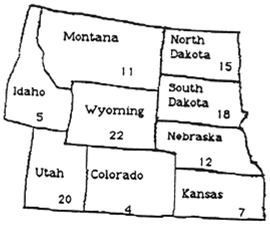
Map Coloring Example

consider the states in alphabetical order. colors = { red, yellow, green, blue }.



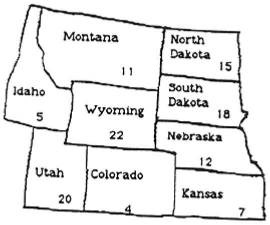
We color Colorado through Utah without a problem.

Colorado:
Idaho:
Kansas:
Montana:
Nebraska:
North Dakota:
South Dakota:
Utah:



No color works for Wyoming, so we backtrack…

Map Coloring Example (cont.)



Now we can complete
the coloring:

Recursive Backtracking in General

- Useful for *constraint satisfaction problems* that involve assigning values to variables according to a set of constraints.
  - n-Queens:
    - variables = Queen's position in each row
    - constraints = no two queens in same row, column, diagonal
  - map coloring
    - variables = each state's color
    - constraints = no two bordering states with the same color
  - many others: factory scheduling, room scheduling, etc.
- Backtracking reduces the # of possible value assignments that we consider, because it never considers invalid assignments.…
- Using recursion allows us to easily handle an arbitrary number of variables.
  - stores the state of each variable in a separate stack frame

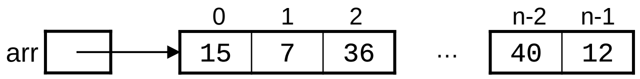
Unit 7

Sorting and Algorithm Analysis

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort "in place," using only a small amount of additional storage
- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of **comparisons *C*** and the number of **moves *M*** needed to sort the array.
  - move = copying an element from one position to another
    example: `arr[3] = arr[5];`

Defining a Class for our Sort Methods

```
public class Sort {
    public static void bubbleSort(int[] arr) {
        ...
    }
    public static void insertionSort(int[] arr) {
        ...
    }
    ...
}
```

- Our `Sort` class is simply a collection of methods like Java's built-in `Math` class.
- Because we never create `Sort` objects, all of the methods in the class must be *static*.
    - outside the class, we invoke them using the class name: e.g., `Sort.bubbleSort(arr)`

Defining a Swap Method

- It would be helpful to have a method that swaps two elements of the array.
- Why won't the following work?

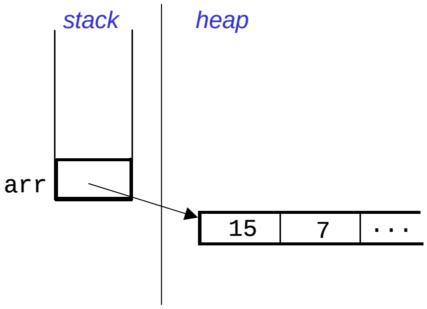
```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

An Incorrect Swap Method

```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Trace through the following lines to see the problem:

```
int[] arr = {15, 7, …};
swap(arr[0], arr[1]);
```



A Correct Swap Method

- This method works:

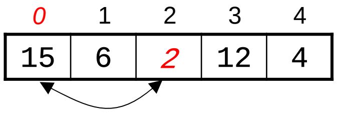
```
public static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

- Trace through the following with a memory diagram to convince yourself that it works:

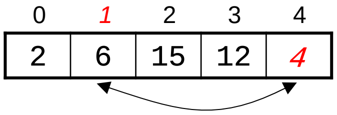
```
int[] arr = {15, 7, …};
swap(arr, 0, 1);
```

Selection Sort

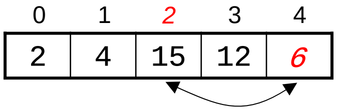
- Basic idea:
  - consider the positions in the array from left to right
  - for each position, find the element that belongs there and put it in place by swapping it with the element that's currently there
- Example:



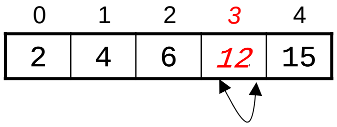
0	1	2	3	4
15	6	*2*	12	4



0	*1*	2	3	4
2	6	15	12	*4*



0	1	*2*	3	4
2	4	15	12	*6*



0	1	2	*3*	4
2	4	6	*12*	15

Why don't we need to consider position `4`?

Selecting an Element

- When we consider position `i`, the elements in positions `0` through `i – 1` are already in their final positions.

example for `i` = 3:

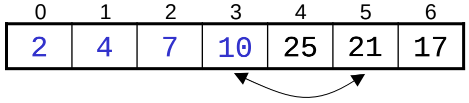
0	1	2	*3*	4	5	6
2	4	7	21	25	10	17

- To select an element for position `i`:
  - consider elements `i`, `i+1,i+2,…,arr.length – 1`, and keep track of `indexMin`, the index of the smallest element seen thus far

`indexMin: 3, 5`

0	1	2	3	4	5	6
2	4	7	21	25	*10*	17

  - when we finish this pass, `indexMin` is the index of the element that belongs in position `i`.
  - swap `arr[i]` and `arr[indexMin]`:



0	1	2	3	4	5	6
2	4	7	10	25	21	17

Implementation of Selection Sort

- Use a helper method to find the index of the smallest element:

```
private static int indexSmallest(int[] arr,
  int lower, int upper) {
    int indexMin = lower;

    for (int i = lower+1; i <= upper; i++)
        if (arr[i] < arr[indexMin])
            indexMin = i;

    return indexMin;
}
```

- The actual sort method is very simple:

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
        int j = indexSmallest(arr, i, arr.length-1);
        swap(arr, i, j);
    }
}
```

Time Analysis

- Some algorithms are much more efficient than others.
- The *time efficiency* or *time complexity* of an algorithm is some measure of the number of “operations” that it performs.
  - for sorting algorithms, we’ll focus on two types of operations: comparisons and moves
- The number of operations that an algorithm performs typically depends on the size, n, of its input.
  - for sorting algorithms, n is the # of elements in the array
  - `C(n)` = number of comparisons
  - `M(n)` = number of moves
- To express the time complexity of an algorithm, we’ll express the number of operations performed as a function of `n`.
  - examples: `C(n)` = $n^2 + 3n$
    `M(n)` = $2n^2 - 1$

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int lower, int upper){
    int indexMin = lower;

    for (int i = lower+1; i <= upper; i++)
        if (arr[i] < arr[indexMin])
            indexMin = i;

    return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
        int j = indexSmallest(arr, i, arr.length-1);
        swap(arr, i, j);
    }
}
```

- To sort n elements, selection sort performs `n - 1` passes:

  on `1`st pass, it performs `n - 1` comparisons to find `indexSmallest`
  on `2`nd pass, it performs `n - 2` comparisons
  …
  on the (`n-1`)st pass, it performs `1` comparison

- Adding up the comparisons for each pass, we get:

  `C(n) = 1 + 2 + … + (n - 2) + (n - 1)`

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for `C(n)` is the sum of an arithmetic sequence:

$$C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1) = \sum_{i=1}^{n-1} i$$

- Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

- Thus, we can simplify our expression for C(n) as follows:

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &= \frac{(n-1)n}{2} \end{aligned}$$

$$\boxed{C(n) = n^2/2 - n/2}$$

Focusing on the Largest Term

- When n is large, mathematical expressions of n are dominated by their “largest” term — i.e., the term that grows fastest as a function of n.
  - example:

n	$n^2/2$	n/2	$n^2/2$ – n/2
10	50	5	45
100	5000	50	4950
10000	50,000,000	5000	49,995,000

- In characterizing the time complexity of an algorithm, we’ll focus on the largest term in its operation-count expression.
  - for selection sort, $C(n) = n^2/2 - n/2 \approx n^2/2$
- In addition, we’ll typically ignore the coefficient of the largest term (e.g., $n^2/2 \rightarrow n^2$).

Big-*O* Notation

- We specify the largest term using big-*O* notation.
  - e.g., we say that $C(n) = n^2/2 – n/2$ is $O(n^2)$
- Common classes of algorithms:

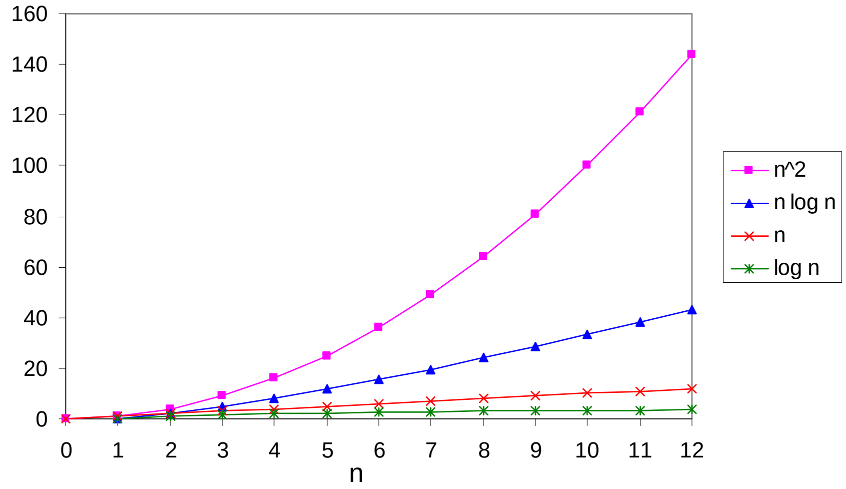
name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n – 2\log_2 n$	$O(n)$
$n\log n$ time	$4n\log_2 n$, $n\log_2 n + n$	$O(n\log n)$
quadratic time	$2n^2 + 3n$, $n^2 – 1$	$O(n^2)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$

slower

- For large inputs, efficiency matters more than CPU speed.
  - e.g., an $O(\log n)$ algorithm on a slow machine will outperform an $O(n)$ algorithm on a fast machine

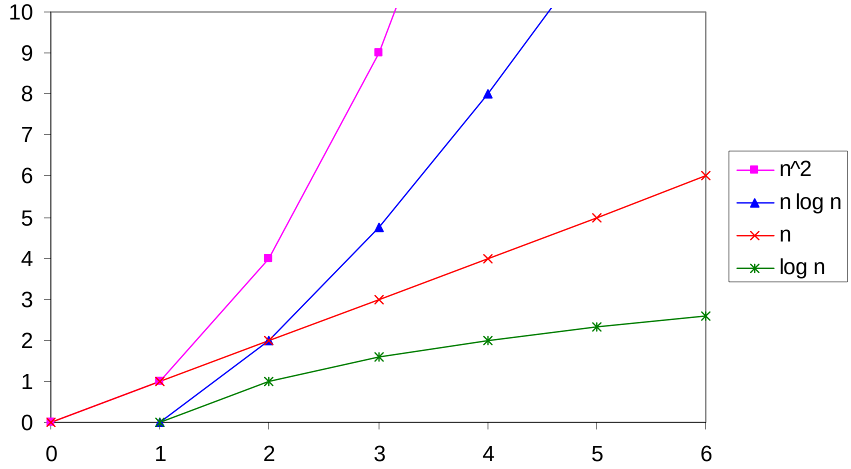
Ordering of Functions

- We can see below that: n^2 grows faster than $n\log_2 n$
  $n\log_2 n$ grows faster than n
  n grows faster than $\log_2 n$



Ordering of Functions (cont.)

- Zooming in, we see that: n^2 >= n for all n >= 1
  $n\log_2 n$ >= n for all n >= 2
  n > $\log_2 n$ for all n >= 1

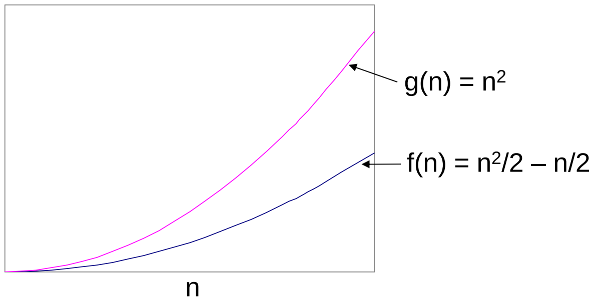


Mathematical Definition of Big-*O* Notation

- f(n) = *O*(g(n)) if there exist positive constants c and n_0 such that f(n) <= c g(n) for all n >= n_0
- Example: f(n) = $n^2/2 - n/2$ is *O*(n^2), because

  $n^2/2 - n/2 \;\overset{}{<=}\; n^2$ for all n >= 0.

  c = 1 $\quad$ $n_0 = 0$



- Big-*O* notation specifies an *upper bound* on a function f(n) as n grows large.

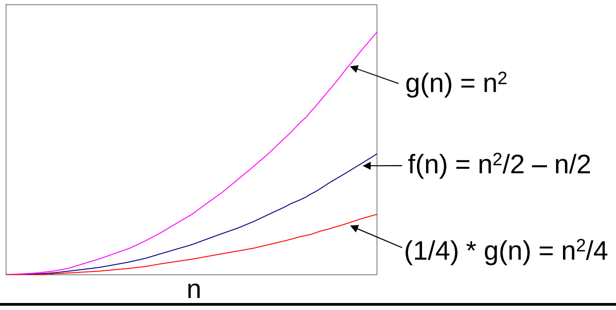
Big-*O* Notation and Tight Bounds

- Big-O notation provides an upper bound, *not* a tight bound (upper and lower).
- Example:
  - 3n – 3 is *O*(n^2) because 3n – 3 <= n^2 for all n >= 1
  - 3n – 3 is also *O*(2^n) because 3n – 3 <= 2^n for all n >= 1
- However, we generally try to use big-O notation to characterize a function as closely as possible – i.e., as if we were using it to specify a tight bound.
  - for our example, we would say that 3n – 3 is *O*(n)

Big-Theta Notation

- In theoretical computer science, *big-theta* notation (Θ) is used to specify a tight bound.
- $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2, and n_0 such that $c_1 g(n) <= f(n) <= c_2 g(n)$ for all $n > n_0$
- Example: $f(n) = n^2/2 – n/2$ is $\Theta(n^2)$, because
  $(1/4)*n^2 <= n^2/2 – n/2 <= n^2$ for all $n >= 2$

$c_1 = 1/4$ $c_2 = 1$ $n_0 = 2$



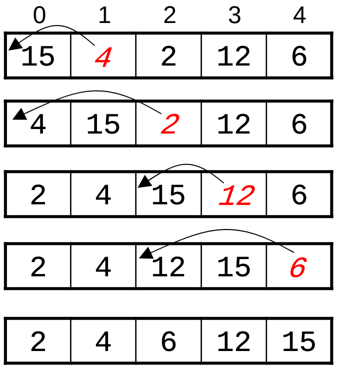
Big-*O* Time Analysis of Selection Sort

- Comparisons: we showed that `C(n)` = `n²/2 – n/2`
  - selection sort performs O(`n²`) comparisons
- Moves: after each of the `n-1` passes to find the smallest remaining element, the algorithm performs a swap to put the element in place.
  - `n–1` swaps, 3 moves per swap
  - `M(n) = 3(n-1) = 3n-3`
  - selection sort performs O(`n`) moves.
- Running time (i.e., total operations): ?

Sorting by Insertion I: Insertion Sort

- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left, “sliding over” other elements to make room.

- Example:



0	1	2	3	4
15	*4*	2	12	6
4	15	*2*	12	6
2	4	15	*12*	6
2	4	12	15	*6*
2	4	6	12	15

Comparing Selection and Insertion Strategies

- In selection sort, we start with the *positions* in the array and *select* the correct elements to fill them.

- In insertion sort, we start with the *elements* and determine where to *insert* them in the array.

- Here’s an example that illustrates the difference:

0	1	2	3	4	5	6
18	12	15	9	25	2	17

- Sorting by selection:
  - consider position 0: find the element (2) that belongs there
  - consider position 1: find the element (9) that belongs there
  - …

- Sorting by insertion:
  - consider the 12: determine where to insert it
  - consider the 15; determine where to insert it
  - …

Inserting an Element

- When we consider element `i`, elements `0` through `i – 1` are already sorted with respect to each other.

example for `i = 3`:

0	1	2	3	4
6	14	19	9	…

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

`toInsert` 9

0	1	2	3
6	14	19	9

  - consider elements `i-1`, `i-2`, …
    - if an element `>` `toInsert`, slide it over to the right
    - stop at the first element `<=` `toInsert`

`toInsert` 9

0	1	2	3
6		14	19

  - copy `toInsert` into the resulting “hole”:

0	1	2	3
6	9	14	19

Insertion Sort Example (done together)

description of steps

12	5	2	13	18	4

Implementation of Insertion Sort

```
public class Sort {
  ...
  public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int toInsert = arr[i];

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Time Analysis of Insertion Sort

- The number of operations depends on the contents of the array.
- *best case:*
- *worst case:*
- *average case:*

Sorting by Insertion II: Shell Sort

- Developed by Donald Shell in 1959
- Improves on insertion sort
- Takes advantage of the fact that insertion sort is fast when an array is almost sorted.
- Seeks to eliminate a disadvantage of insertion sort:
  if an element is far from its final location, many “small” moves are required to put it where it belongs.
- Example: if the largest element starts out at the beginning of the array, it moves one place to the right on *every* insertion!

0	1	2	3	4	5	…	1000
999	42	56	30	18	23	…	11

- Shell sort uses “larger” moves that allow elements to quickly get close to where they belong.

Sorting Subarrays

- Basic idea:
  - use insertion sort on subarrays that contain elements separated by some increment
    - increments allow the data items to make larger “jumps”
  - repeat using a decreasing sequence of increments
- Example for an initial increment of 3:

0	1	2	3	4	5	6	7
36	18	10	27	3	20	9	8

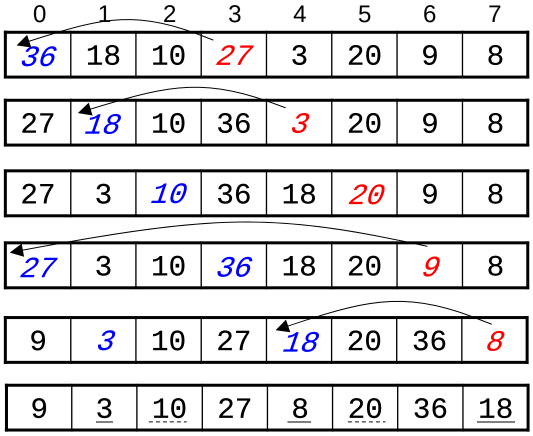
  - three subarrays:
    1) elements 0, 3, 6   2) elements 1, 4, 7   3) elements 2 and 5
- Sort the subarrays using insertion sort to get the following:

0	1	2	3	4	5	6	7
9	3	10	27	8	20	36	18

- Next, we complete the process using an increment of 1.

Shell Sort: A Single Pass

- We *don't* consider the subarrays one at a time.
- We consider elements `arr[incr]` through `arr[arr.length-1]`, inserting each element into its proper place with respect to the elements *from its subarray* that are to the left of the element.
- The same example (`incr` = 3):



0	1	2	3	4	5	6	7
36	18	10	*27*	3	20	9	8
27	*18*	10	36	*3*	20	9	8
27	3	*10*	36	18	*20*	9	8
27	3	10	*36*	18	20	*9*	8
9	*3*	10	27	*18*	20	36	*8*
9	3	10	27	8	20	36	18

Inserting an Element in a Subarray

- When we consider element `i`, the other elements in its subarray are already sorted with respect to each other.

example for `i = 6`:
`(incr = 3)`

0	1	2	3	4	5	6	7
27	3	10	*36*	18	20	*9*	8

the other element's in 9's subarray (the 27 and 36)
are already sorted with respect to each other

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

`toInsert` *9*

0	1	2	3	4	5	6	7
27	3	10	*36*	18	20	*9*	8

  - consider elements `i-incr`, `i-(2*incr)`, `i-(3*incr)`,...
    - if an element `>` `toInsert`, slide it right *within the subarray*
    - stop at the first element `<=` `toInsert`

`toInsert` *9*

0	1	2	3	4	5	6	7
	3	10	*27*	18	20	*36*	8

  - copy `toInsert` into the "hole":

0	1	2	3	4	
9	3	10	*27*	18	…

The Sequence of Increments

- Different sequences of decreasing increments can be used.
- Our version uses values that are one less than a power of two.
  - $2^k - 1$ for some k
  - … 63, 31, 15, 7, 3, 1
  - can get to the next lower increment using integer division:

    ```
    incr = incr/2;
    ```
- Should avoid numbers that are multiples of each other.
  - otherwise, elements that are sorted with respect to each other in one pass are grouped together again in subsequent passes
    - repeat comparisons unnecessarily
    - get fewer of the large jumps that speed up later passes
  - example of a bad sequence: 64, 32, 16, 8, 4, 2, 1
    - what happens if the largest values are all in odd positions?

Implementation of Shell Sort

```
public static void shellSort(int[] arr) {
    int incr = 1;
    while (2 * incr <= arr.length)
        incr = 2 * incr;
    incr = incr - 1;

    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            if (arr[i] < arr[i-incr]) {
                int toInsert = arr[i];

                int j = i;
                do {
                     arr[j] = arr[j-incr];
                     j = j - incr;
                } while (j > incr-1 &&
                   toInsert < arr[j-incr]);

                arr[j] = toInsert;
            }
        }
        incr = incr/2;
    }
}
```

(If you replace `incr` *with* `1` *in the for-loop, you get the code for insertion sort.)*

Time Analysis of Shell Sort

- Difficult to analyze precisely
  - typically use experiments to measure its efficiency
- With a bad interval sequence, it's $O(n^2)$ in the worst case.
- With a good interval sequence, it's better than $O(n^2)$.
  - at least $O(n^{1.5})$ in the average and worst case
  - some experiments have shown average-case running times of $O(n^{1.25})$ or even $O(n^{7/6})$
- Significantly better than insertion or selection for large n:

n	n^2	$n^{1.5}$	$n^{1.25}$
10	100	31.6	17.8
100	10,000	1000	316
10,000	100,000,000	1,000,000	100,000
10^6	10^{12}	10^9	3.16×10^7

- We've wrapped insertion sort in another loop and increased its efficiency! The key is in the larger jumps that Shell sort allows.

Sorting by Exchange I: Bubble Sort

- Perform a sequence of passes through the array.
- On each pass: proceed from left to right, swapping adjacent elements if they are out of order.
- Larger elements "bubble up" to the end of the array.
- At the end of the kth pass, the k rightmost elements are in their final positions, so we don't need to consider them in subsequent passes.
- Example:

	0	1	2	3
	28	24	27	18
after the first pass:	24	27	18	*28*
after the second:	24	18	*27*	*28*
after the third:	18	*24*	*27*	*28*

Implementation of Bubble Sort

```
public class Sort {
    ...
    public static void bubbleSort(int[] arr) {
        for (int i = arr.length – 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (arr[j] > arr[j+1])
                    swap(arr, j, j+1);
            }
        }
    }
}
```

- One for-loop nested in another:
  - the inner loop performs a single pass
  - the outer loop governs the number of passes, and the ending point of each pass

Time Analysis of Bubble Sort

- Comparisons: the kth pass performs ______ comparisons,

  so we get `C(n) =`

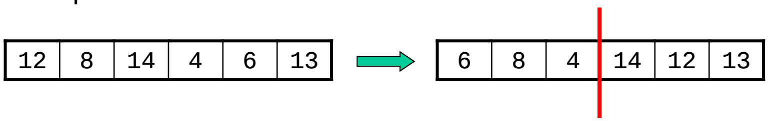
- Moves: depends on the contents of the array
  - in the worst case:
  - in the best case:
- Running time:

Sorting by Exchange II: Quicksort

- Like bubble sort, quicksort uses an approach based on exchanging out-of-order elements, but it's more efficient.
- A recursive, divide-and-conquer algorithm:
  - *divide:* rearrange the elements so that we end up with two subarrays that meet the following criterion:

    each element in the left array <= each element in the right array

    example:



12	8	14	4	6	13

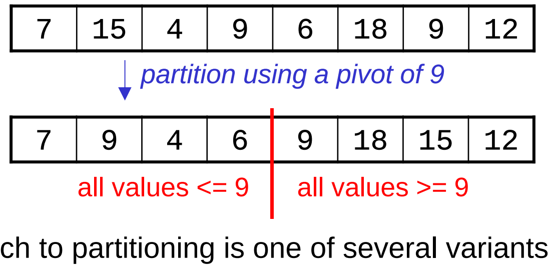
6	8	4	14	12	13

  - *conquer:* apply quicksort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* nothing needs to be done, because of the criterion used in forming the subarrays

Partitioning an Array Using a Pivot

- The process that quicksort uses to rearrange the elements is known as *partitioning* the array.
- Partitioning is done using a value known as the *pivot.*
- We rearrange the elements to produce two subarrays:
  - left subarray: all values <= pivot
  - right subarray: all values >= pivot

  equivalent to the criterion on the previous page.



7	15	4	9	6	18	9	12

partition using a pivot of 9

7	9	4	6	9	18	15	12

all values <= 9

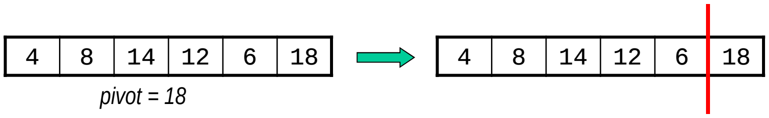
all values >= 9

- Our approach to partitioning is one of several variants.
- Partitioning is useful in its own right.

  ex: find all students with a GPA > 3.0.

Possible Pivot Values

- First element or last element
  - risky, can lead to terrible worst-case behavior
  - especially poor if the array is almost sorted



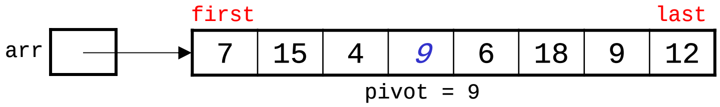
4	8	14	12	6	18

pivot = 18

4	8	14	12	6	18

- Middle element (what we will use)
- Randomly chosen element
- Median of three elements
  - left, center, and right elements
  - three randomly selected elements
  - taking the median of three decreases the probability of getting a poor pivot

Partitioning an Array: An Example

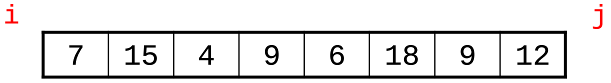


7	15	4	*9*	6	18	9	12

`pivot = 9`

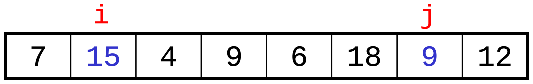
- Maintain indices `i` and `j`, starting them "outside" the array:

```
i = first – 1
j = last + 1
```



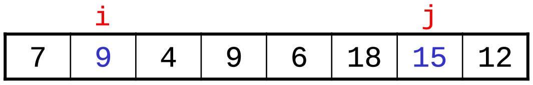
7	15	4	9	6	18	9	12

- *Find* "out of place" elements:
  - increment `i` until `arr[i] >= pivot`
  - decrement `j` until `arr[j] <= pivot`

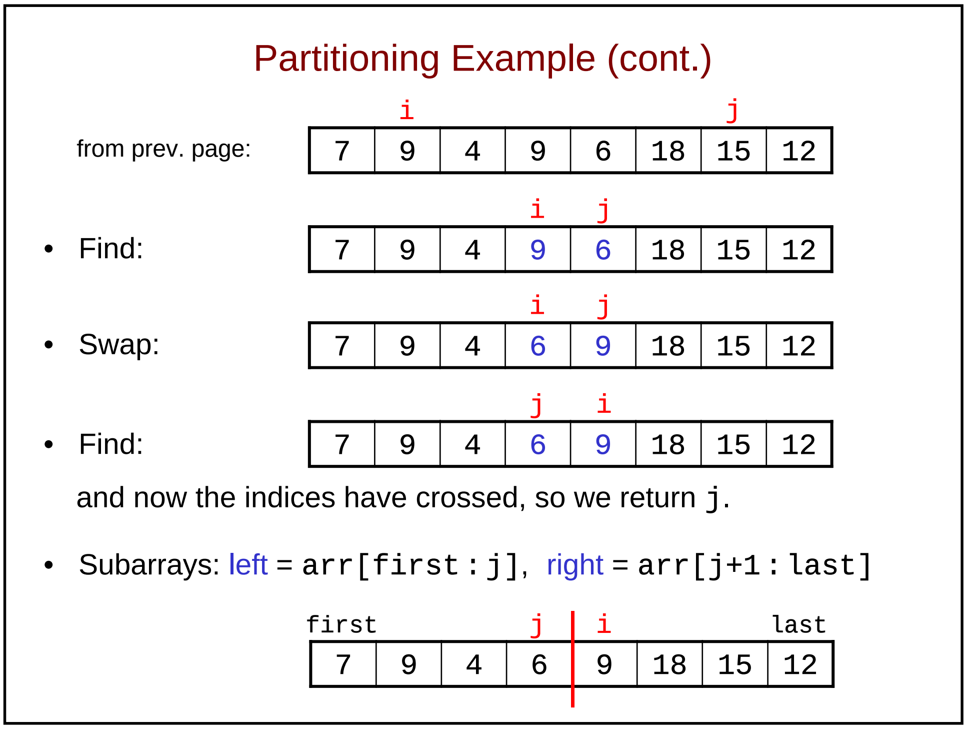


7	15	4	9	6	18	9	12

- *Swap* `arr[i]` and `arr[j]`:



7	9	4	9	6	18	15	12



Partitioning Example (cont.)

from prev. page:

	i					j	
7	9	4	9	6	18	15	12

- Find:

			i	j			
7	9	4	9	6	18	15	12

- Swap:

			i	j			
7	9	4	6	9	18	15	12

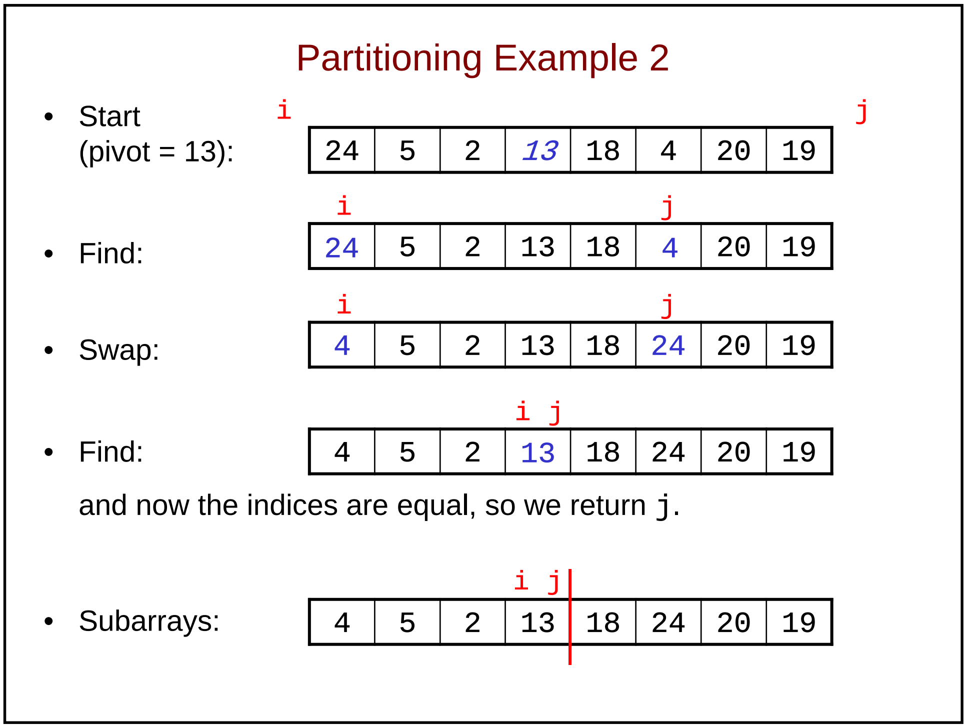
- Find:

			j	i			
7	9	4	6	9	18	15	12

and now the indices have crossed, so we return `j`.

- Subarrays: left = `arr[first : j]`, right = `arr[j+1 : last]`

first			j	i			last
7	9	4	6	9	18	15	12



Partitioning Example 2

- Start (pivot = 13):

i									j
	24	5	2	*13*	18	4	20	19	

- Find:

i					j		
24	5	2	13	18	4	20	19

- Swap:

i					j		
4	5	2	13	18	24	20	19

- Find:

			i j				
4	5	2	13	18	24	20	19

and now the indices are equal, so we return `j`.

- Subarrays:

			i j				
4	5	2	13	18	24	20	19

Partitioning Example 3 (done together)

- Start (pivot = 5):

i									j
	4	14	7	*5*	2	19	26	6	

- Find:

4	14	7	5	2	19	26	6

partition() Helper Method

```
private static int partition(int[] arr, int first, int last)
{
    int pivot = arr[(first + last)/2];
    int i = first - 1;  // index going left to right
    int j = last + 1;   // index going right to left
    while (true) {
        do {
            i++;
        } while (arr[i] < pivot);
        do {
            j--;
        } while (arr[j] > pivot);
        if (i < j)
            swap(arr, i, j);
        else
            return j;   // arr[j] = end of left array
    }
}
```

Implementation of Quicksort

```
public static void quickSort(int[] arr) {
    qSort(arr, 0, arr.length – 1);
}

private static void qSort(int[] arr, int first, int last) {
    int split = partition(arr, first, last);

    if (first < split)
        qSort(arr, first, split);      // left subarray
    if (last > split + 1)
        qSort(arr, split + 1, last);   // right subarray
}
```

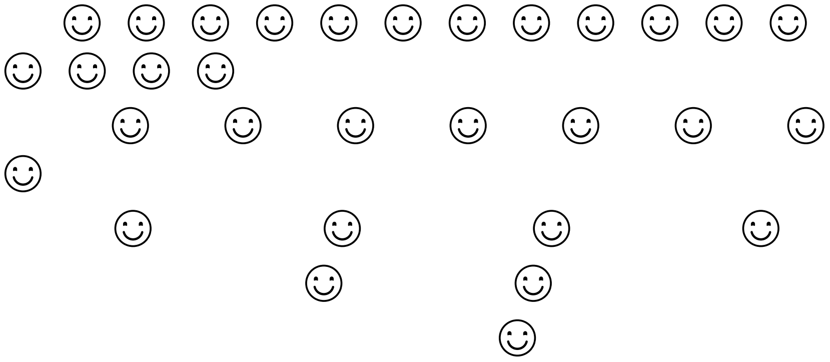
Counting Students: Divide and Conquer

- Everyone stand up.
- You will each carry out the following algorithm:

```
count = 1;
while (you are not the only person standing) {
    find another person who is standing
    if (your first name < other person's first name)
        sit down (break ties using last names)
    else
        count = count + the other person's count
}
if (you are the last person standing)
    report your final count
```

Counting Students: Divide and Conquer (cont.)

- At each stage of the "joint algorithm", the problem size is divided in half.



- How many stages are there as a function of the number of students, n?

- This approach benefits from the fact that you perform the

A Quick Review of Logarithms

- $\log_b n$ = the exponent to which b must be raised to get n
  - $\log_b n = p$ if $b^p = n$
  - examples: $\log_2 8 = 3$ because $2^3 = 8$
    $\log_{10} 10000 = 4$ because $10^4 = 10000$
- Another way of looking at logs:
  - let's say that you repeatedly divide n by b (using integer division)
  - $\log_b n$ is an upper bound on the number of divisions needed to reach 1
  - example: $\log_2 18$ is approx. 4.17
    18/2 = 9   9/2 = 4   4/2 = 2   2/2 = 1

A Quick Review of Logs (cont.)

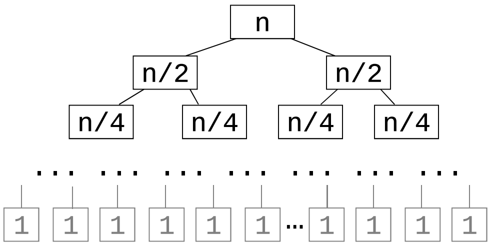
- If the number of operations performed by an algorithm is proportional to $\log_b n$ for any base b, we say it is a *O*(log n) algorithm – dropping the base.
- $\log_b n$ grows much more slowly than n

n	$\log_2 n$
2	1
1024 (1K)	10
1024*1024 (1M)	20

- Thus, for large values of n:
  - a *O*(log n) algorithm is much faster than a *O*(n) algorithm
  - a *O*(n log n) algorithm is much faster than a *O*(n^2) algorithm
- We can also show that an *O*(n log n) algorithm is faster than a *O*($n^{1.5}$) algorithm like Shell sort.

Time Analysis of Quicksort

- Partitioning an array requires n comparisons, because each element is compared with the pivot.
- *best case:* partitioning always divides the array in half
  - repeated recursive calls give:



comparisons

n

2*(n/2) = n

4*(n/4) = n

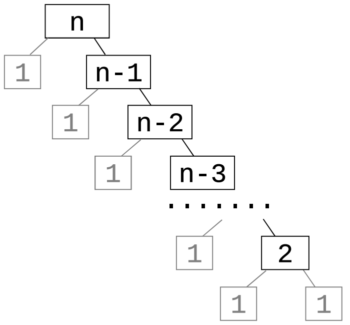
...

0

- at each "row" except the bottom, we perform n comparisons
- there are ________ rows that include comparisons
- C(n) = ?
- Similarly, M(n) and running time are both ___________

Time Analysis of Quicksort (cont.)

- *worst case:* pivot is always the smallest or largest element
  - one subarray has `1` element, the other has `n - 1`
  - repeated recursive calls give:



comparisons

n

n-1

n-2

n-3

...

2

  - $C(n) = \sum_{i=2}^{n} i = O(n^2)$. $M(n)$ and run time are also $O(n^2)$.

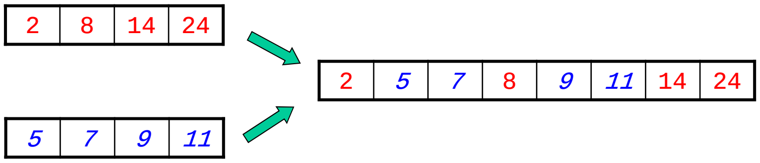
- *average case* is harder to analyze
  - $C(n) > n \log_2 n$, but it's still $O(n \log n)$

Mergesort

- All of the comparison-based sorting algorithms that we've seen thus far have sorted the array in place.
  - used only a small amount of additional memory

- Mergesort is a sorting algorithm that requires an additional temporary array of the same size as the original one.
  - it needs $O(n)$ additional space, where n is the array size

- It is based on the process of *merging* two sorted arrays into a single sorted array.
  - example:



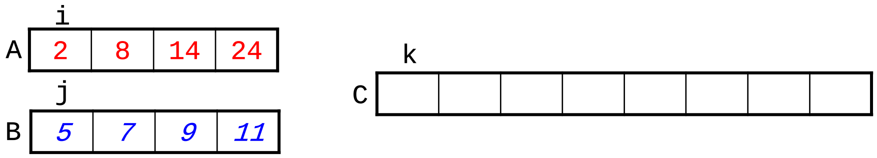
2	8	14	24

5	*7*	*9*	*11*

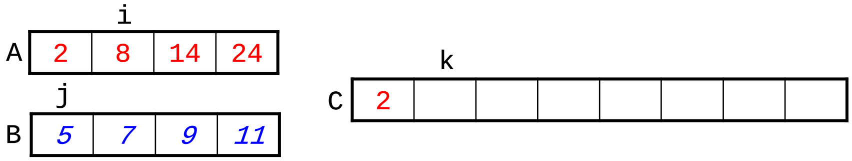
2	*5*	*7*	8	*9*	*11*	14	24

Merging Sorted Arrays

- To merge sorted arrays A and B into an array C, we maintain three indices, which start out on the first elements of the arrays:

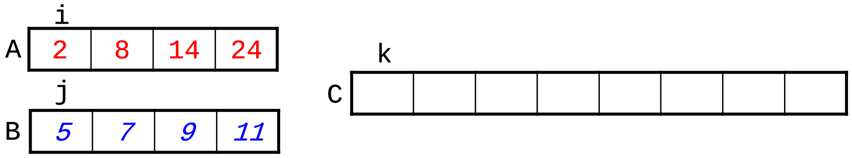


- We repeatedly do the following:
  - compare A[i] and B[j]
  - copy the smaller of the two to C[k]
  - increment the index of the array whose element was copied
  - increment k

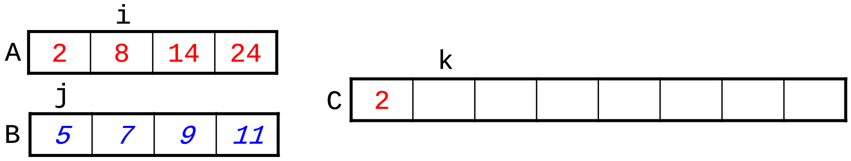


Merging Sorted Arrays (cont.)

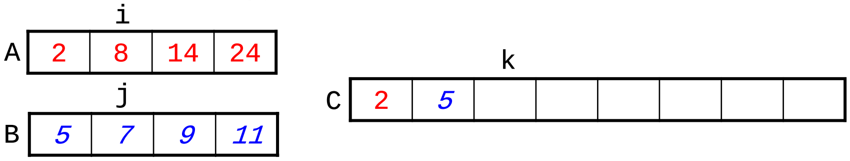
- Starting point:



- After the first copy:

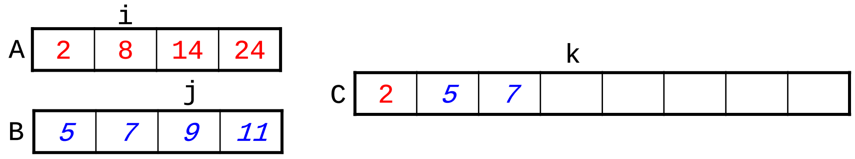


- After the second copy:

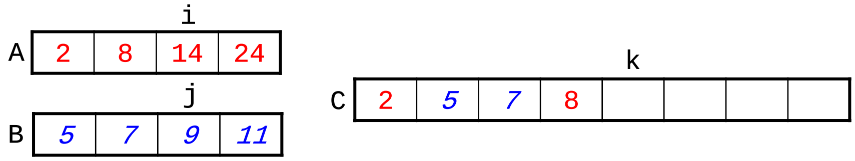


Merging Sorted Arrays (cont.)

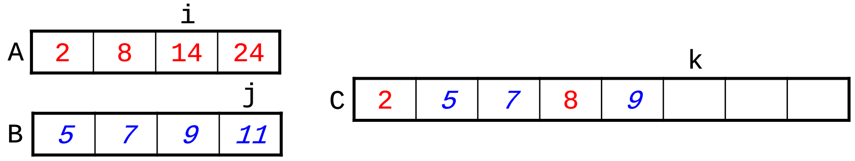
- After the third copy:



- After the fourth copy:

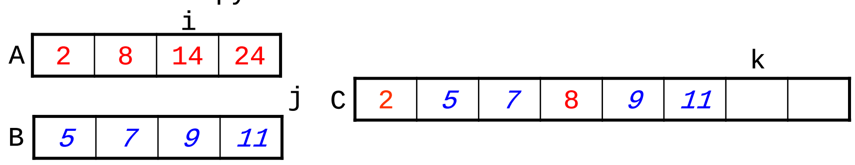


- After the fifth copy:

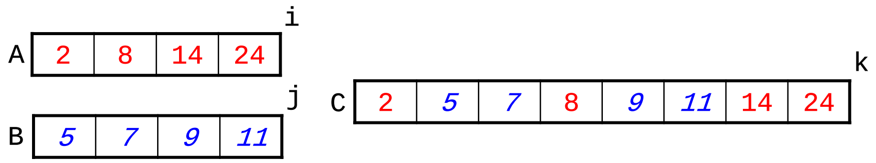


Merging Sorted Arrays (cont.)

- After the sixth copy:

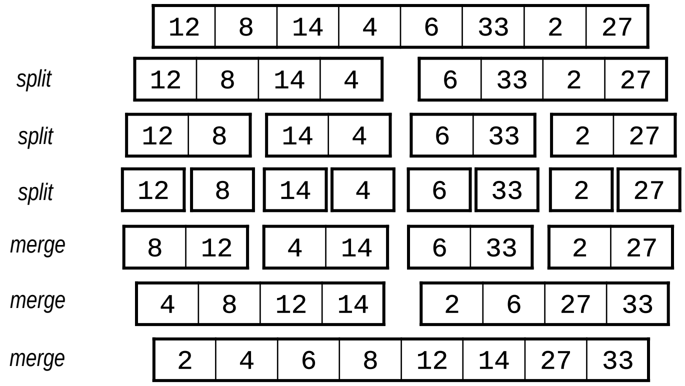


- There's nothing left in B, so we simply copy the remaining elements from A:



Divide and Conquer

- Like quicksort, mergesort is a divide-and-conquer algorithm.
  - *divide:* split the array in half, forming two subarrays
  - *conquer:* apply mergesort recursively to the subarrays, stopping when a subarray has a single element
  - *combine:* merge the sorted subarrays



12 8 14 4 6 33 2 27

split 12 8 14 4 | 6 33 2 27

split 12 8 | 14 4 | 6 33 | 2 27

split 12 | 8 | 14 | 4 | 6 | 33 | 2 | 27

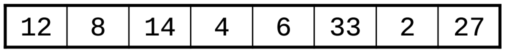
merge 8 12 | 4 14 | 6 33 | 2 27

merge 4 8 12 14 | 2 6 27 33

merge 2 4 6 8 12 14 27 33

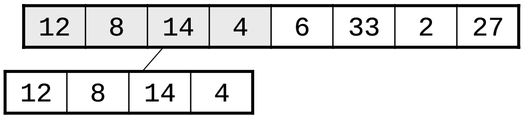
Tracing the Calls to Mergesort

the initial call is made to sort the entire array:



12 8 14 4 6 33 2 27

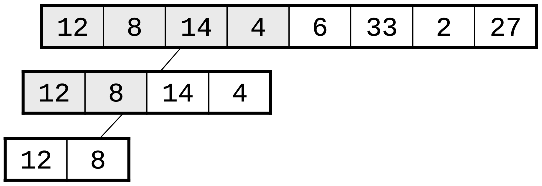
split into two 4-element subarrays, and make a recursive call to sort the left subarray:



12 8 14 4 6 33 2 27

12 8 14 4

split into two 2-element subarrays, and make a recursive call to sort the left subarray:



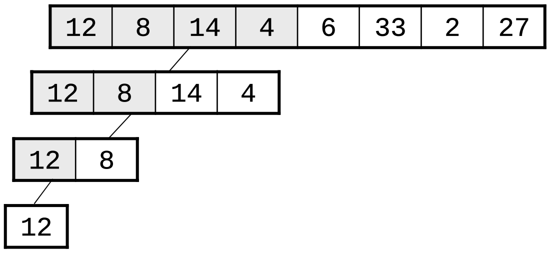
12 8 14 4 6 33 2 27

12 8 14 4

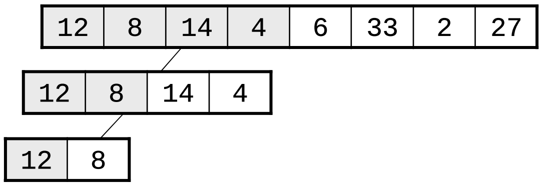
12 8

Tracing the Calls to Mergesort

split into two 1-element subarrays, and make a recursive call to sort the left subarray:

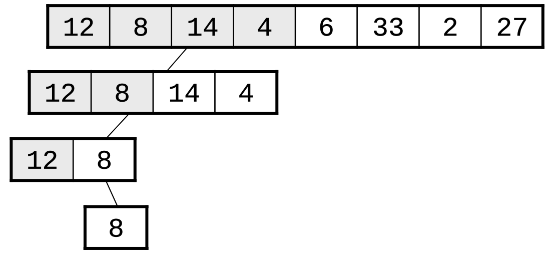


base case, so return to the call for the subarray {12, 8}:

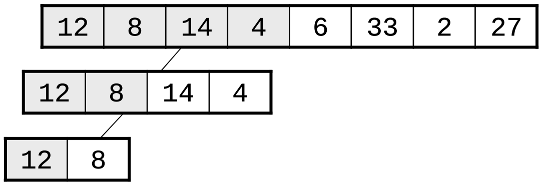


Tracing the Calls to Mergesort

make a recursive call to sort its right subarray:

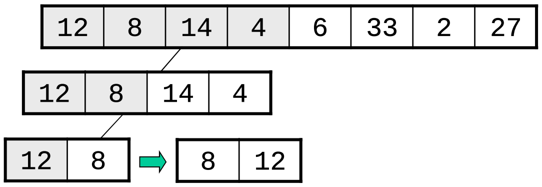


base case, so return to the call for the subarray {12, 8}:

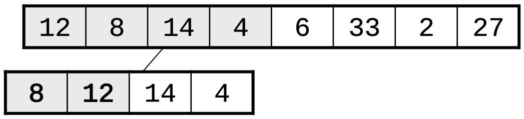


Tracing the Calls to Mergesort

merge the sorted halves of {12, 8}:

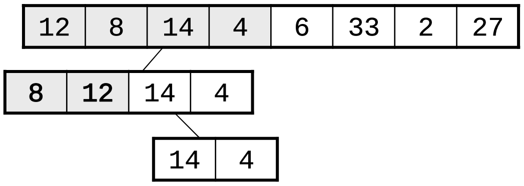


end of the method, so return to the call for the 4-element subarray, which now has a sorted left subarray:

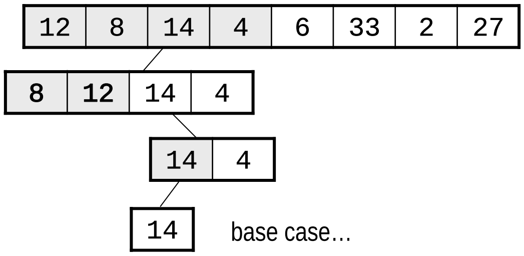


Tracing the Calls to Mergesort

make a recursive call to sort the right subarray of the 4-element subarray



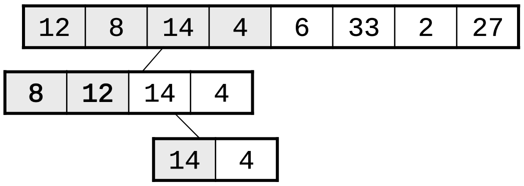
split it into two 1-element subarrays, and make a recursive call to sort the left subarray:



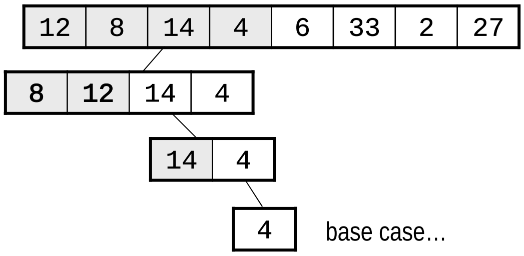
base case…

Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:



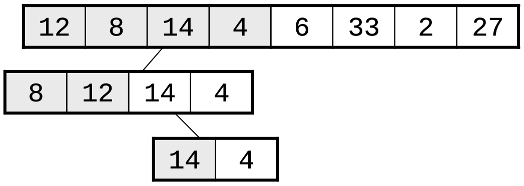
make a recursive call to sort its right subarray:



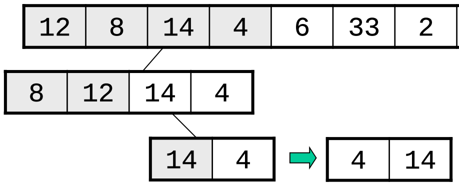
base case…

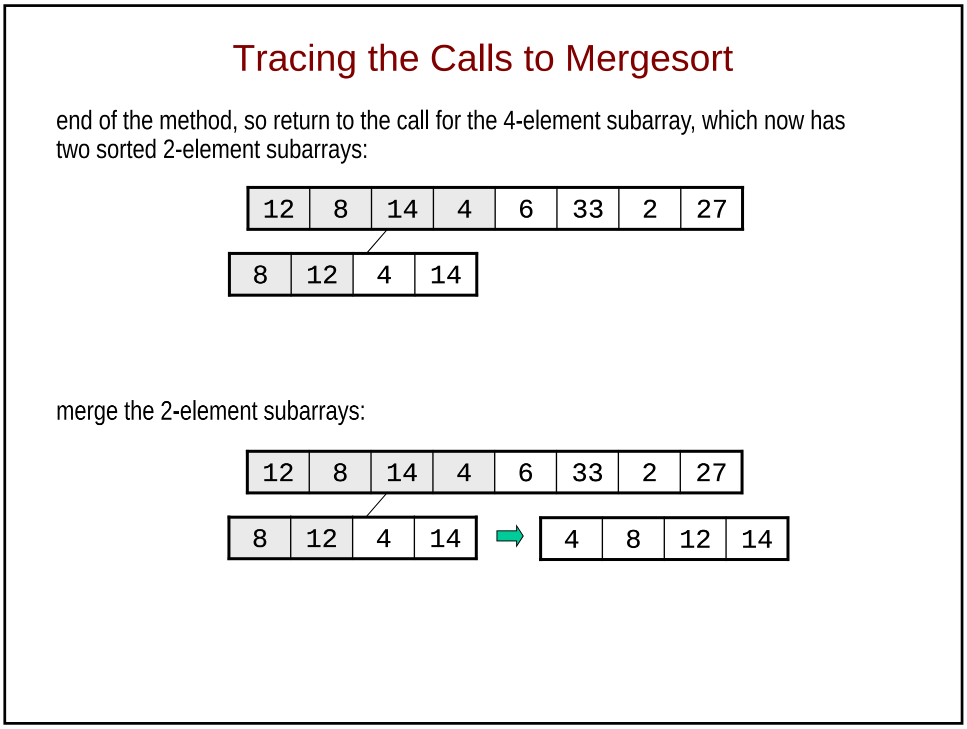
Tracing the Calls to Mergesort

return to the call for the subarray {14, 4}:



merge the sorted halves of {14, 4}:





Tracing the Calls to Mergesort

end of the method, so return to the call for the 4-element subarray, which now has two sorted 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14

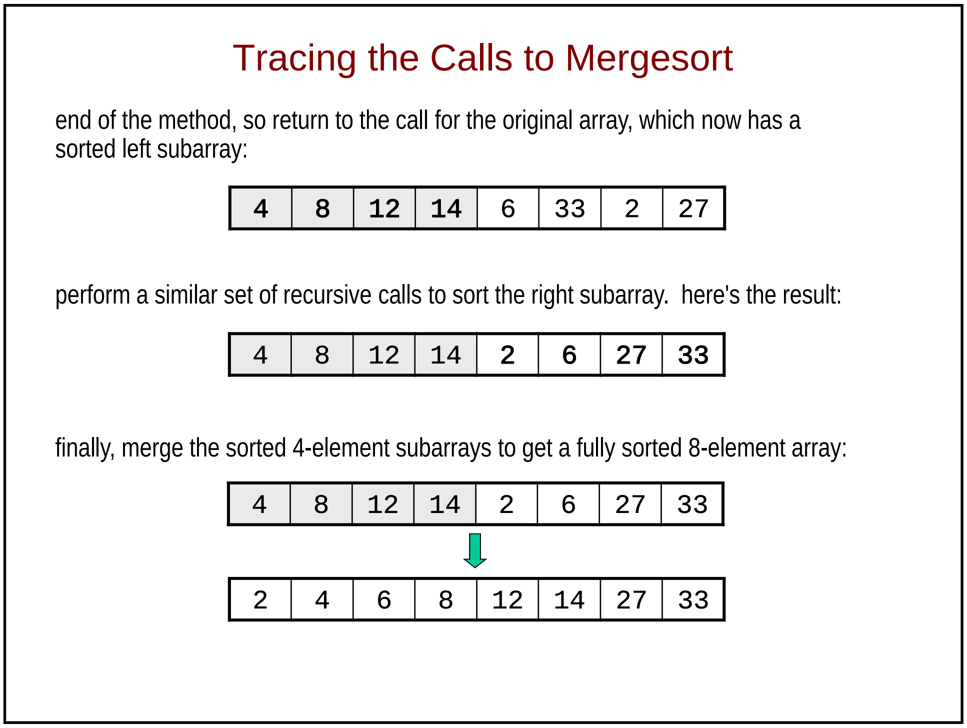
merge the 2-element subarrays:

12	8	14	4	6	33	2	27

8	12	4	14

➡

4	8	12	14



Tracing the Calls to Mergesort

end of the method, so return to the call for the original array, which now has a sorted left subarray:

4	**8**	**12**	**14**	6	33	2	27

perform a similar set of recursive calls to sort the right subarray. here's the result:

4	8	12	14	**2**	**6**	**27**	**33**

finally, merge the sorted 4-element subarrays to get a fully sorted 8-element array:

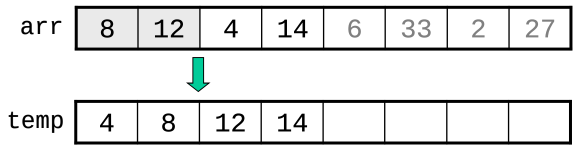
4	8	12	14	2	6	27	33

⬇

2	4	6	8	12	14	27	33

Implementing Mergesort

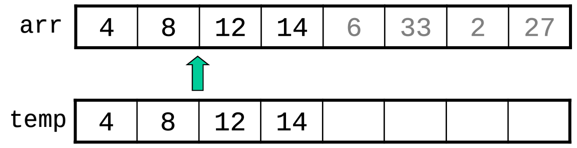
- One approach is to create new arrays for each new set of subarrays, and to merge them back into the array that was split.
- Instead, we'll create a temp. array of the same size as the original.
  - pass it to each call of the recursive mergesort method
  - use it when merging subarrays of the original array:



arr 8 12 4 14 6 33 2 27

temp 4 8 12 14

  - after each merge, copy the result back into the original array:



arr 4 8 12 14 6 33 2 27

temp 4 8 12 14

A Method for Merging Subarrays

```
private static void merge(int[] arr, int[] temp,
  int leftStart, int leftEnd, int rightStart, int rightEnd) {
    int i = leftStart;    // index into left subarray
    int j = rightStart;   // index into right subarray
    int k = leftStart;    // index into temp

    while (i <= leftEnd && j <= rightEnd) {
        if (arr[i] < arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while (i <= leftEnd)
        temp[k++] = arr[i++];

    while (j <= rightEnd)
        temp[k++] = arr[j++];

    for (i = leftStart; i <= rightEnd; i++)
        arr[i] = temp[i];
}
```

Methods for Mergesort

- We use a wrapper method to create the temp. array, and to make the initial call to a separate recursive method:

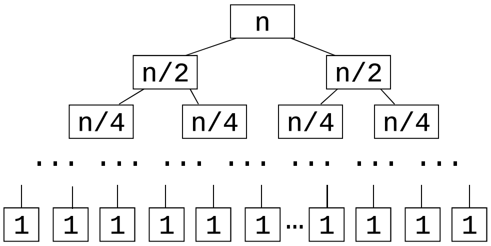
```
public static void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    mSort(arr, temp, 0, arr.length - 1);
}
```

- Let's implement the recursive method together:

```
private static void mSort(int[] arr, int[] temp,
  int start, int end) {









}
```

Time Analysis of Mergesort

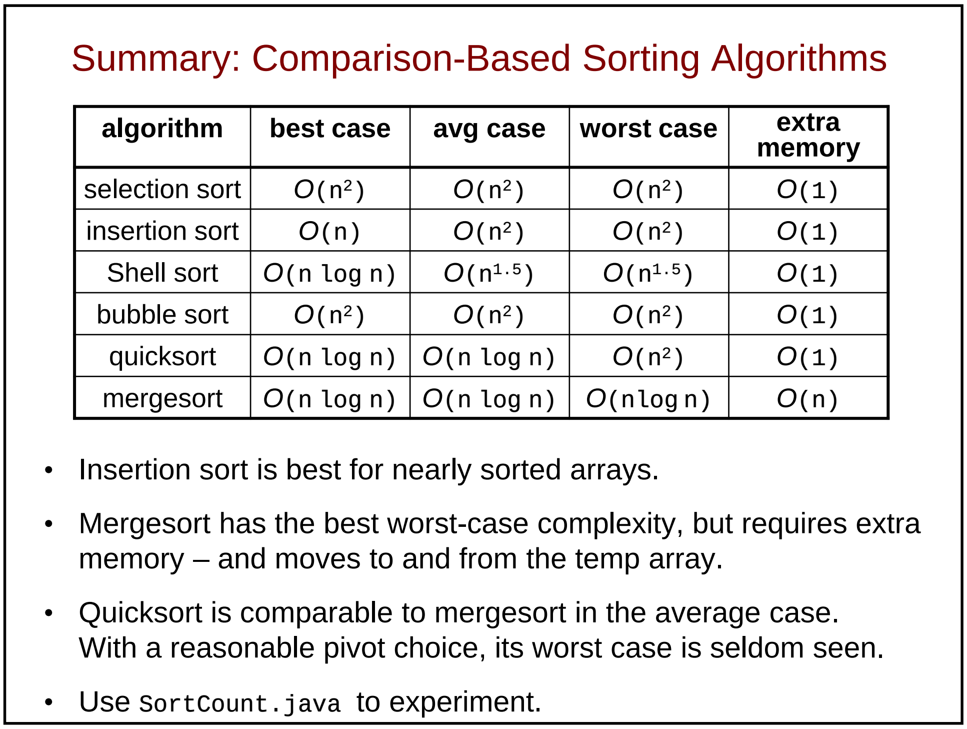
- Merging two halves of an array of size n requires 2n moves. Why?
- Mergesort repeatedly divides the array in half, so we have the following call tree (showing the sizes of the arrays):



moves

```
  2n
2*2*(n/2) = 2n
4*2*(n/4) = 2n
   ...
```

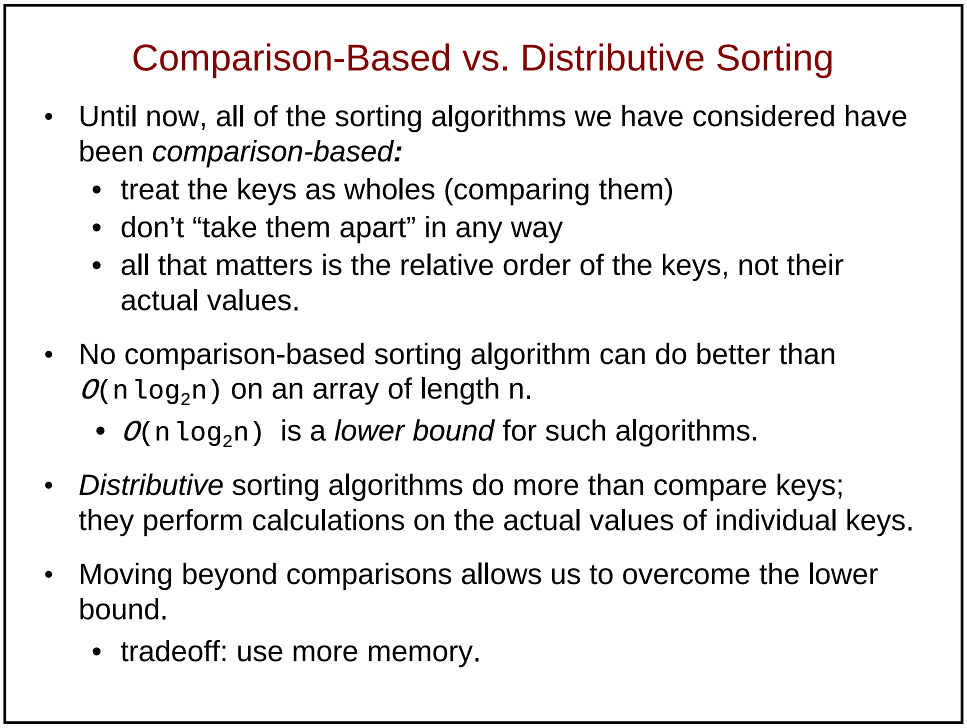
  - at all but the last level of the call tree, there are 2n moves
  - how many levels are there?
  - `M(n)` = ?
  - `C(n)` = ?



Summary: Comparison-Based Sorting Algorithms

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n\log n)$	$O(n)$

- Insertion sort is best for nearly sorted arrays.
- Mergesort has the best worst-case complexity, but requires extra memory – and moves to and from the temp array.
- Quicksort is comparable to mergesort in the average case. With a reasonable pivot choice, its worst case is seldom seen.
- Use `SortCount.java` to experiment.



Comparison-Based vs. Distributive Sorting

- Until now, all of the sorting algorithms we have considered have been *comparison-based:*
  - treat the keys as wholes (comparing them)
  - don't "take them apart" in any way
  - all that matters is the relative order of the keys, not their actual values.
- No comparison-based sorting algorithm can do better than $O(n \log_2 n)$ on an array of length n.
  - $O(n \log_2 n)$ is a *lower bound* for such algorithms.
- *Distributive* sorting algorithms do more than compare keys; they perform calculations on the actual values of individual keys.
- Moving beyond comparisons allows us to overcome the lower bound.
  - tradeoff: use more memory.

Distributive Sorting Example: Radix Sort

- Relies on the representation of the data as a sequence of **m** quantities with **k** possible values.
- Examples:

	m	**k**
• integer in range 0 ... 999	3	10
• string of 15 upper-case letters	15	26
• 32-bit integer	32	2 (in binary)
	4	256 (as bytes)

- Strategy: Distribute according to the last element in the sequence, then concatenate the results:

```
         33   41   12   24   31   14   13   42   34

get:     41   31  |  12   42  |  33   13  |  24   14   34
```

- Repeat, moving back one digit each time:

```
get:                      |          |                    |
```

Analysis of Radix Sort

- Recall that we treat the values as a sequence of `m` quantities with `k` possible values.
- Number of operations is *O*(`n*m`) for an array with `n` elements
  - better than *O*(`n log n`) when `m < log n`
- Memory usage increases as `k` increases.
  - `k` tends to increase as `m` decreases
  - tradeoff: increased speed requires increased memory usage

Big-*O* Notation Revisited

- We've seen that we can group functions into classes by focusing on the fastest-growing term in the expression for the number of operations that they perform.
  - e.g., an algorithm that performs $n^2/2 - n/2$ operations is a $O(n^2)$-time or quadratic-time algorithm

- Common classes of algorithms:

name	example expressions	big-O notation
constant time	$1, 7, 10$	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
$n\log n$ time	$4n\log_2 n$, $n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
cubic time	$n^2 + 3n^3$, $5n^3 - 5$	$O(n^3)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$
factorial time	$3n!$, $5n + n!$	$O(n!)$

slower

How Does the Number of Operations Scale?

- Let's say that we have a problem size of 1000, and we measure the number of operations performed by a given algorithm.
- If we double the problem size to 2000, how would the number of operations performed by an algorithm increase if it is:
  - $O(n)$-time
  - $O(n^2)$-time
  - $O(n^3)$-time
  - $O(\log_2 n)$-time
  - $O(2^n)$-time

How Does the Actual Running Time Scale?

- How much time is required to solve a problem of size n?
  - assume that each operation requires 1 μsec (1 x 10^{-6} sec)

time function	problem size (n) 10	20	30	40	50	60
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	.00006 s
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	.0036 s
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	.001 s	1.0 s	17.9 min	12.7 days	35.7 yrs	36,600 yrs

- sample computations:
  - when n = 10, an n^2 algorithm performs 10^2 operations.
    10^2 * (1 x 10^{-6} sec) = .0001 sec
  - when n = 30, a 2^n algorithm performs 2^{30} operations.
    2^{30} * (1 x 10^{-6} sec) = 1073 sec = 17.9 min

What's the Largest Problem That Can Be Solved?

- What's the largest problem size n that can be solved in a given time T? (again assume 1 μsec per operation)

time function	time available (T) 1 min	1 hour	1 week	1 year
n	60,000,000	3.6 x 10^9	6.0 x 10^{11}	3.1 x 10^{13}
n^2	7745	60,000	777,688	5,615,692
n^5	35	81	227	500
2^n	25	31	39	44

- sample computations:
  - 1 hour = 3600 sec
    that's enough time for 3600/(1 x 10^{-6}) = 3.6 x 10^9 operations
    - n^2 algorithm:
      n^2 = 3.6 x 10^9 → n = (3.6 x 10^9)$^{1/2}$ = 60,000
    - 2^n algorithm:
      2^n = 3.6 x 10^9 → n = $\log_2$(3.6 x 10^9) ~= 31

Unit 8, Part 1

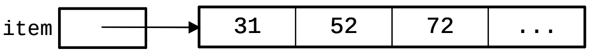
Linked Lists

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

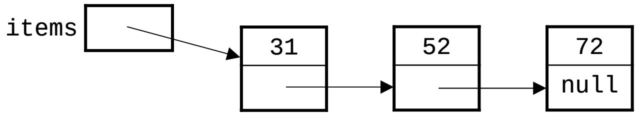
Representing a Sequence of Data

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues
- Most common representation = an array
- Advantages of using an array:
  - easy and efficient access to *any* item in the sequence
    - `item[i]` gives you the item at position i
    - every item can be accessed in constant time
    - this feature of arrays is known as *random access*
  - very compact (but can waste space if positions are empty)
- Disadvantages of using an array:
  - have to specify an initial array size and resize it as needed
  - difficult to insert/delete items at arbitrary positions
    - ex: insert 63 between 52 and 72



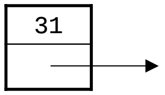
Alternative Representation: A Linked List

- Example:



- A linked list stores a sequence of items in separate *nodes*.
- Each node contains:
  - a single item
  - a “link” (i.e., a reference) to the node containing the next item

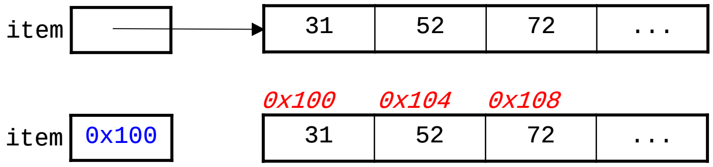
example node:



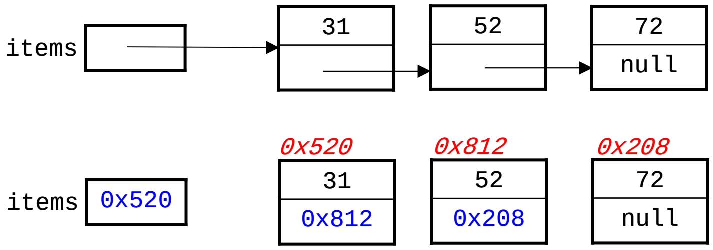
- The last node in the linked list has a link value of `null`.
- The linked list as a whole is represented by a variable that holds a reference to the first node (e.g., `items` in the example above).

Arrays vs. Linked Lists in Memory

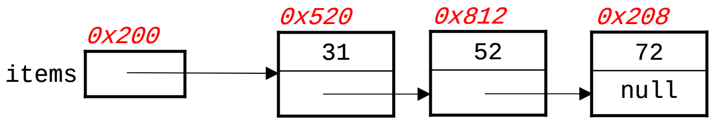
- In an array, the elements occupy consecutive memory locations:



- In a linked list, each node is a distinct object on the heap. The nodes do *not* have to be next to each other in memory. That’s why we need the links to get from one node to the next.



Linked Lists in Memory

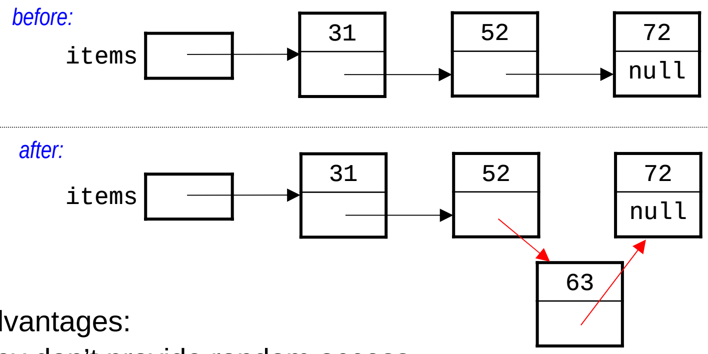


- Here's how the above linked list might actually look in memory:

0x200	0x520	← *the variable* `items`
0x204		
0x208	72	*the last node*
0x212	null	
0x216		
…	…	
0x520	31	*the first node*
0x524	0x812	
0x528		
…	…	
0x812	52	*the second node*
0x816	0x208	

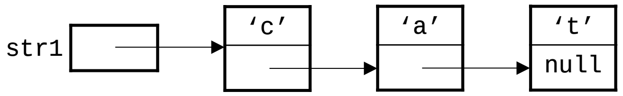
Features of Linked Lists

- They can grow without limit (provided there is enough memory).
- Easy to insert/delete an item – no need to "shift over" other items.
    - for example, to insert 63 between 52 and 72, we just modify the links as needed to accommodate the new node:



- Disadvantages:
    - they don't provide random access
        - need to "walk down" the list to access an item
    - the links take up additional memory

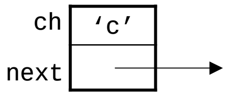
A String as a Linked List of Characters



- Each node in the linked list represents one character.
- Java class for this type of node:

```
public class StringNode {
    private char ch;
    private StringNode next;
    …
}
```

same type as the node itself!



- The string as a whole will be represented by a variable that holds a reference to the node containing the first character.

  example:

```
    StringNode str1;   // shown in the diagram above
```

- Alternative approach: use another class for the string as a whole.

```
    public class LLString {
        StringNode first;
        …
```

(we will *not* do this for strings)

A String as a Linked List (cont.)

- An empty string will be represented by a null value.

  example:

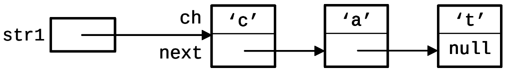
```
  StringNode str2 = null;
```

- We will use *static* methods that take the string as a parameter.
  - e.g., we will write `length(str1)` instead of `str1.length()`
  - outside the class, need the class name: `StringNode.length(str1)`
- This approach is necessary so that the methods can handle empty strings.
  - if `str1 == null`, `length(str1)` will work, but `str1.length()` will throw a `NullPointerException`
- Constructor for our `StringNode` class:

```
    public StringNode(char c, StringNode n) {
        ch = c;
        next = n;
    }
```

A Linked List Is a Recursive Data Structure

- Recursive definition of a linked list: a linked list is either
  - a) empty or
  - b) a single node, followed by a linked list
- Viewing linked lists in this way allows us to write recursive methods that operate on linked lists.

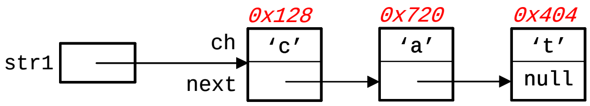


- Example: length of a string
  - length of “cat” = 1 + the length of “at”
  - length of “at” = 1 + the length of “t”
  - length of “t” = 1 + the length of the empty string (which = 0)
- In Java:

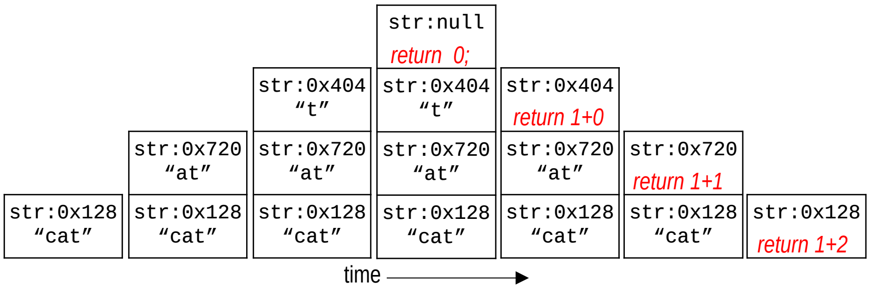
```
public static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```

Tracing `length()`

```
public static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```



- Example: `StringNode.length(str1)`



str:0x128 “cat”

str:0x720 “at”
str:0x128 “cat”

str:0x404 “t”
str:0x720 “at”
str:0x128 “cat”

str:null
return 0;
str:0x404 “t”
str:0x720 “at”
str:0x128 “cat”

str:0x404
return 1+0
str:0x720 “at”
str:0x128 “cat”

str:0x720
return 1+1
str:0x128 “cat”

str:0x128
return 1+2

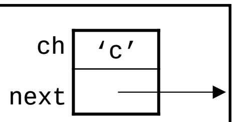
time

Getting the Node at Position i in a Linked List

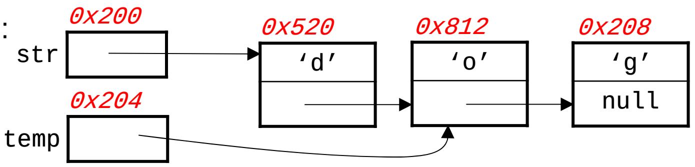
- `getNode(str, i)` – a private helper method that returns a reference to the ith node in the linked list (i == 0 for the first node)
- Recursive approach:
  node at position 2 in the linked list representing “linked”
  = node at position 1 in the linked list representing “inked”
  = node at position 0 in the linked list representing “nked”
  (return a reference to the node containing ‘n’)
- We’ll write the method together:

```
private static StringNode getNode(StringNode str, int i) {







}
```

Review of Variables



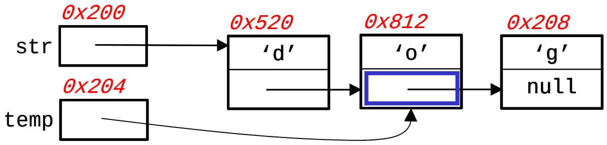
- A variable or variable expression represents both:
  - a “box” or location in memory (the *address* of the variable)
  - the contents of that “box” (the *value* of the variable)
- Practice:



```
StringNode str;   // points to the first node
StringNode temp;  // points to the second node
```

expression	*address*	*value*
`str`	`0x200`	`0x520` (reference to the ‘d’ node)
`str.ch`		
`str.next`		

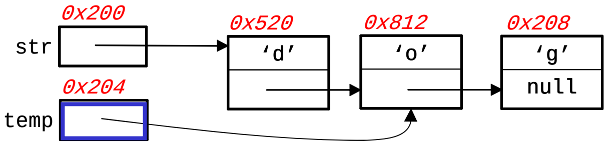
More Complicated Expressions



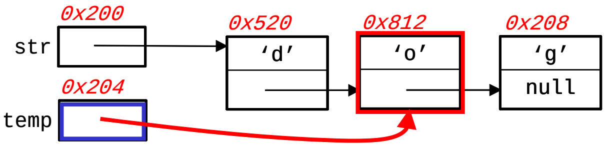
- Example: `temp.next.ch`
- Start with the start of the expression: `temp.next`
  It represents the `next` field of the node to which `temp` refers.
  - address =
  - value =
- Next, consider `temp.next.ch`
  It represents the `ch` field of the node to which `temp.next` refers.
  - address =
  - value =

Dereferencing a Reference

- Each dot causes us to *dereference* the reference represented by the expression preceding the dot.
- Consider again `temp.next.ch`
- Start with temp: `temp.next.ch`

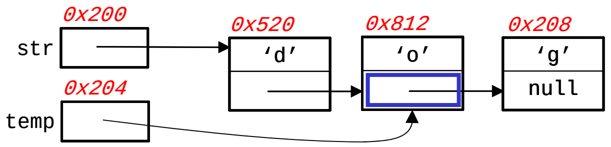


- Dereference: `temp.next.ch`

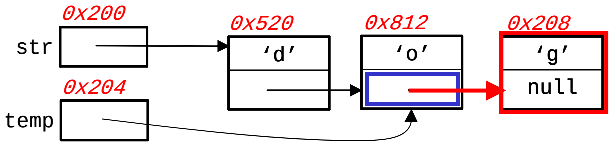


Dereferencing a Reference (cont.)

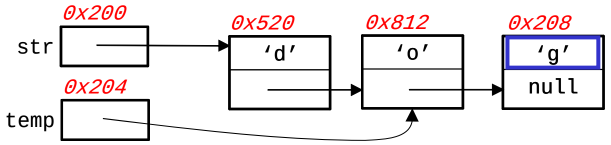
- Get the `next` field: `temp.next.ch`



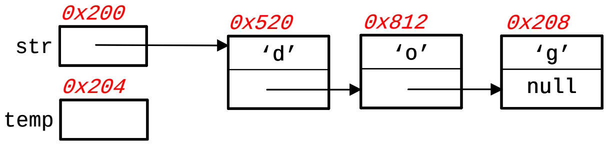
- Dereference: `temp.next.ch`



- Get the `ch` field: `temp.next.ch`



More Complicated Expressions (cont.)



- Here's another example: `str.next.next`
  - address = ?
  - value = ?

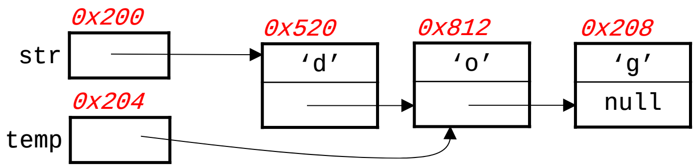
Assignments Involving References

- An assignment of the form

  `var1 = var2;`

  takes the *value* of `var2` and copies it into the location in memory given by the *address* of `var1`.

- Practice:



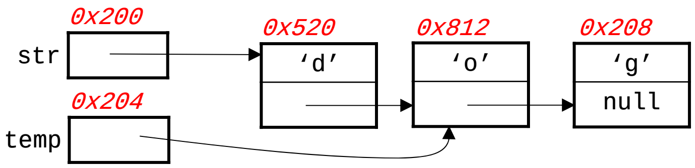
- What happens if we do the following?

  1) `str.next = temp.next;`

  2) `temp = temp.next;`

Assignments Involving References (cont.)

- Beginning with the original diagram, if `temp` didn't already refer to the `'o'` node, what assignment would we need to perform to make it refer to that node?



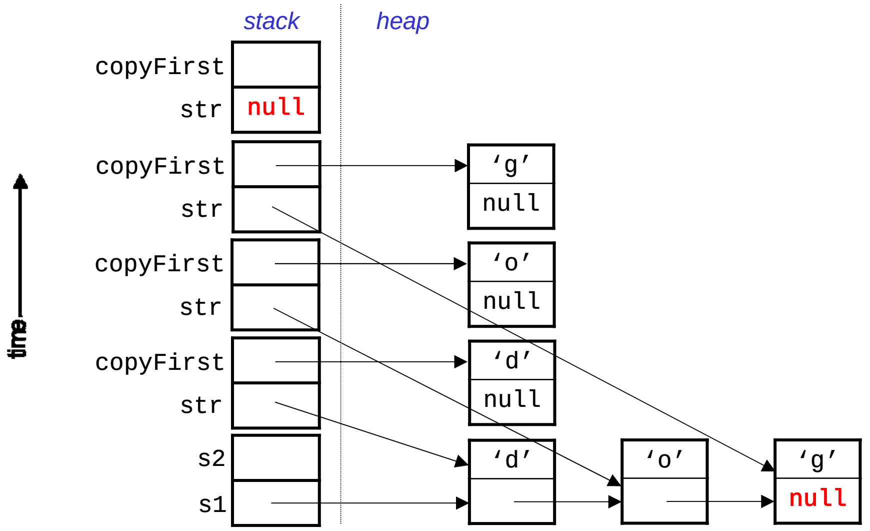
Creating a Copy of a Linked List

- `copy(str)` – create a copy of `str` and return a reference to it
- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```
public static StringNode copy(StringNode str) {
    if (str == null)         // base case
        return null;

    // create the first node of the copy, copying the
    // first character into it
    StringNode copyFirst = new StringNode(str.ch, null);

    // make a recursive call to get a copy the rest and
    // store the result in the first node's next field
    copyFirst.next = copy(str.next);

    return copyFirst;
}
```

Tracing `copy()`: part I

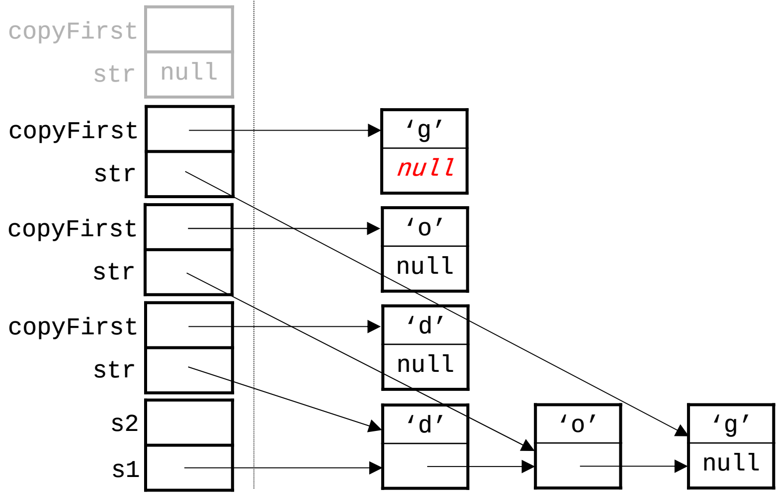
- Example: `StringNode s2 = StringNode.copy(s1);`
- The stack grows as a series of recursive calls are made:



Tracing copy(): part II

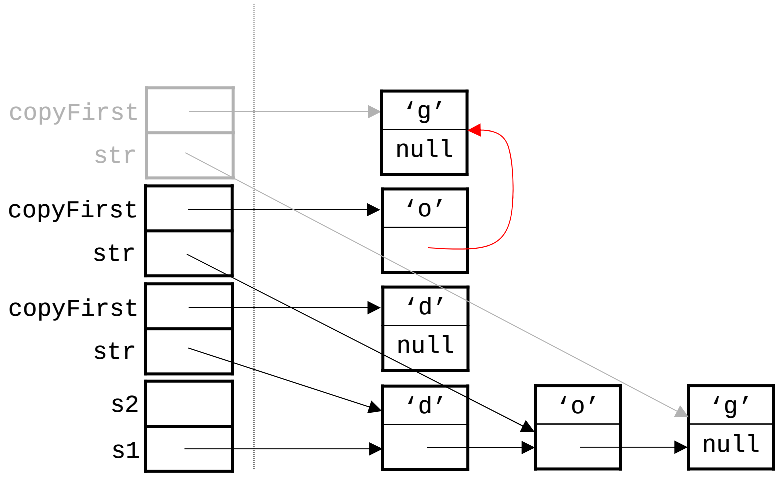
- The base case is reached, so the final recursive call returns `null`.
- This return value is stored in the `next` field of the `‘g’` node:

```
copyFirst.next = copy(str.next)
```



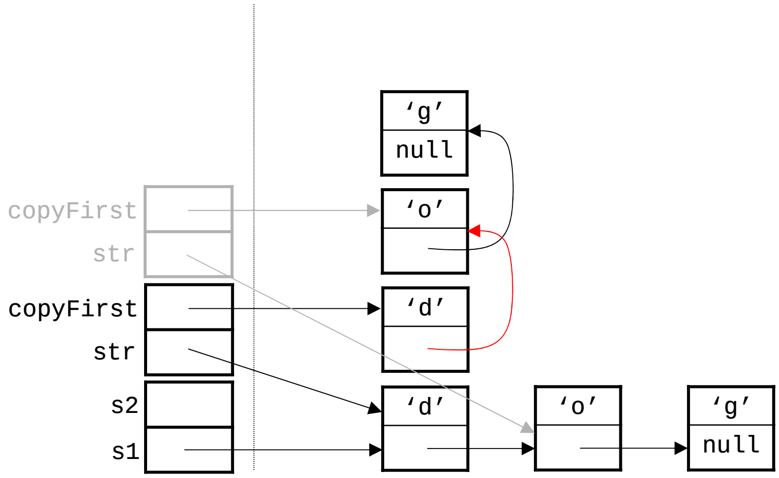
Tracing copy(): part III

- The recursive call that created the `‘g’` node now completes, returning a reference to the `‘g’` node.
- This return value is stored in the `next` field of the `‘o’` node:



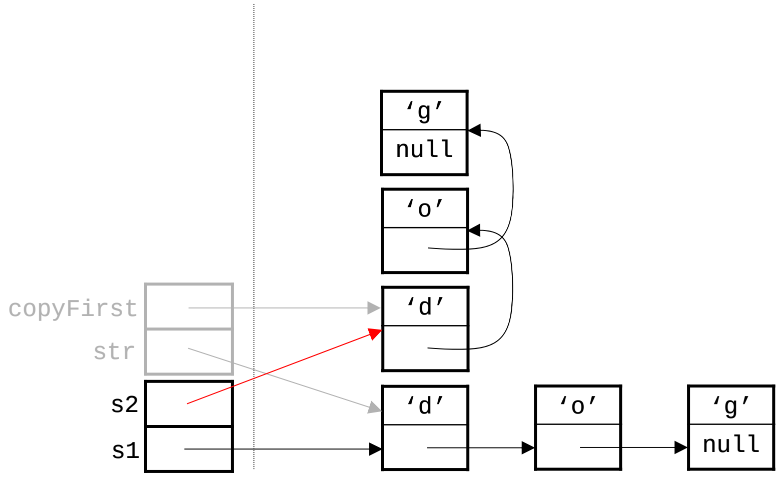
Tracing `copy()`: part IV

- The recursive call that created the `'o'` node now completes, returning a reference to the `'o'` node.
- This return value is stored in the `next` field of the `'d'` node:



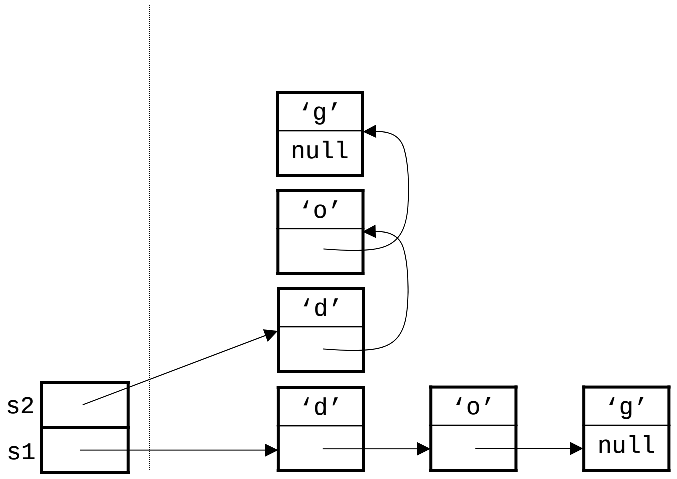
Tracing `copy()`: part V

- The original call (which created the `'d'` node) now completes, returning a reference to the `'d'` node.
- This return value is stored in `s2`:



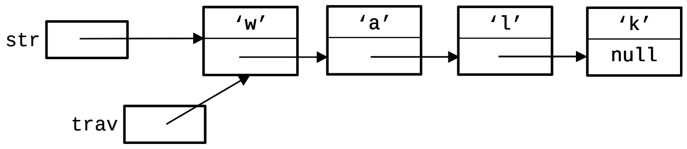
Tracing copy(): Final Result

- `StringNode s2 = StringNode.copy(s1);`
- `s2` now holds a reference to a linked list that is a copy of the linked list to which `s1` holds a reference.



Using Iteration to Traverse a Linked List

- Many tasks require us to traverse or “walk down” a linked list.
- We’ve already seen methods that use recursion to do this.
- It can also be done using iteration (for loops, while loops, etc.).
- We make use of a variable (call it `trav`) that keeps track of where we are in the linked list.

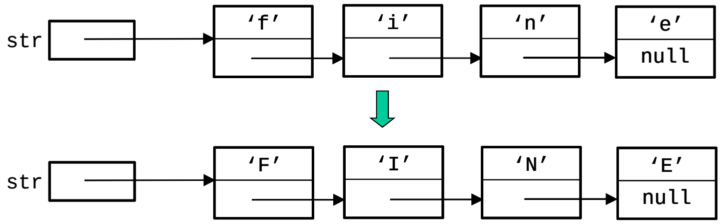


- Template for traversing an entire linked list:

```
StringNode trav = str;    // start with the first node
while (trav != null) {
    // usually do something here
    trav = trav.next;    // move trav down one node
}
```

Example of Iterative Traversal

- `toUpperCase(str)`: converting `str` to all upper-case letters

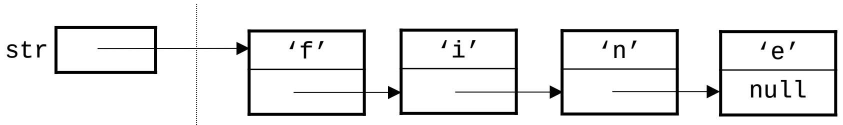


- Java method:

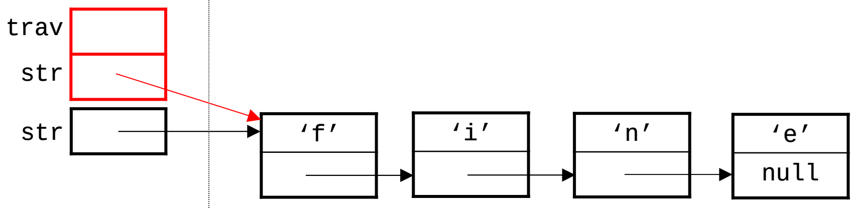
```
public static void toUpperCase(StringNode str) {
     StringNode trav = str;
     while (trav != null) {
         trav.ch = Character.toUpperCase(trav.ch);
         trav = trav.next;
     }
}
```

(makes use of the `toUpperCase()` method from Java's built-in `Character` class)

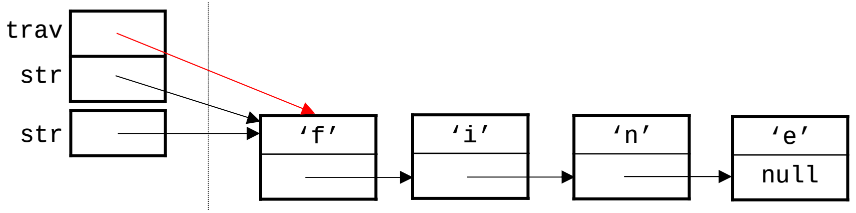
Tracing `toUpperCase()`: Part I



Calling `StringNode.toUpperCase(str)` adds a stack frame to the stack:

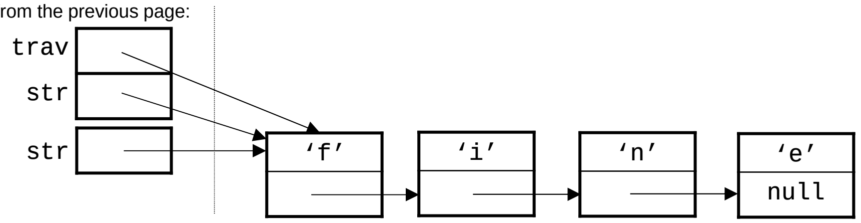


`StringNode trav = str;`



Tracing toUpperCase(): Part II

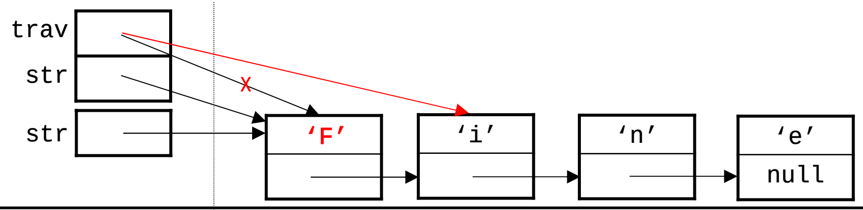
from the previous page:



we enter the while loop:

```
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
```

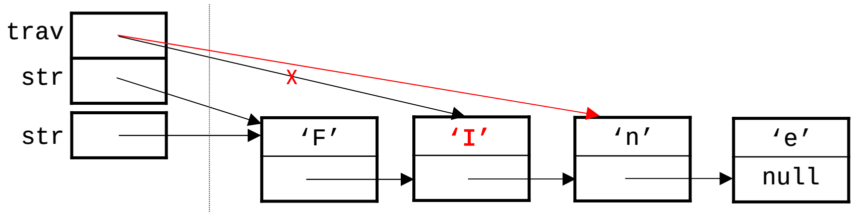
results of the first pass through the loop:



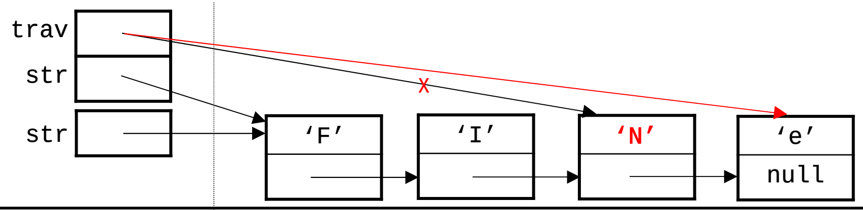
Tracing toUpperCase(): Part III

```
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
```

results of the second pass through the loop:



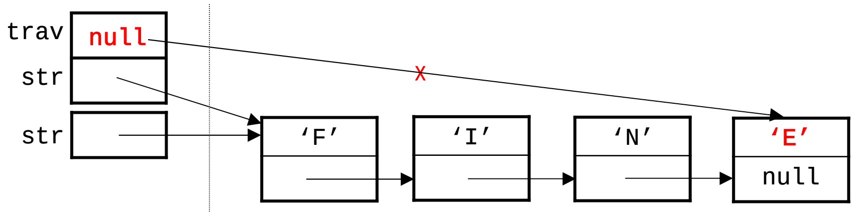
results of the third pass:



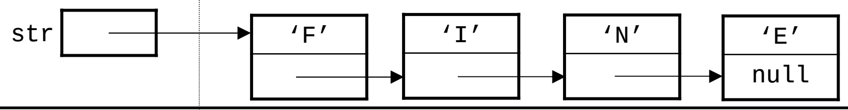
Tracing `toUpperCase()`: Part IV

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

results of the fourth pass through the loop:



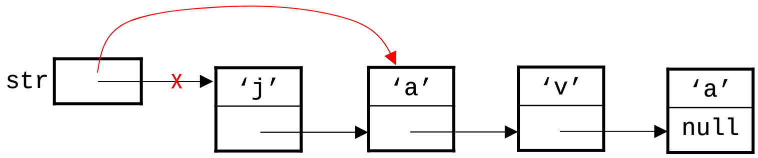
and now `trav == null`, so we break out of the loop and return:



Deleting the Item at Position i

- Special case: `i == 0` (deleting the first item)
- Update our reference to the first node by doing:

```
str = str.next;
```

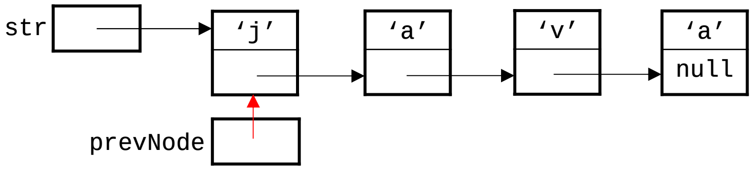


Deleting the Item at Position i (cont.)

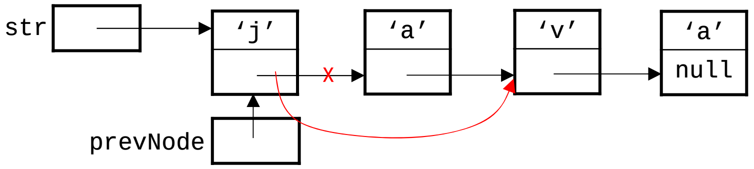
- General case: `i > 0`
- First obtain a reference to the *previous* node:

```
StringNode prevNode = getNode(i – 1);
```

(example for i == 1*)*

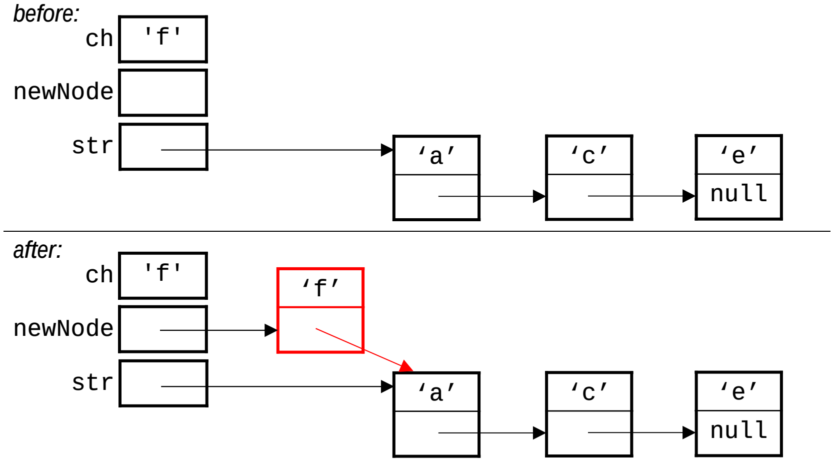


- What remains to be done? (to get the picture below)



Inserting an Item at Position i

- Special case: `i == 0` (insertion at the front of the list)
- What line of code will *create* the new node?

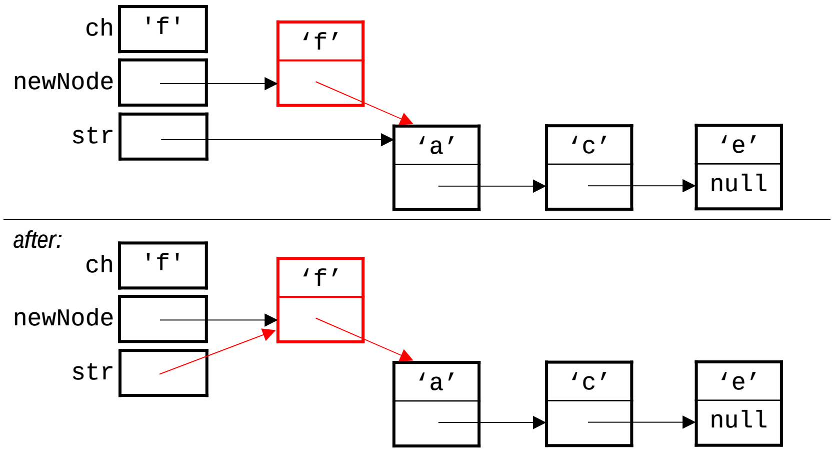


```
StringNode newNode = new StringNode(_______, _______);
```

Inserting an Item at Position i (cont.)

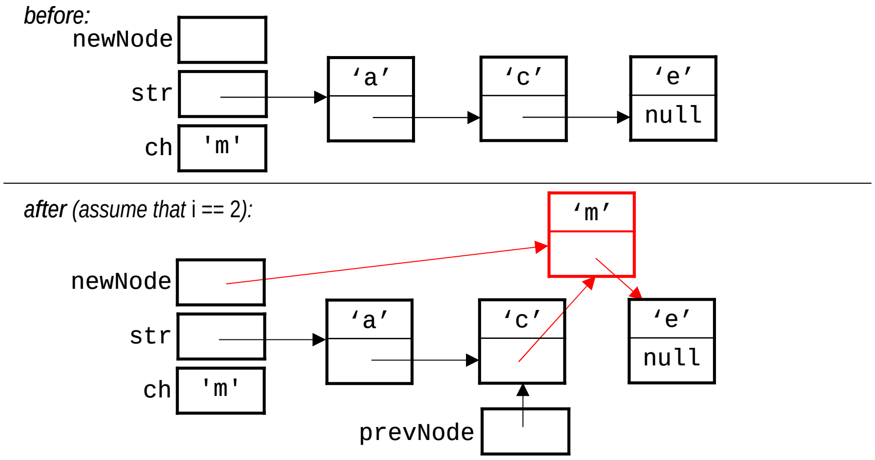
- Special case: `i == 0` (continued)
- What line of code will *insert* the new node?

before (result of previous slide):



Inserting an Item at Position i (cont.)

- General case: `i > 0` (insert *before* the item currently in posn `i`)



```
StringNode prevNode = getNode(i – 1);
StringNode newNode = new StringNode(ch, ________________);
___________________________________ // one more line
```

Returning a Reference to the First Node

- Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```
private static StringNode deleteChar(StringNode str, int i) {
    …
    if (i == 0)                 // case 1
        str = str.next;
    else {                      // case 2
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null)
            prevNode.next = prevNode.next.next;
        …
    }
    return str;
}
```

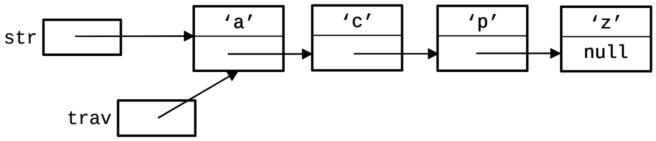
- They do so because the first node may change.
- Invoke as follows:

```
str = StringNode.deleteChar(str, i);
str = StringNode.insertChar(str, i, ch);
```

- If the first node changes, `str` will point to the new first node.

Using a "Trailing Reference" During Traversal

- When traversing a linked list, using a single `trav` reference isn't always good enough.
- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:



- Traverse the list to find the right position:

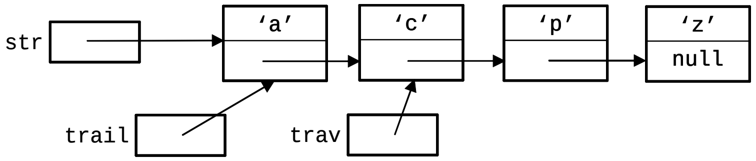
```
StringNode trav = str;
while (trav != null && trav.ch < ch)
    trav = trav.next;
```

- When we exit the loop, where will `trav` point? Can we insert `'n'`?
- The following changed version doesn't work either. Why not?

```
StringNode trav = str;
while (trav != null && trav.next.ch < ch)
    trav = trav.next;
```

Using a “Trailing Reference” (cont.)

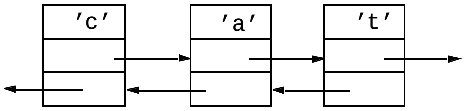
- To get around the problem seen on the previous page, we traverse the list using two different references:
  - `trav`, which we use as before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

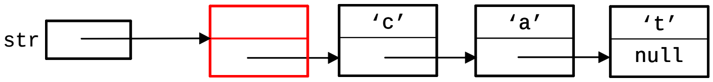
Other Variants of Linked Lists

- Doubly linked list

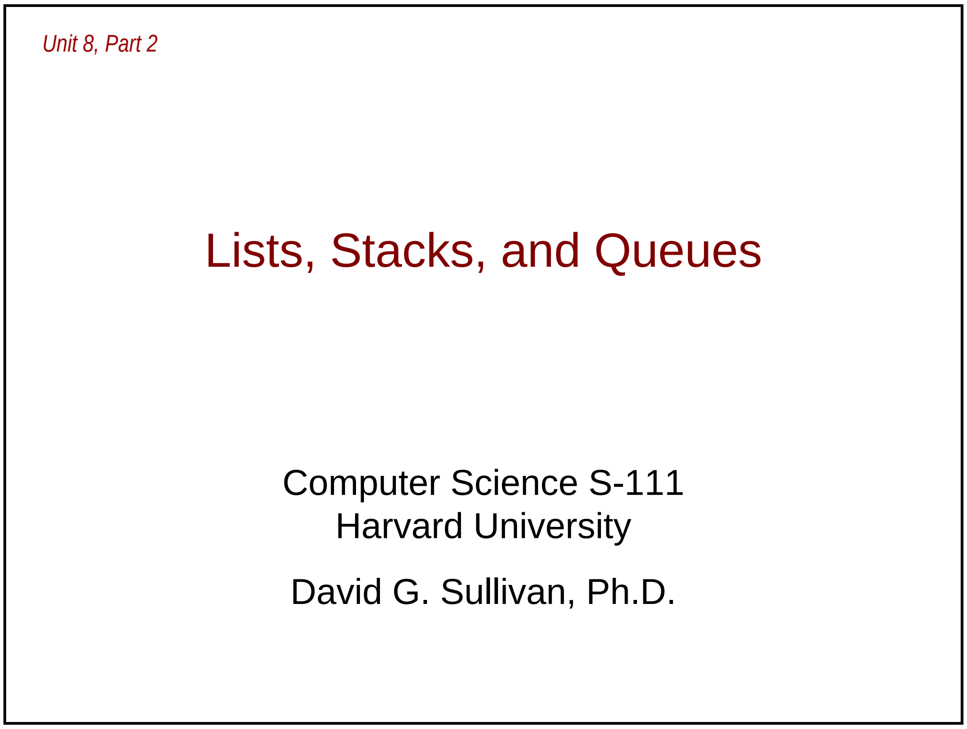


  - add a `prev` reference to each node -- refers to the previous node
  - allows us to “back up” from a given node

- Linked list with a dummy node at the front:



  - the dummy node doesn’t contain a data item
  - it eliminates the need for special cases to handle insertion and deletion at the front of the list
    - more on this in the next set of notes

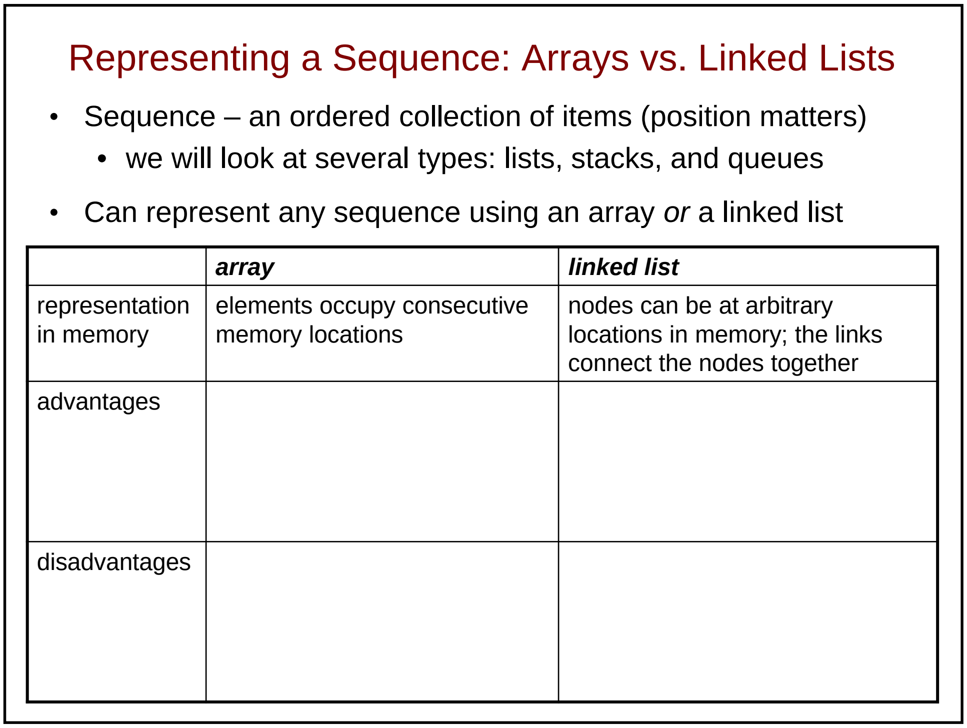


Unit 8, Part 2

Lists, Stacks, and Queues

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.



Representing a Sequence: Arrays vs. Linked Lists

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues
- Can represent any sequence using an array *or* a linked list

	array	***linked list***
representation in memory	elements occupy consecutive memory locations	nodes can be at arbitrary locations in memory; the links connect the nodes together
advantages		
disadvantages		

A List as an Abstract Data Type

- list = a sequence of items that supports at least the following functionality:
  - accessing an item at an arbitrary position in the sequence
  - adding an item at an arbitrary position
  - removing an item at an arbitrary position
  - determining the number of items in the list (the list's *length*)
- ADT: specifies *what* a list will do, without specifying the implementation

Review: Specifying an ADT Using an Interface

- Recall that in Java, we can use an interface to specify an ADT:

```
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    int length();
    …
}
```

- We make any implementation of the ADT a class that implements the interface:

```
public class MyList implements List {
    ...
```

- This approach allows us to write code that will work with different implementations of the ADT:

```
public static void processList(List l) {
    for (int i = 0; i < l.length(); i++) {
        Object item = l.getItem(i);
        …
```

Our List Interface

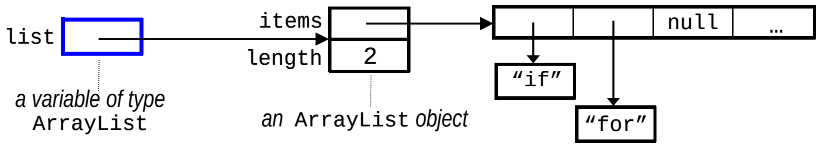
```
public interface List {
    Object getItem(int i);
    boolean addItem(Object item, int i);
    Object removeItem(int i);
    int length();
    boolean isFull();
}
```

- We include an `isFull()` method to test if the list already has the maximum number of items
- Recall that all methods in an interface are assumed to be public.
- The actual interface definition includes comments that describe what each method should do.

Implementing a List Using an Array

```
public class ArrayList implements List {
    private Object[] items;
    private int length;

    public ArrayList(int maxSize) {
        items = new Object[maxSize];
        length = 0;
    }

    public int length() {
        return length;
    }

    public boolean isFull() {
        return (length == items.length);
    }
    ...
}
```

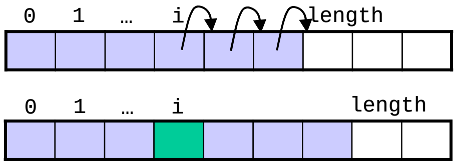
- Sample list:



Adding an Item to an ArrayList

- Adding at position i (shifting items i, i+1, … to the right by one):

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length)
        throw new IndexOutOfBoundsException();
    if (isFull())
        return false;

    // make room for the new item
    for (int j = length – 1; j >= i; j--)
        items[j + 1] = items[j];

    items[i] = item;
    length++;
    return true;
}
```



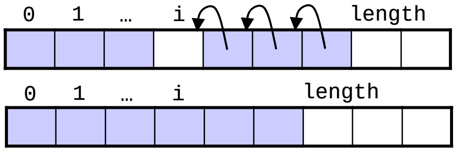
Other ArrayList Methods

- Getting item i:

```
public Object getItem(int i) {
    if (i < 0 || i >= length)
        throw new IndexOutOfBoundsException();
    return items[i];
}
```

- Removing item i (shifting items i+1, i+2, … to the left by one):

```
public Object removeItem(int i) {
    if (i < 0 || i >= length)
        throw new IndexOutOfBoundsException();




}
```



Converting an ArrayList to a String

- The toString() method is designed to allow objects to be displayed in a human-readable format.
- This method is called implicitly when you attempt to print an object or when you perform string concatenation:

```
ArrayList l = new ArrayList();
System.out.println(l);
String str = “My list: ” + l;
System.out.println(str);
```

- A default version of this method is inherited from the Object class.
  - returns a String consisting of the type of the object and a hash code for the object.
- It usually makes sense to override the default version.

toString() Method for the ArrayList Class

```
public String toString() {
    String str = "{";

    if (length > 0) {
        for (int i = 0; i < length - 1; i++)
            str = str + items[i] + ", ";
        str = str + items[length – 1];
    }

    str = str + "}"

    return str;
}
```

- Produces a string of the following form:

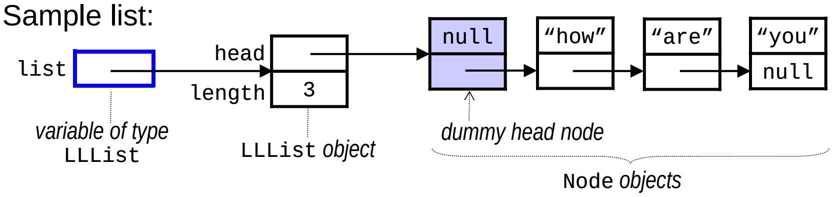
```
{items[0], items[1], … }
```

- Why is the last item added outside the loop?
- Why do we need the if statement?

Implementing a List Using a Linked List

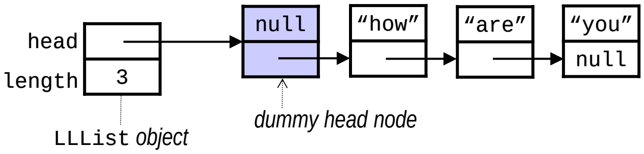
```
public class LLList implements List {
    private Node head;    // dummy head node
    private int length;
    …
}
```

- Sample list:

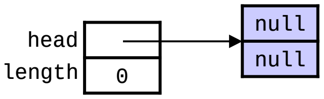


- Differences from the linked list we used for strings:
  - we “embed” the linked list inside another class
    - users of our `LLList` class will never actually touch the nodes
    - users of our `StringNode` class hold a reference to the first node
  - we use a dummy head node
  - we use instance methods instead of static methods
    - `myList.length()` instead of `length(myList)`

Using a Dummy Head Node



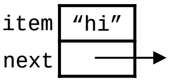
- The dummy head node is always at the front of the linked list.
  - like the other nodes in the linked list, it’s of type `Node`
  - it does *not* store an item
  - it does *not* count towards the length of the list
- An empty `LLList` still has a dummy head node:



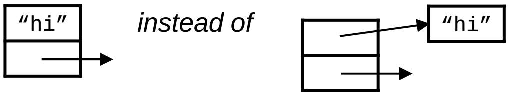
- Using a dummy head node allows us to avoid special cases when adding and removing nodes from the linked list.

An Inner Node Class

```
public class LLList implements List {
    private class Node {
        private Object item;
        private Node next;

        private Node(Object i, Node n) {
            item = i;
            next = n;
        }
    }
    ...
}
```



- We make `Node` an *inner class*, defining it within `LLList`.
    - allows the `LLList` methods to directly access `Node`'s private members, while restricting all other access
    - the compiler creates this class file: `LLList$Node.class`
- For simplicity, our diagrams show the items inside the nodes.



Other Details of Our `LLList` Class

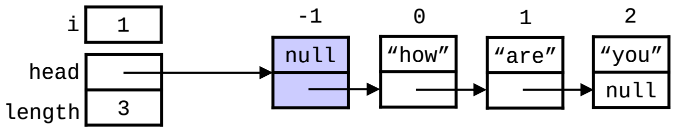
```
public class LLList implements List {
    private class Node {
        ...
    }

    private Node head;
    private int length;

    public LLList() {
        head = new Node(null, null);
        length = 0;
    }

    public boolean isFull() {
        return false;
    }
}
```

- Unlike `ArrayList`, there's no need to preallocate space for the items. The constructor simply creates the dummy head node.
- The linked list can grow indefinitely, so the list is never full!

Getting a Node

- Private helper method for getting node `i`
  - to get the dummy head node, use `i = -1`

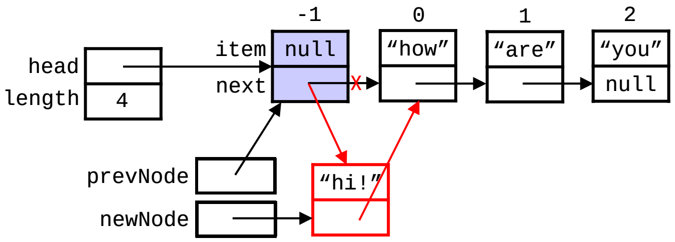
```
private Node getNode(int i) {
    // private method, so we assume i is valid!

    Node trav = __________;
    int travIndex = -1;
    while ( _________________ ) {
        travIndex++;
        ________________________;
    }
    return trav;
}
```



Adding an Item to an `LLList`

```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length)
        throw new IndexOutOfBoundsException();
    Node newNode = new Node(item, null);
    Node prevNode = getNode(i – 1);
    newNode.next = prevNode.next;
    prevNode.next = newNode;
    length++;
    return true;
}
```

- This works even when adding at the front of the list (`i == 0`):



`addItem()` Without a Dummy Head Node

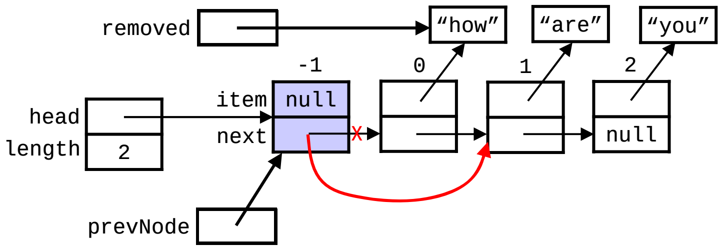
```
public boolean addItem(Object item, int i) {
    if (i < 0 || i > length)
        throw new IndexOutOfBoundsException();

    Node newNode = new Node(item, null);

    if (i == 0) {                    // case 1: add to front
        newNode.next = first;
        first = newNode;
    } else {                         // case 2: i > 0
        Node prevNode = getNode(i – 1);
        newNode.next = prevNode.next;
        prevNode.next = newNode;
    }

    length++;
    return true;
}
```

(instead of a reference named `head` *to the dummy head node, this implementation maintains a reference named* `first` *to the first node, which does hold an item).*

Removing an Item from an `LLList`

```
public Object removeItem(int i) {
    if (i < 0 || i >= length)
        throw new IndexOutOfBoundsException();
    Node prevNode = getNode(i – 1);
    Object removed = prevNode.next.item;
    ___________________________      // what line goes here?
    length--;
    return removed;
}
```

- This works even when removing the first item (`i == 0`):



toString() Method for the LLList Class

```
public String toString() {
    String str = "{";

    // what should go here?










    str = str + " }"

    return str;
}
```

Counting the Number of Occurrences of an Item

- One possible approach:

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        for (int i = 0; i < l.length(); i++) {
            Object itemAt = l.getItem(i);
            if (itemAt.equals(item))
                numOccur++;
        }
        return numOccur;
    } …
```

- Problem: for `LLList` objects, each call to `getItem()` starts at the head of the list and traverses to item `i`.
  - to access item 0, access 1 node
  - to access item 1, access 2 nodes
  - to access item i, access i+1 nodes
  - if length = n, total nodes accessed = 1 + 2 + … + n = $O(n^2)$

Solution 1: Make `numOccur()` an `LLList` Method

```
public class LLList {
    public int numOccur(Object item) {
        int numOccur = 0;
        Node trav = head.next;  // skip the dummy head node
        while (trav != null) {
            if (trav.item.equals(item))
                numOccur++;
            trav = trav.next;
        }
        return numOccur;
    } …
```

- Each node is only visited once, so the # of accesses = n = *O*(n)
- Problem: we can't anticipate all of the types of operations that users may wish to perform.
- We would like to give users the general ability to iterate over the list.

Solution 2: Give Access to the Internals of the List

- Make our private helper method `getNode()` a public method.
- Make `Node` a non-inner class and provide getter methods.
- This would allow us to do the following:

```
public class MyClass {
    public static int numOccur(LLList l, Object item) {
        int numOccur = 0;
        Node trav = l.getNode(0);
        while (trav != null) {
            Object itemAt = trav.getItem();
            if (itemAt.equals(item))
                numOccur++;
            trav = trav.getNext();
        }
        return numOccur;
    } …
}
```

- What's wrong with this approach?

Solution 3: Provide an Iterator

- An iterator is an object that provides the ability to iterate over a list *without* violating encapsulation.
- Our iterator class will implement the following interface:

```
public interface ListIterator {
    // Are there more items to visit?
    boolean hasNext();

    // Return next item and advance the iterator.
    Object next();
}
```

- The iterator starts out prepared to visit the first item in the list, and we use `next()` to access the items sequentially.
- Ex: position of the iterator is shown by the cursor symbol (|)

after the iterator `i` *is created:*	|	“do”		“we”		“go”	…
after calling `i.next()`*, which returns* “do”:		“do”	|	“we”		“go”	…
after calling `i.next()`*, which returns* “we”:		“do”		“we”	|	“go”	…

`numOccur()` Using an Iterator

```
public class MyClass {
    public static int numOccur(List l, Object item) {
        int numOccur = 0;
        ListIterator iter = l.iterator();
        while (iter.hasNext()) {
            Object itemAt = iter.next();
            if (itemAt.equals(item))
                numOccur++;
        }
        return numOccur;
    } …
}
```

- The `iterator()` method returns an iterator object that is ready to visit the first item in the list. (Note: we also need to add the header of this method to the `List` interface.)
- Note that `next()` does two things at once:
  - gets an item
  - advances the iterator.

Using an Inner Class for the Iterator

```
public class LLList {
    public ListIterator iterator() {
        return new LLListIterator();
    }

    private class LLListIterator implements ListIterator {
        private Node nextNode;
        private Node lastVisitedNode;

        public LLListIterator() {
            …
    }
}
```

- Using a inner class gives the iterator access to the list's internals.
- Because `LLListIterator` is a private inner class, methods outside `LLList` can't create `LLListIterator` objects or have variables that are declared to be of type `LLListIterator`.
- Other classes use the *interface name* as the declared type, e.g.:

```
    ListIterator iter = l.iterator();
```

`LLListIterator` Implementation

```
private class LLListIterator implements ListIterator {
    private Node nextNode;
    private Node lastVisitedNode;

    public LLListIterator() {
        nextNode = head.next;    // skip over head node
        lastVisitedNode = null;
    }
    …
}
```

- Two instance variables:
  - `nextNode` keeps track of the next node to visit
  - `lastVisitedNode` keeps track of the most recently visited node
    - not needed by `hasNext()` and `next()`
    - what iterator operations might we want to add that *would* need this reference?

`LLListIterator` Implementation (cont.)

```
private class LLListIterator implements ListIterator {
    private Node nextNode;
    private Node lastVisitedNode;

    public LLListIterator() {
        nextNode = head.next;    // skip over dummy node
        lastVisitedNode = null;
    }

    public boolean hasNext() {
         return (nextNode != null);
    }

    public Object next() {
        if (nextNode == null)
            throw new NoSuchElementException();

        Object item = nextNode.item;
        lastVisited = nextNode;
        nextNode = nextNode.next;

        return item;
    }
}
```

More About Iterators

- In theory, we could write list-iterator methods that were methods of the list class itself.
- Instead, our list-iterator methods are encapsulated within an iterator object.
  - allows us to have multiple iterations active at the same time:

```
ListIterator i = l.iterator();
while (i.hasNext()) {
    ListIterator j = l.iterator();
    while (j.hasNext()) {
        …
```

- Java's built-in *collection classes* all provide iterators.
  - `LinkedList`, `ArrayList`, etc.
  - the built-in `Iterator` interface specifies the iterator methods
    - they include `hasNext()` and `next()` methods like ours

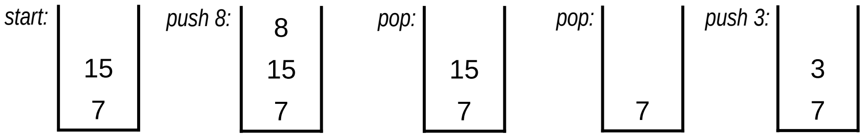
Efficiency of the List Implementations

n = number of items in the list

	`ArrayList`	`LLList`
`getItem()`		
`addItem()`		
`removeItem()`		
space efficiency		

Stack ADT

- A stack is a sequence in which:
  - items can be added and removed only at one end (the *top*)
  - you can only access the item that is currently at the top
- Operations:
  - push: add an item to the top of the stack
  - pop: remove the item at the top of the stack
  - peek: get the item at the top of the stack, but don’t remove it
  - isEmpty: test if the stack is empty
  - isFull: test if the stack is full
- Example: a stack of integers



A Stack Interface: First Version

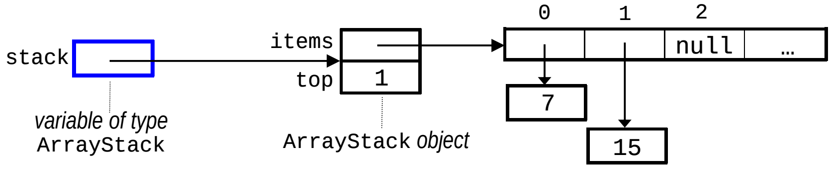
```
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

- `push()` returns `false` if the stack is full, and `true` otherwise.
- `pop()` and `peek()` take no arguments, because we know that we always access the item at the top of the stack.
    - return `null` if the stack is empty.
- The interface provides no way to access/insert/delete an item at an arbitrary position.
    - encapsulation allows us to ensure that our stacks are manipulated only in ways that are consistent with what it means to be stack

Implementing a Stack Using an Array: First Version

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item

    public ArrayStack(int maxSize) {
        items = new Object[maxSize];
        top = -1;
    }
    ...
}
```

- Example: the stack | 15 | 7 | would be represented as follows:



- Items are added from left to right. The instance variable `top` stores the index of the item at the top of the stack.

Limiting a Stack to Objects of a Given Type

- We can do this by using a *generic* interface and class.
- Here is a generic version of our `Stack` interface:

```
public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- It includes a *type variable* `T` in its header and body.
- This type variable is used as a placeholder for the actual type of the items on the stack.

A Generic `ArrayStack` Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable `T` is used as a placeholder for the actual type of the items.
- When we create an `ArrayStack`, we specify the type of items that we intend to store in the stack:

```
ArrayStack<Integer> s1 = new ArrayStack<Integer>(10);
ArrayStack<String> s2 = new ArrayStack<String>(5);
ArrayStack<Object> s3 = new ArrayStack<Object>(20);
```

ArrayStack Constructor

- Java doesn't allow you to create an object or array using a type variable. Thus, we *cannot* do this:

```
public ArrayStack(int maxSize) {
    items = new T[maxSize];   // not allowed
    top = -1;
}
```

- To get around this limitation, we create an array of type `Object` and cast it to be an array of type `T`:

```
public ArrayStack(int maxSize) {
    items = (T[])new Object[maxSize];
    top = -1;
}
```

(This doesn't produce a `ClassCastException` at runtime, because in the compiled version of the class, `T` is replaced with `Object`.)

- The cast generates a compile-time warning, but we'll ignore it.
- Java's built-in `ArrayList` class takes this same approach.

More on Generics

- When a collection class uses the type `Object` for its items, we often need to use casting:

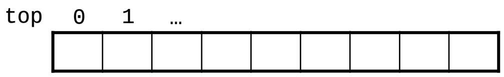
```
LLList list = new LLList();
list.addItem(“hello”);
list.addItem(“world”);
String item = (String)list.getItem(0);
```

- Using generics allows us to avoid this:

```
ArrayStack<String> s = new ArrayStack<String>;
s.push(“hello”);
s.push(“world”);
String item = s.pop();    // no casting needed
```

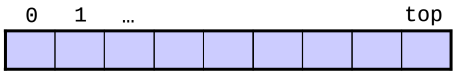
Testing if an `ArrayStack` is Empty or Full

- Empty stack:



```
public boolean isEmpty() {
    return (top == -1);
}
```

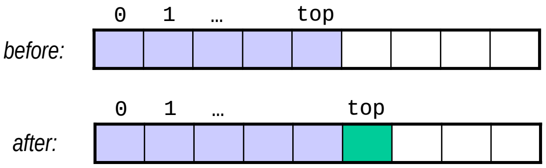
- Full stack:



```
public boolean isFull() {
    return (top == items.length - 1);
}
```

Pushing an Item onto an `ArrayStack`

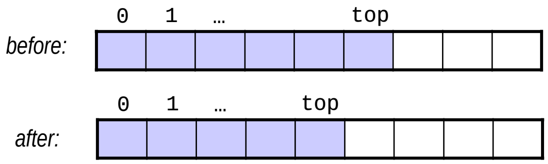
- We increment `top` before adding the item:



```
public boolean push(T item) {
    if (isFull())
        return false;
    top++;
    items[top] = item;
    return true;
}
```

ArrayStack pop() and peek()

- pop: need to get `items[top]` *before* we decrement `top`.



```
public T pop() {
    if (isEmpty())
        return null;
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

- peek just returns `items[top]` without decrementing `top`.

toString() Method for the ArrayStack Class

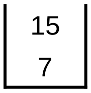
- Assume that we want the method to show us everything in the stack – returning a string of the form

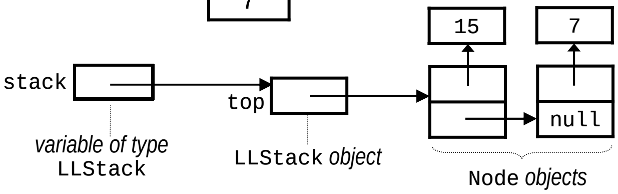
```
    "{top, one-below-top, two-below-top, … bottom}"
```

```
public String toString() {
    String str = "{";

    // what should go here?







    str = str + "}"

    return str;
}
```

Implementing a Generic Stack Using a Linked List

```
public class LLStack<T> implements Stack<T> {
    private Node top;    // top of the stack
    …
}
```

- Example: the stack  would be represented as follows:



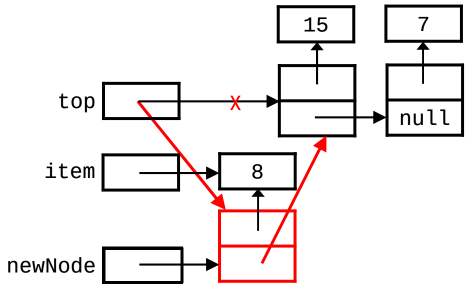
- Things worth noting:
  - our `LLStack` class needs only a single instance variable—a reference to the first node, which holds the top item
  - top item = leftmost item (vs. rightmost item in `ArrayStack`)
  - we don't need a dummy node, because we always insert at the front, and thus the insertion code is already simple

Other Details of Our `LLStack` Class

```
public class LLStack<T> implements Stack<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }
    private Node top;

    public LLStack() {
        top = null;
    }
    public boolean isEmpty() {
        return (top == null);
    }
    public boolean isFull() {
        return false;
    }
}
```

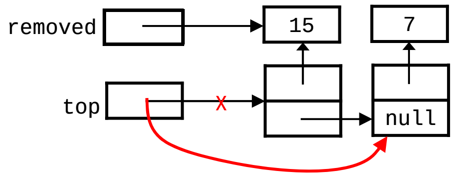
- The inner `Node` class uses the type parameter `T` for the item.
- We don't need to preallocate any memory for the items.
- The stack is never full!

LLStack.push



```
public boolean push(T item) {



}
```

LLStack pop() and peek()



```
public T pop() {
    if (isEmpty())
        return null;

    T removed = __________________;


}

public T peek() {
    if (isEmpty())
        return null;

    return top.item;
}
```

`toString()` Method for the `LLStack` Class

- Again, assume that we want a string of the form

```
"{top, one-below-top, two-below-top, … bottom}"
```

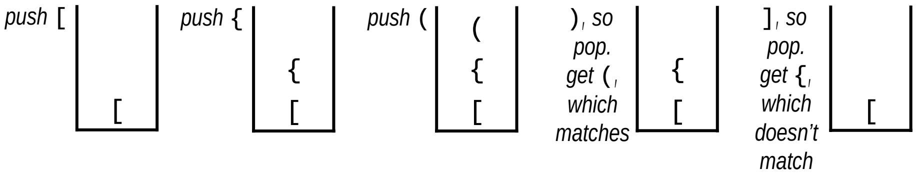
```
public String toString() {
    String str = "{";

    // what should go here?







    str = str + "}"

    return str;
}
```

Efficiency of the Stack Implementations

	`ArrayStack`	`LLStack`
`push()`	*O*(1)	*O*(1)
`pop()`	*O*(1)	*O*(1)
`peek()`	*O*(1)	*O*(1)
space efficiency	*O*(m) where m is the *anticipated* maximum number of items	*O*(n) where n is the number of items currently on the stack

Applications of Stacks

- The runtime stack in memory
- Converting a recursive algorithm to an iterative one by using a stack to emulate the runtime stack
- Making sure that delimiters (parens, brackets, etc.) are balanced:
  - push open (i.e., left) delimiters onto a stack
  - when you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
  - example: `5 * [3 + {(5 + 16 – 2)]`



push [

[

push {

{
[

push (

(
{
[

), so pop. get (, which matches

{
[

], so pop. get {, which doesn't match

[

- Evaluating arithmetic expressions (see textbooks)

An Example of Switching Between Implementations

- In the example code for this unit, there is a test program for each type of sequence:

  `ListTester.java`, `StackTester.java`, `QueueTester.java`

- Each test program uses a variable that has the appropriate *interface* as its type. For example:

```
Stack<String> myStack;
```

- The program asks you which implementation you want to test, and it calls the corresponding constructor:

```
if (type == 1)
    myStack = new ArrayStack<String>(10);
else if (type == 2)
    myStack = new LLStack<String>();
```

- This is an example of what principle of object-oriented programming?

Queue ADT

- A queue is a sequence in which:
  - items are added at the rear and removed from the front
    - first in, first out (FIFO) (vs. a stack, which is last in, first out)
  - you can only access the item that is currently at the front
- Operations:
  - insert: add an item at the rear of the queue
  - remove: remove the item at the front of the queue
  - peek: get the item at the front of the queue, but don't remove it
  - isEmpty: test if the queue is empty
  - isFull: test if the queue is full
- Example: a queue of integers

start: `12 8`

insert 5: `12 8 5`

remove: `8 5`

Our Generic Queue Interface

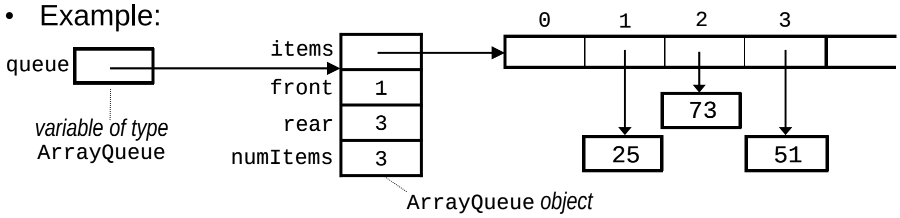
```
public interface Queue<T> {
    boolean insert(T item);
    T remove();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- `insert()` returns `false` if the queue is full, and `true` otherwise.
- `remove()` and `peek()` take no arguments, because we know that we always access the item at the front of the queue.
  - return `null` if the queue is empty.
- Here again, we will use encapsulation to ensure that the data structure is manipulated only in valid ways.

Implementing a Queue Using an Array

```
public class ArrayQueue<T> implements Queue<T> {
    private T[] items;
    private int front;
    private int rear;
    private int numItems;

    ...
}
```

- Example:



- We maintain two indices:
  - `front`: the index of the item at the front of the queue
  - `rear`: the index of the item at the rear of the queue

Avoiding the Need to Shift Items

- Problem: what do we do when we reach the end of the array?

  example: a queue of integers:

front					rear		
54	4	21	17	89	65		

the same queue after removing two items and inserting one:

		front				rear	
		21	17	89	65	43	

to insert two or more additional items, would need to shift items left

- Solution: maintain a *circular queue*. When we reach the end of the array, we wrap around to the beginning.

  the same queue after inserting two additional items:

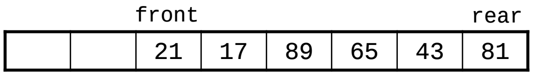
rear		front					
5		21	17	89	65	43	81

A Circular Queue

- To get the front and rear indices to wrap around, we use the modulus operator (%).
- `x % y` = the remainder produced when you divide `x` by `y`
  - examples:
    - `10 % 7 = 3`
    - `36 % 5 = 1`
- Whenever we increment `front` or `rear`, we do so modulo the length of the array.

```
front = (front + 1) % items.length;
rear = (rear + 1) % items.length;
```

- Example:



front rear

		21	17	89	65	43	81

`items.length = 8, rear = 7`

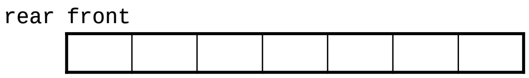
before inserting the next item: `rear = (7 + 1) % 8 = 0`

which wraps `rear` around to the start of the array

Testing if an `ArrayQueue` is Empty

- Initial configuration:

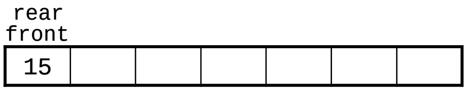
```
rear = -1
front = 0
```



rear front

- We increment `rear` on every insertion, and we increment `front` on every removal.

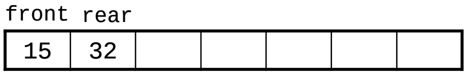
after one insertion:



rear
front

15							

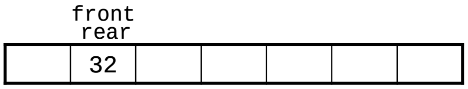
after two insertions:



front rear

15	32						

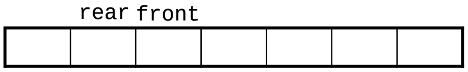
after one removal:



front
rear

	32						

after two removals:



rear front

- The queue is empty when `rear` is one position "behind" `front`:

```
((rear + 1) % items.length) == front
```

Testing if an `ArrayQueue` is Full

- Problem: if we use all of the positions in the array, our test for an empty queue will also hold when the queue is full!

  example: what if we added one more item to this queue?

rear		front					
5		21	17	89	65	43	81

- This is why we maintain `numItems`!

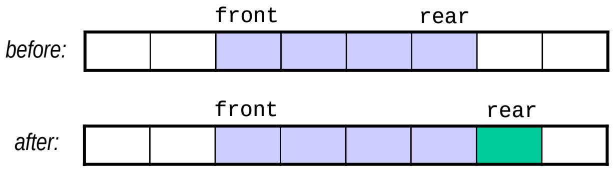
```
public boolean isEmpty() {
    return (numItems == 0);
}

public boolean isFull() {
    return (numItems == items.length);
}
```

Constructor

```
public ArrayQueue(int maxSize) {
    items = (T[])new Object[maxSize];
    front = 0;
    rear = -1;
    numItems = 0;
}
```

Inserting an Item in an `ArrayQueue`

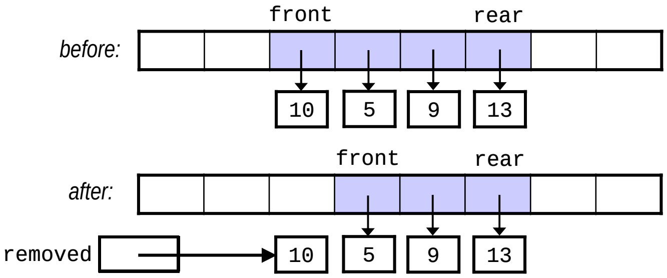
- We increment `rear` before adding the item:



```
public boolean insert(T item) {
    if (isFull())
        return false;
    rear = (rear + 1) % items.length;
    items[rear] = item;
    numItems++;
    return true;
}
```

`ArrayQueue remove()`

- remove: need to get `items[front]` *before* we increment `front`.

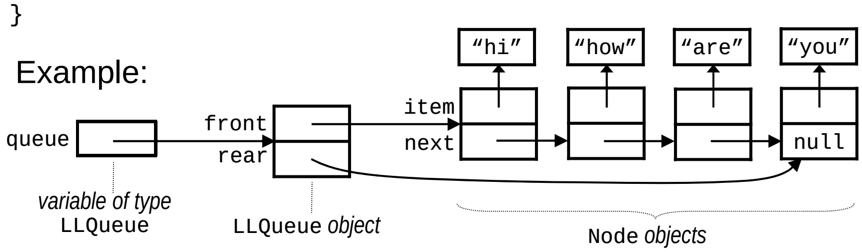


```
public T remove() {
    if (isEmpty())
        return null;
    T removed = items[front];
    items[front] = null;
    front = (front + 1) % items.length;
    numItems--;
    return removed;
}
```

Implementing a Queue Using a Linked List

```
public class LLQueue<T> implements Queue<T> {
    private Node front;    // front of the queue
    private Node rear;     // rear of the queue
    …
}
```

- Example:



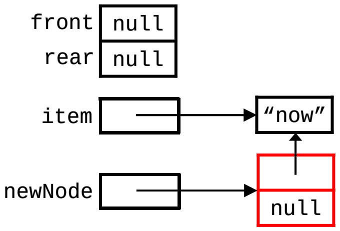
- Because a linked list can be easily modified on both ends, we don’t need to take special measures to avoid shifting items, as we did in our array-based implementation.

Other Details of Our LLQueue Class

```
public class LLQueue<T> implements Queue<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }

    private Node front;
    private Node rear;

    public LLQueue() {
        front = rear = null;
    }
    public boolean isEmpty() {
        return (front == null);
    }
    public boolean isFull() {
        return false;
    }
    …
}
```

- Much simpler than the array-based queue!

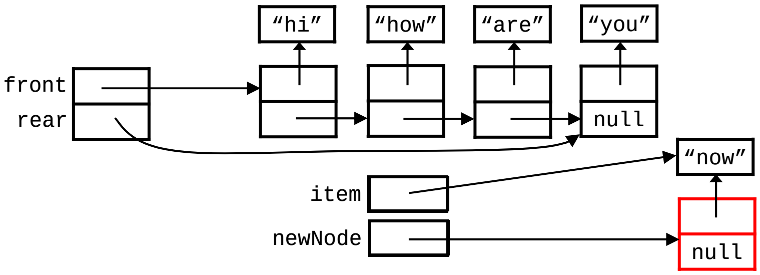
Inserting an Item in an Empty `LLQueue`



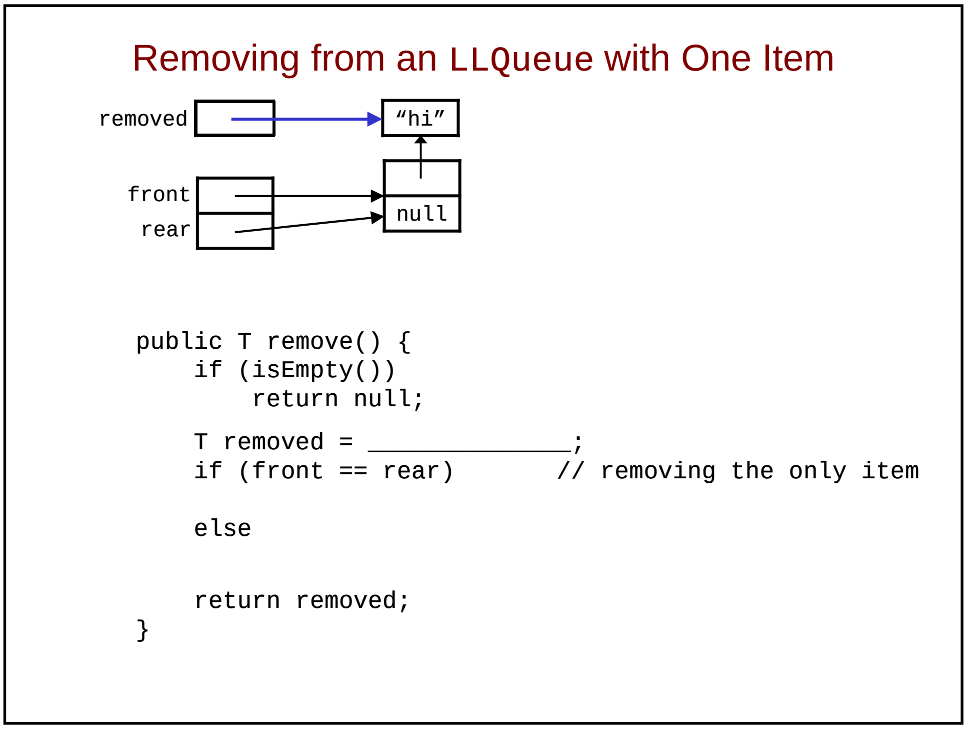
The `next` *field in the* `newNode` *will be* `null` *in either case. Why?*

```
public boolean insert(T item) {
    Node newNode = new Node(item, null);
    if (isEmpty())

    else {


    }
    return true;
}
```

Inserting an Item in a Non-Empty `LLQueue`

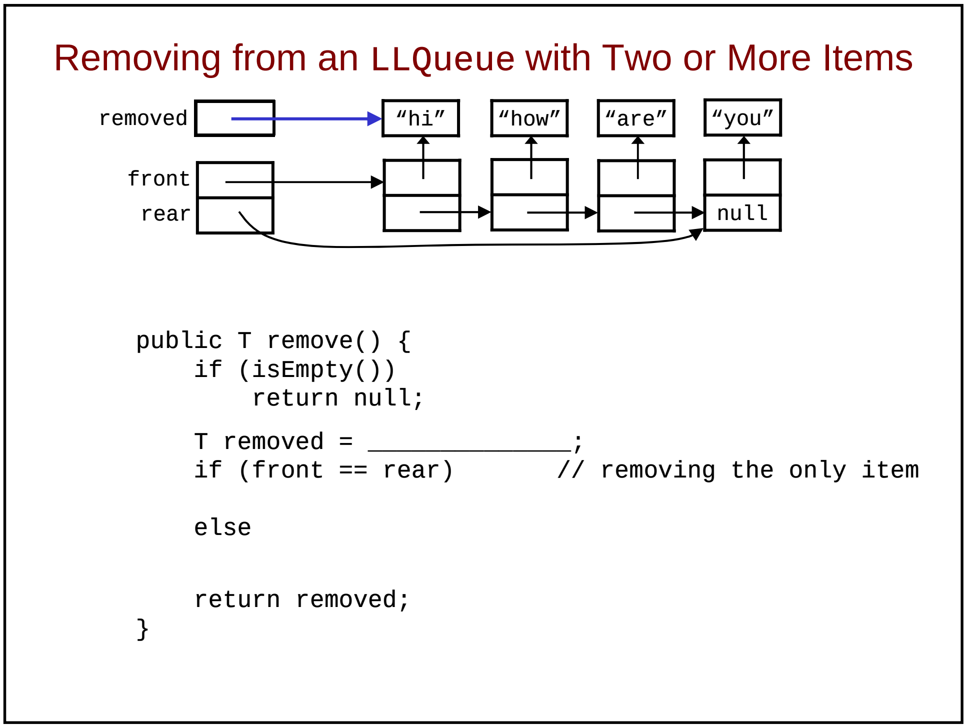


```
public boolean insert(T item) {
    Node newNode = new Node(item, null);
    if (isEmpty())

    else {


    }
    return true;
}
```



Removing from an LLQueue with One Item

```
public T remove() {
    if (isEmpty())
        return null;

    T removed = ______________;
    if (front == rear)       // removing the only item

    else


    return removed;
}
```



Removing from an LLQueue with Two or More Items

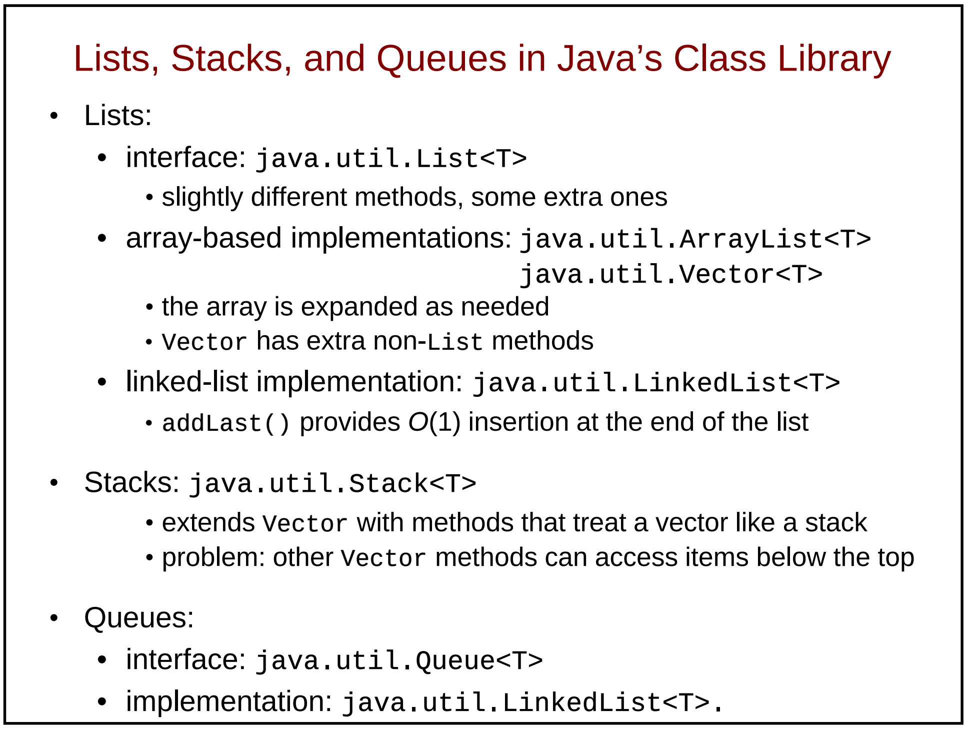
```
public T remove() {
    if (isEmpty())
        return null;

    T removed = ______________;
    if (front == rear)       // removing the only item

    else


    return removed;
}
```

Efficiency of the Queue Implementations

	`ArrayQueue`	`LLQueue`
`insert()`	*O*(1)	*O*(1)
`remove()`	*O*(1)	*O*(1)
`peek()`	*O*(1)	*O*(1)
space efficiency	*O*(m) where m is the *anticipated* maximum number of items	*O*(n) where n is the number of items currently in the queue

Applications of Queues

- first-in first-out (FIFO) inventory control
- OS scheduling: processes, print jobs, packets, etc.
- simulations of banks, supermarkets, airports, etc.
- breadth-first traversal of a graph or level-order traversal of a binary tree (more on these later)



Lists, Stacks, and Queues in Java's Class Library

- Lists:
  - interface: `java.util.List<T>`
    - slightly different methods, some extra ones
  - array-based implementations: `java.util.ArrayList<T>`
    `java.util.Vector<T>`
    - the array is expanded as needed
    - `Vector` has extra non-`List` methods
  - linked-list implementation: `java.util.LinkedList<T>`
    - `addLast()` provides *O*(1) insertion at the end of the list
- Stacks: `java.util.Stack<T>`
    - extends `Vector` with methods that treat a vector like a stack
    - problem: other `Vector` methods can access items below the top
- Queues:
  - interface: `java.util.Queue<T>`
  - implementation: `java.util.LinkedList<T>`.

Unit 9, Part 1

Binary Trees and Huffman Encoding
Binary Search Trees

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

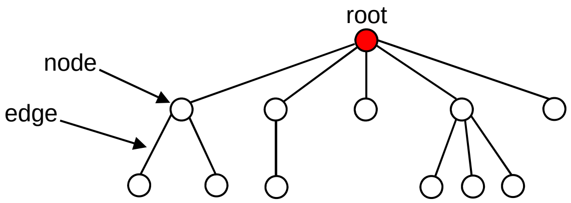
Motivation: Maintaining a Sorted Collection of Data

- A *data dictionary* is a sorted collection of data with the following key operations:
  - *search* for an item (and possibly delete it)
  - *insert* a new item
- If we use a list to implement a data dictionary, efficiency = *O*(n).

data structure	***searching for an item***	***inserting an item***
a list implemented using an array		
a list implemented using a linked list		

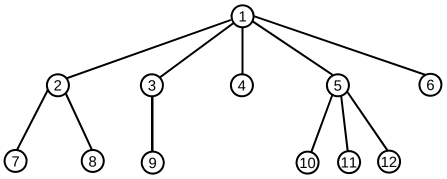
- In the next few lectures, we'll look at data structures (trees and hash tables) that can be used for a more efficient data dictionary.
- We'll also look at other applications of trees.

What Is a Tree?



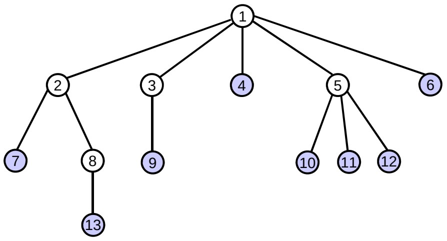
- A tree consists of:
  - a set of *nodes*
  - a set of *edges*, each of which connects a pair of nodes
- Each node may have one or more *data items.*
  - each data item consists of one or more fields
  - *key field* = the field used when searching for a data item
  - multiple data items with the same key are referred to as *duplicates*
- The node at the “top” of the tree is called the *root* of the tree.

Relationships Between Nodes



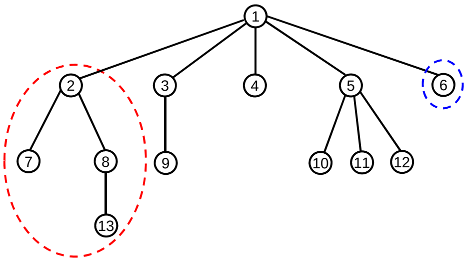
- If a node N is connected to other nodes that are directly below it in the tree, N is referred to as their *parent* and they are referred to as its *children*.
  - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of *at most one* parent.
- Other family-related terms are also used:
  - nodes with the same parent are *siblings*
  - a node’s *ancestors* are its parent, its parent’s parent, etc.
    - example: node 9’s ancestors are 3 and 1
  - a node’s *descendants* are its children, their children, etc.
    - example: node 1’s descendants are *all* of the other nodes

Types of Nodes



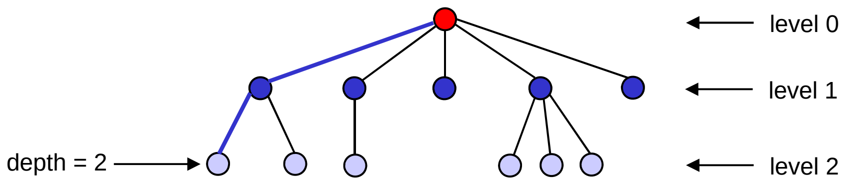
- A *leaf node* is a node without children.
- An *interior node* is a node with one or more children.

A Tree is a Recursive Data Structure



- Each node in the tree is the root of a smaller tree!
  - refer to such trees as *subtrees* to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node
- We'll see that tree algorithms often lend themselves to recursive implementations.

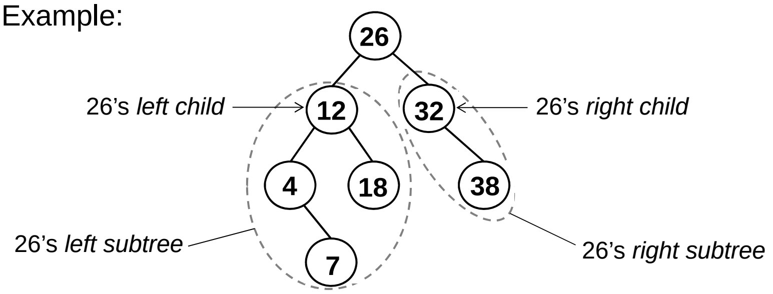
Path, Depth, Level, and Height



- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the maximum depth of its nodes.
    - example: the tree above has a height of 2

Binary Trees

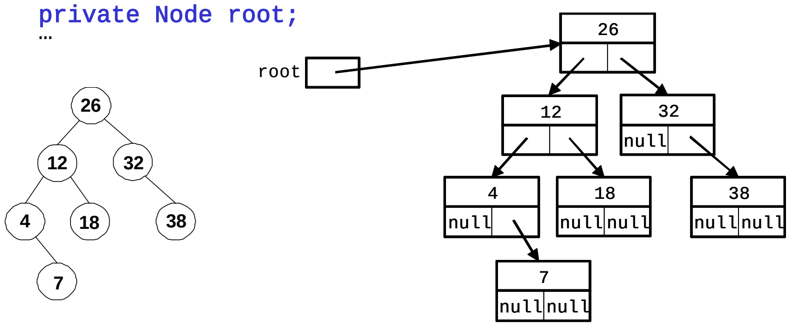
- In a *binary tree*, nodes have *at most two* children.
- Recursive definition: a binary tree is either:
    1) empty, or
    2) a node (the root of the tree) that has
        - one or more data fields
        - a *left child*, which is itself the root of a binary tree
        - a *right child*, which is itself the root of a binary tree
- Example:



- How are the edges of the tree represented?

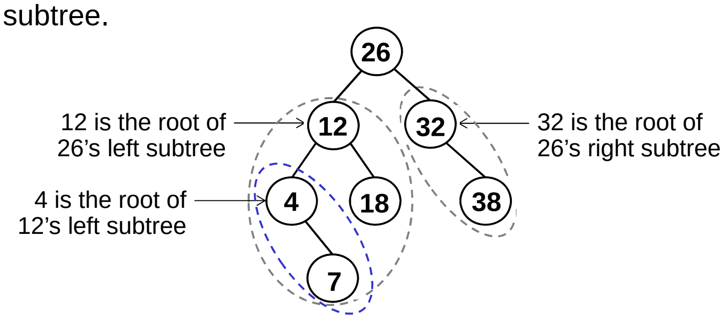
Representing a Binary Tree Using Linked Nodes

```
public class LinkedTree {
    private class Node {
        private int key;
        private LLList data;   // list of data for that key
        private Node left;     // reference to left child
        private Node right;    // reference to right child
        …
    }

    private Node root;
    …
}
```



Traversing a Binary Tree

- Traversing a tree involves *visiting* all of the nodes in the tree.
    - visiting a node = processing its data in some way
        - example: print the key
- We will look at four types of traversals. Each of them visits the nodes in a different order.
- To understand traversals, it helps to remember the recursive definition of a binary tree, in which every node is the root of a subtree.



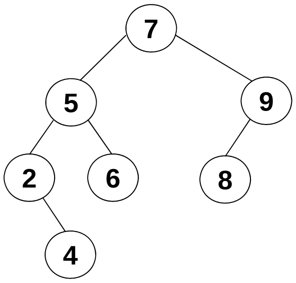
12 is the root of
26's left subtree

32 is the root of
26's right subtree

4 is the root of
12's left subtree

Preorder Traversal

- preorder traversal of the tree whose root is N:
  1) visit the root, N
  2) recursively perform a preorder traversal of N's left subtree
  3) recursively perform a preorder traversal of N's right subtree



- Preorder traversal of the tree above:

  7 5 2 4 6 9 8

Implementing Preorder Traversal

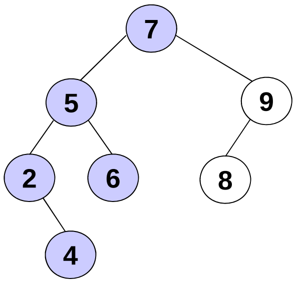
```
public class LinkedTree {
    …
    private Node root;

    public void preorderPrint() {
        if (root != null)
            preorderPrintTree(root);
    }

    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + “  ”);
        if (root.left != null)
            preorderPrintTree(root.left);
        if (root.right != null)
            preorderPrintTree(root.right);
    }
}
```

Not always the same as the root of the entire tree.

- `preorderPrintTree()` is a static, recursive method that takes as a parameter the root of the tree/subtree that you want to print.
- `preorderPrint()` is a non-static method that makes the initial call. It passes in the root of the entire tree as the parameter.

Tracing Preorder Traversal

```
void preorderPrintTree(Node root) {
   System.out.print(root.key + “  ”);
   if (root.left != null)
       preorderPrintTree(root.left);
   if (root.right != null)
       preorderPrintTree(root.right);
}
```

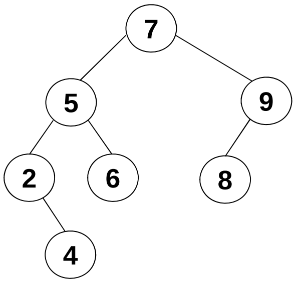


			root: 4 *print 4*				
		root: 2 *print 2*	root: 2	root: 2		root: 6 *print 6*	
	root: 5 *print 5*	root: 5	root: 5	root: 5	root: 5	root: 5	...
root: 7 *print 7*	root: 7	root: 7	root: 7	root: 7	root: 7	root: 7	

time →

Postorder Traversal

- postorder traversal of the tree whose root is N:
  1) recursively perform a postorder traversal of N’s left subtree
  2) recursively perform a postorder traversal of N’s right subtree
  3) visit the root, N



- Postorder traversal of the tree above:

  4 2 6 5 8 9 7

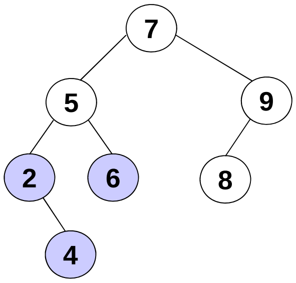
Implementing Postorder Traversal

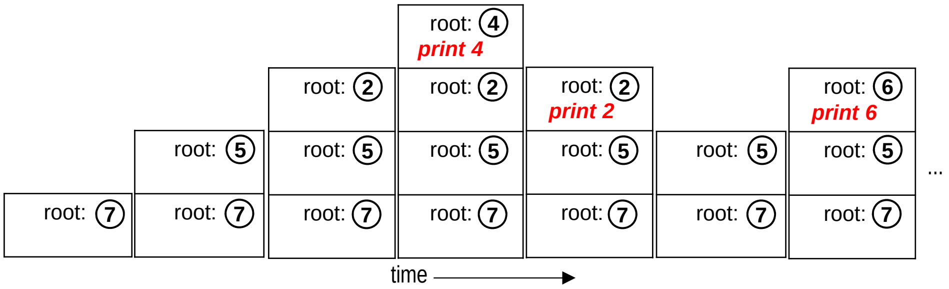
```
public class LinkedTree {
    …
    private Node root;

    public void postorderPrint() {
        if (root != null)
            postorderPrintTree(root);
    }

    private static void postorderPrintTree(Node root) {
        if (root.left != null)
            postorderPrintTree(root.left);
        if (root.right != null)
            postorderPrintTree(root.right);
        System.out.print(root.key + “  ”);
    }
}
```

- Note that the root is printed *after* the two recursive calls.

Tracing Postorder Traversal

```
void postorderPrintTree(Node root) {
   if (root.left != null)
       postorderPrintTree(root.left);
   if (root.right != null)
       postorderPrintTree(root.right);
   System.out.print(root.key + “  ”);
}
```




root: 7

root: 5
root: 7

root: 2
root: 5
root: 7

root: 4
print 4
root: 2
root: 5
root: 7

root: 2
print 2
root: 5
root: 7

root: 5
root: 7

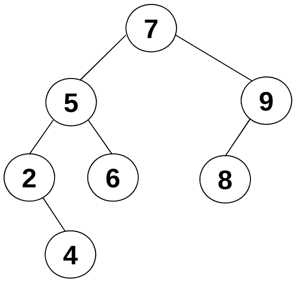
root: 6
print 6
root: 5
root: 7

...

time

Inorder Traversal

- inorder traversal of the tree whose root is N:
  1) recursively perform an inorder traversal of N’s left subtree
  2) visit the root, N
  3) recursively perform an inorder traversal of N’s right subtree



- Inorder traversal of the tree above:

  2 4 5 6 7 8 9

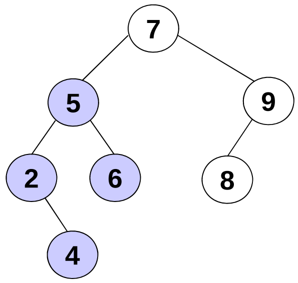
Implementing Inorder Traversal

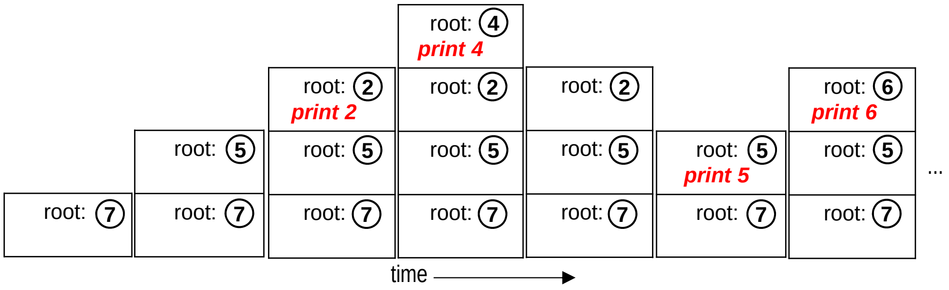
```
public class LinkedTree {
    …
    private Node root;

    public void inorderPrint() {
        if (root != null)
            inorderPrintTree(root);
    }

    private static void inorderPrintTree(Node root) {
        if (root.left != null)
            inorderPrintTree(root.left);
        System.out.print(root.key + “  ”);
        if (root.right != null)
            inorderPrintTree(root.right);
    }
}
```

- Note that the root is printed *between* the two recursive calls.

Tracing Inorder Traversal

```
void inorderPrintTree(Node root) {
   if (root.left != null)
       inorderPrintTree(root.left);
   System.out.print(root.key + “  ”);
   if (root.right != null)
       inorderPrintTree(root.right);
}
```





root: 7

root: 5
root: 7

root: 2
print 2
root: 5
root: 7

root: 4
print 4
root: 2
root: 5
root: 7

root: 2
root: 5
root: 7

root: 5
print 5
root: 7

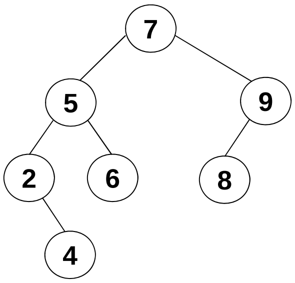
root: 6
print 6
root: 5
root: 7

...

time →

Level-Order Traversal

- Visit the nodes one level at a time, from top to bottom and left to right.



- Level-order traversal of the tree above: **7 5 9 2 6 8 4**
- How could we implement this type of traversal?

Tree-Traversal Summary

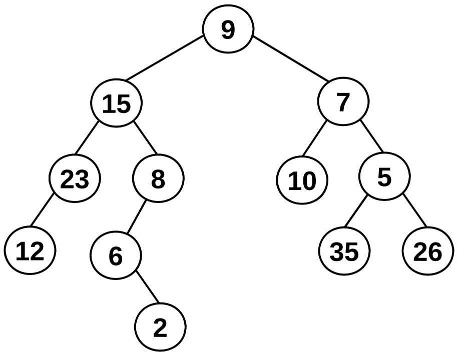
preorder: root, left subtree, right subtree

postorder: left subtree, right subtree, root

inorder: left subtree, root, right subtree

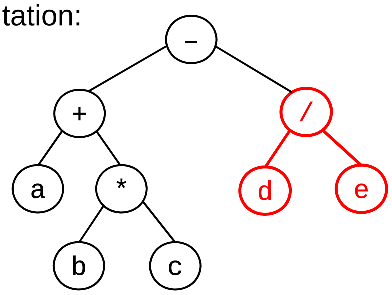
level-order: top to bottom, left to right

- Perform each type of traversal on the tree below:



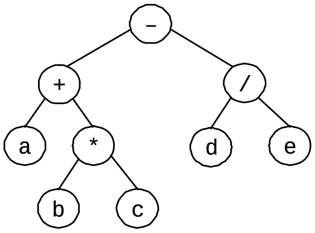
Using a Binary Tree for an Algebraic Expression

- We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: `+`, –, `*`, `/`
- Example expression: `((a + (b * c)) – (d / e))`
- Tree representation:



- Leaf nodes are variables or constants; interior nodes are operators.
- Because the operators are binary, either a node has two children or it has none.

Traversing an Algebraic-Expression Tree



- Inorder gives conventional algebraic notation.
  - print ‘(’ before the recursive call on the left subtree
  - print ‘)’ after the recursive call on the right subtree
  - for tree at right: `((a + (b * c)) – (d / e))`
- Preorder gives functional notation.
  - print ‘(’s and ‘)’s as for inorder, and commas after the recursive call on the left subtree
  - for tree above: `subtr(add(a, mult(b, c)), divide(d, e))`
- Postorder gives the order in which the computation must be carried out on a stack/RPN calculator.
  - for tree above: `push a, push b, push c, multiply, add,…`

Fixed-Length Character Encodings

- A character encoding maps each character to a number.
- Computers usually use fixed-length character encodings.
  - ASCII (American Standard Code for Information Interchange) uses 8 bits per character.

char	dec	binary
a	97	01100001
b	98	01100010
c	99	01100011
…	…	…

example: `“bat”` is stored in a text file as the following sequence of bits:

`01100010 01100001 01110100`

  - Unicode uses 16 bits per character to accommodate foreign-language characters. (ASCII codes are a subset.)
- Fixed-length encodings are simple, because
  - all character encodings have the same length
  - a given character always has the same encoding

Variable-Length Character Encodings

- Problem: fixed-length encodings waste space.
- Solution: use a variable-length encoding.
  - use encodings of different lengths for different characters
  - assign shorter encodings to frequently occurring characters
- Example:

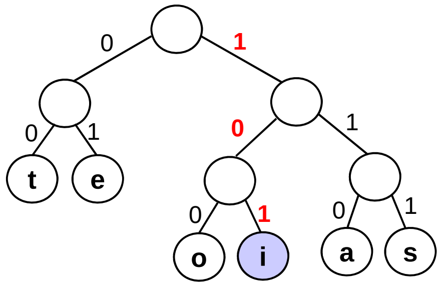
e	`01`
o	`100`
s	`111`
t	`00`

"test" would be encoded as
`00 01 111 00` → `000111100`

- Challenge: when decoding/decompressing an encoded document, how do we determine the boundaries between characters?
  - example: for the above encoding, how do we know whether the next character is 2 bits or 3 bits?
- One requirement: no character's encoding can be the prefix of another character's encoding (e.g., couldn't have `00` and `001`).

Huffman Encoding

- Huffman encoding is a type of variable-length encoding that is based on the actual character frequencies in a given document.
- Huffman encoding uses a binary tree:
  - to determine the encoding of each character
  - to decode an encoded file – i.e., to decompress a compressed file, putting it back into ASCII
- Example of a Huffman tree (for a text with only six chars):



Leaf nodes are characters.

Left branches are labeled with a 0, and right branches are labeled with a 1.

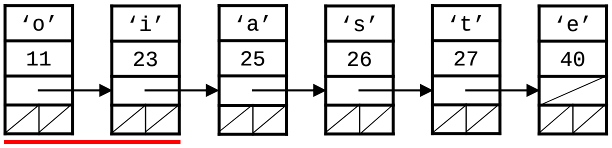
If you follow a path from root to leaf, you get the encoding of the character in the leaf

example: 101 = 'i'

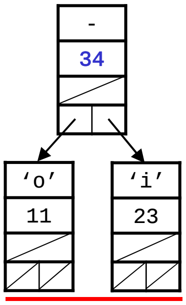
Building a Huffman Tree

1) Begin by reading through the text to determine the frequencies.

2) Create a list of nodes that contain (character, frequency) pairs for each character that appears in the text.

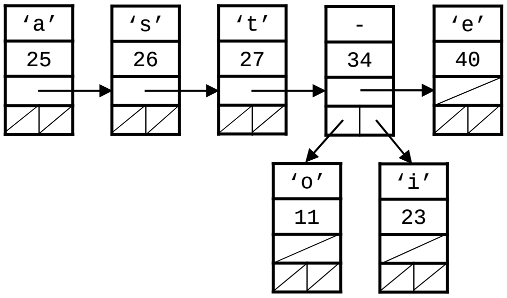


3) Remove and "merge" the nodes with the two lowest frequencies, forming a new node that is their parent.
- left child = lowest frequency node
- right child = the other node
- frequency of parent = sum of the frequencies of its children
  - in this case, 11 + 23 = 34



Building a Huffman Tree (cont.)

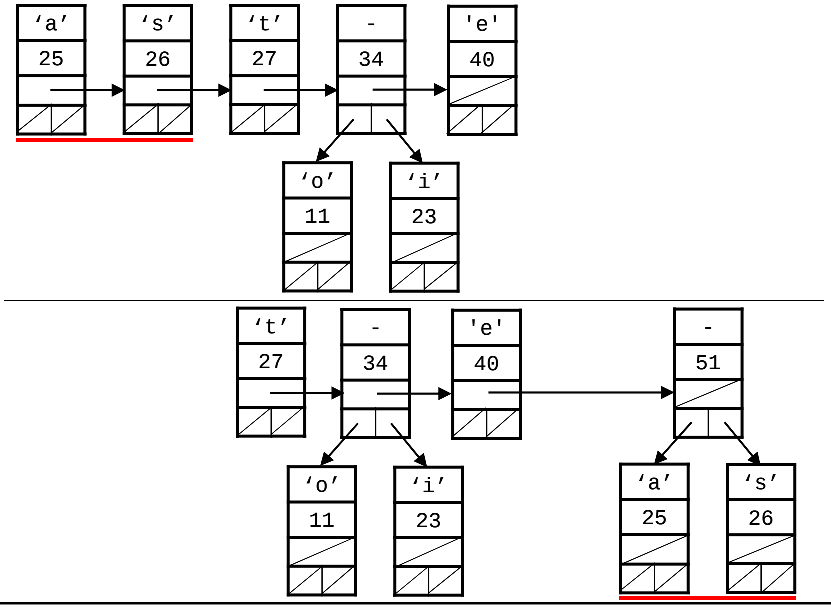
4) Add the parent to the list of nodes (maintaining sorted order):



5) Repeat steps 3 and 4 until there is only a single node in the list, which will be the root of the Huffman tree.

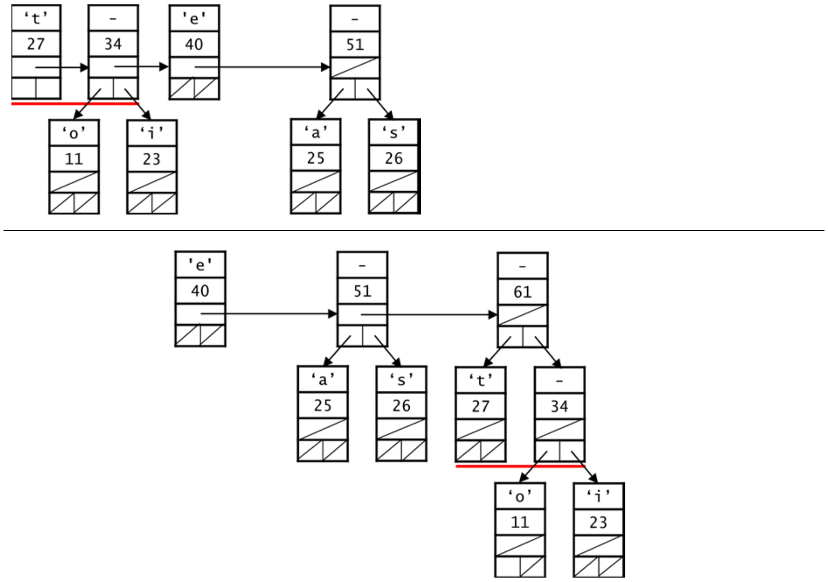
Completing the Huffman Tree Example I

- Merge the two remaining nodes with the lowest frequencies:



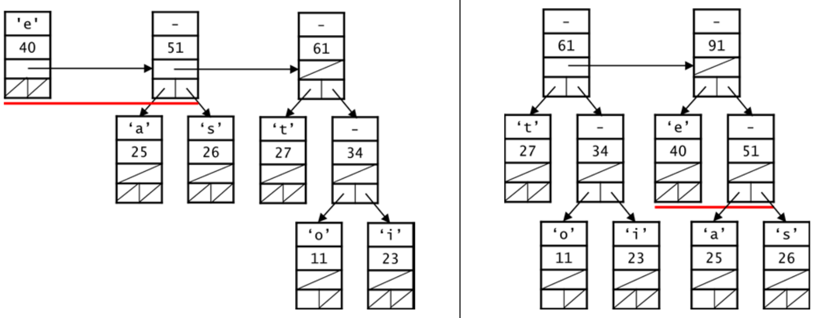
Completing the Huffman Tree Example II

- Merge the next two nodes:



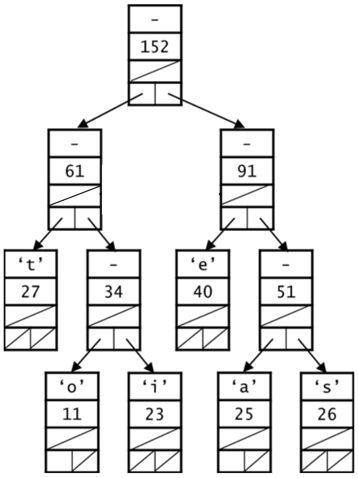
Completing the Huffman Tree Example II

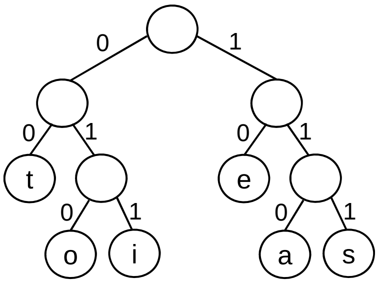
- Merge again:



Completing the Huffman Tree Example IV

- The next merge creates the final tree:

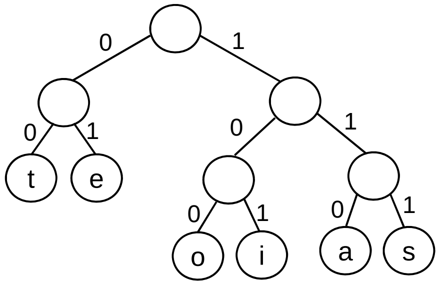




- Characters that appear more frequently end up higher in the tree, and thus their encodings are shorter.

The Shape of the Huffman Tree

- The tree on the last slide is fairly symmetric.
- This won't always be the case!
  - depends on the frequencies of the characters in the document being compressed
- For example, changing the frequency of `'o'` from 11 to 21 would produce the tree shown below:



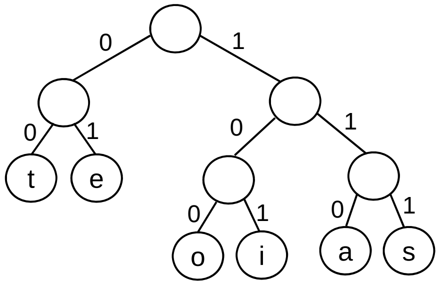
- This is the tree that we'll use in the remaining slides.

Using Huffman Encoding to Compress a File

1) Read through the input file and build its Huffman tree.

2) Write a file header for the output file.
   – include an array containing the frequencies so that the tree can be rebuilt when the file is decompressed.

3) Traverse the Huffman tree to create a table containing the encoding of each character:



a	?
e	?
i	101
o	100
s	111
t	00

4) Read through the input file a second time, and write the Huffman code for each character to the output file.

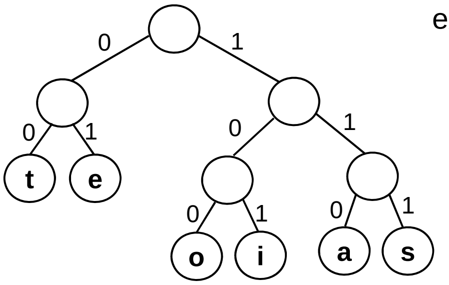
Using Huffman Decoding to Decompress a File

1) Read the frequency table from the header and rebuild the tree.

2) Read one bit at a time and traverse the tree, starting from the root:

when you read a bit of 1, go to the right child
when you read a bit of 0, go to the left child
when you reach a leaf node, record the character,
return to the root, and continue reading bits

The tree allows us to easily overcome the challenge of determining the character boundaries!

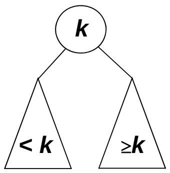


example: 101111110000111100

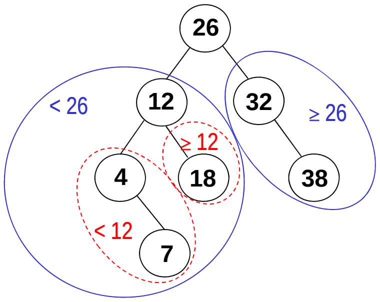
101 =	right,left,right	=	i
111 =	right,right,right	=	s
110 =	right,right,left	=	a
00 =	left,left	=	t
01 =	left,right	=	e
111 =	right,right,right	=	s
00 =	left,left	=	t

Binary *Search* Trees

- Search-tree property: for each node *k:*
  - all nodes in *k*'s left subtree are < *k*
  - all nodes in *k*'s right subtree are >= *k*



- Our earlier binary-tree example is a search tree:

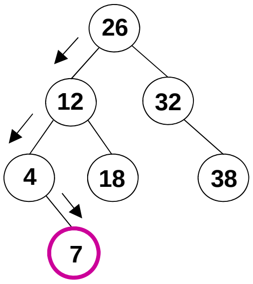


Searching for an Item in a Binary Search Tree

- Algorithm for searching for an item with a key *k*:

  if *k* == the root node's key, you're done
  else if *k* < the root node's key, search the left subtree
  else search the right subtree

- Example: search for 7



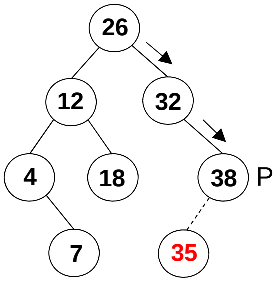
Implementing Binary-Tree Search

```
public class LinkedTree {   // Nodes have keys that are ints
    …
    private Node root;

    public LLList search(int key) {
        Node n = searchTree(root, key);
        if (n == null)
            return null;     // no such key
        else
            return n.data;   // return list of values for key
    }

    private static Node searchTree(Node root, int key) {
        // write together








    }
}
```

Inserting an Item in a Binary Search Tree

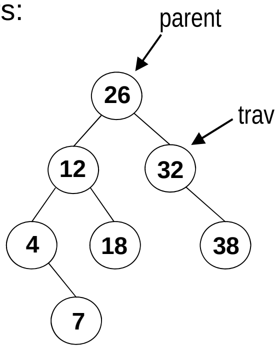
- We want to insert an item whose key is *k*.
- We traverse the tree as if we were searching for *k.*
- If we find a node with key *k*, we add the data item to the list of items for that node.
- If we don't find it, the last node we encounter will be the parent P of the new node.
  - if *k* < P's key, make the new node P's left child
  - else make the node P's right child
- *Special case:* if the tree is empty, make the new node the root of the tree.
- The resulting tree is still a search tree.

example: insert 35



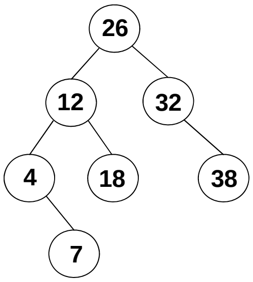
Implementing Binary-Tree Insertion

- We'll implement part of the `insert()` method together.
- We'll use iteration rather than recursion.
- Our method will use two references/pointers:
  - `trav`: performs the traversal down to the point of insertion
  - `parent`: stays one behind `trav`
    - like the `trail` reference that we sometimes use when traversing a linked list



Implementing Binary-Tree Insertion

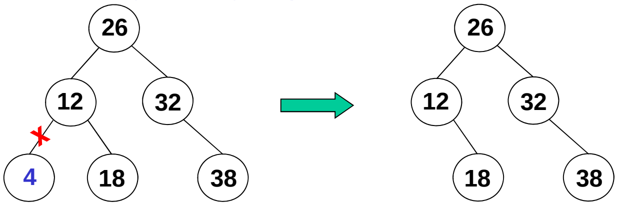
```
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }



    }
    Node newNode = new Node(key, data);
    if (root == null)    // the tree was empty
        root = newNode;
    else if (key < parent.key)
        parent.left = newNode;
    else
        parent.right = newNode;
}
```



Deleting Items from a Binary Search Tree

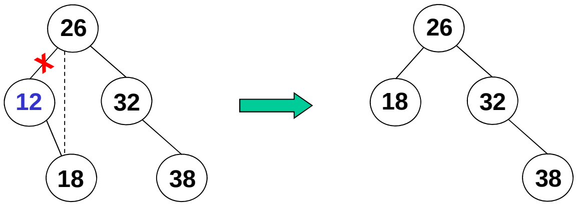
- Three cases for deleting a node *x*
- **Case 1:** *x* has no children.
  Remove *x* from the tree by setting its parent's reference to null.

ex: delete 4



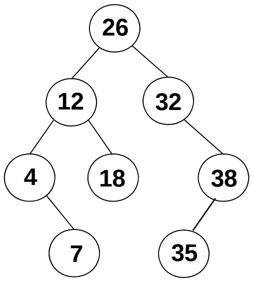
- **Case 2:** *x* has one child.
  Take the parent's reference to *x* and make it refer to *x*'s child.

ex: delete 12



Deleting Items from a Binary Search Tree (cont.)

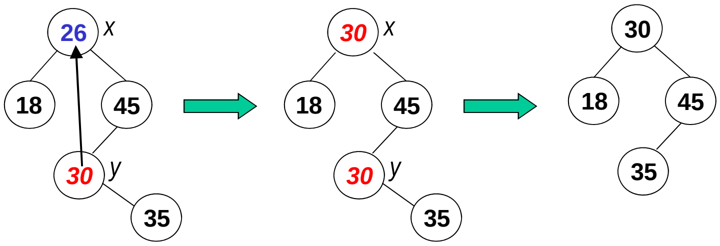
- **Case 3:** *x* has two children
  - we can't just delete *x*. why?
  - instead, we replace *x* with a node from elsewhere in the tree
  - to maintain the search-tree property, we must choose the replacement carefully
    - example: what nodes could replace 26 below?



Deleting Items from a Binary Search Tree (cont.)

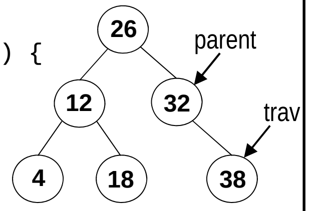
- **Case 3:** *x* has two children (continued):
  - replace *x* with the smallest node in *x*'s right subtree—call it *y*
  - *y* will either be a leaf node or will have one right child. why?
- After copying *y*'s item into *x*, we delete *y* using case 1 or 2.

ex:
delete 26



Implementing Binary-Tree Deletion

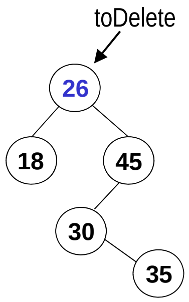
```
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null)    // no such key
        return null;
    else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```



- This method uses a helper method to delete the node.

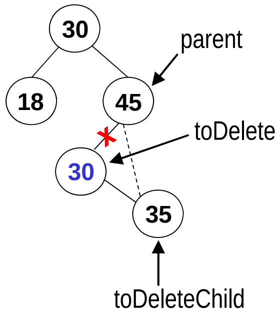
Implementing Case 3

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement – and
        // the replacement's parent.
        Node replaceParent = toDelete;

        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?



        // Replace toDelete's key and data
        // with those of the replacement item.
        toDelete.key = replace.key;
        toDelete.data = replace.data;

        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
}
```



Implementing Cases 1 and 2

```
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        else
            toDeleteChild = toDelete.right;
        // Note: in case 1, toDeleteChild
        // will have a value of null.

        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    }
}
```

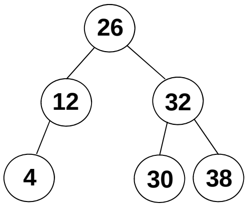


Efficiency of a Binary Search Tree

- The three key operations (search, insert, and delete) all have the same time complexity.
  - insert and delete both involve a search followed by a constant number of additional operations
- Time complexity of searching a binary search tree:
  - best case: *O*(1)
  - worst case: *O*(*h*), where *h* is the height of the tree
  - average case: *O*(*h*)
- What is the height of a tree containing n items?
  - it depends! why?

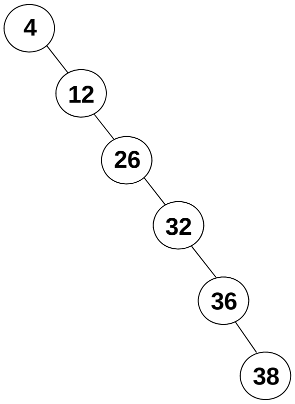
Balanced Trees

- A tree is *balanced* if, for each node, the node's subtrees have the same height or have heights that differ by 1.
- For a balanced tree with n nodes:
  - height = $O(\log_2 n)$.
  - gives a worst-case time complexity that is logarithmic ($O(\log_2 n)$)
    - the best worst-case time complexity for a binary tree



What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
  - height = n - 1
  - worst-case time complexity = $O(n)$
- We'll look next at search-tree variants that take special measures to ensure balance.



Unit 9, Part 2

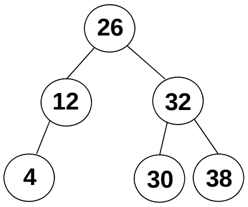
Balanced Search Trees

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Balanced Trees

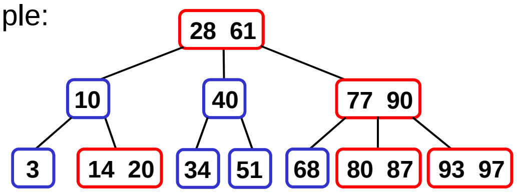
- A tree is *balanced* if, for each node, the node's subtrees have the same height or have heights that differ by 1.
- For a balanced tree with n nodes:
  - height = $O(\log_2 n)$.
  - gives a worst-case time complexity that is logarithmic ($O(\log_2 n)$)
    - the best worst-case time complexity for a binary search tree

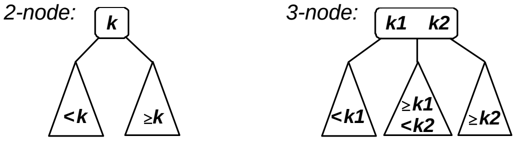


- With a binary search tree, there's no way to ensure that the tree remains balanced.
  - can degenerate to $O(n)$ time

2-3 Trees

- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and 0 or 2 children
    - a **3-node**, which contains two data items and 0 or 3 children
  - the keys form a search tree
- Example:

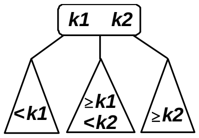




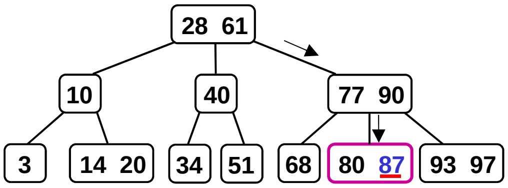
Search in 2-3 Trees

- Algorithm for searching for an item with a key *k*:

```
if k == one of the root node's keys, you're done
else if k < the root node's first key
    search the left subtree
else if the root is a 3-node and k < its second key
    search the middle subtree
else
    search the right subtree
```



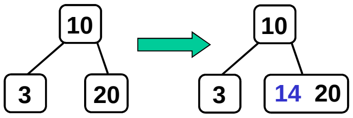
- Example: search for 87



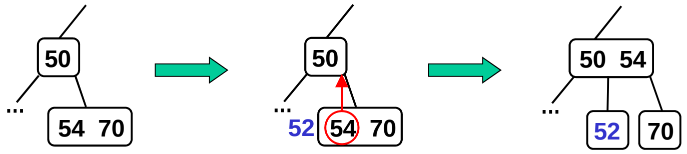
Insertion in 2-3 Trees

- Algorithm for inserting an item with a key *k*:

```
search for k, but don't stop until you hit a leaf node
let L be the leaf node at the end of the search
if L is a 2-node
     add k to L, making it a 3-node
else if L is a 3-node
     split L into two 2-nodes containing the items with the
       smallest and largest of: k, L's 1st key, L's 2nd key
     the middle item is "sent up" and inserted in L's parent
```

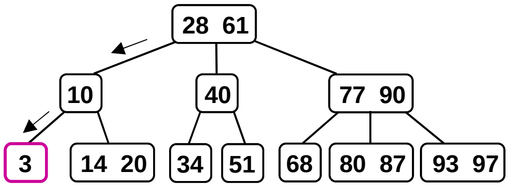


example: add 52

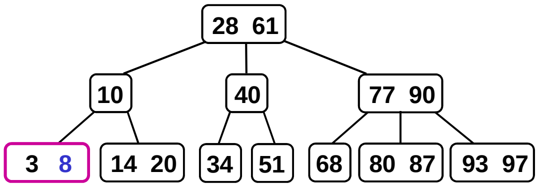


Example 1: Insert 8

- Search for 8:

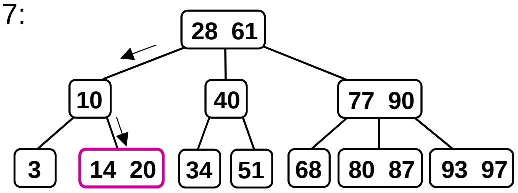


- Add 8 to the leaf node, making it a 3-node:

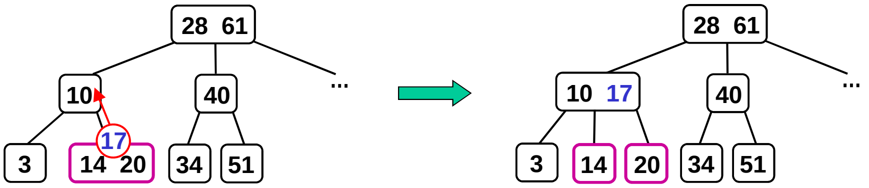


Example 2: Insert 17

- Search for 17:

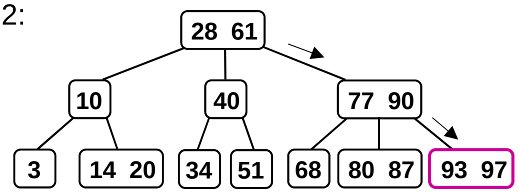


- Split the leaf node, and send up the middle of 14, 17, 20 and insert it the leaf node's parent:

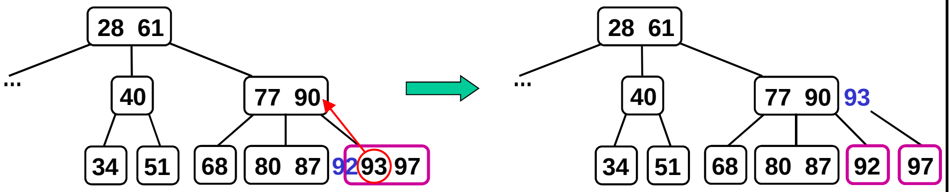


Example 3: Insert 92

- Search for 92:



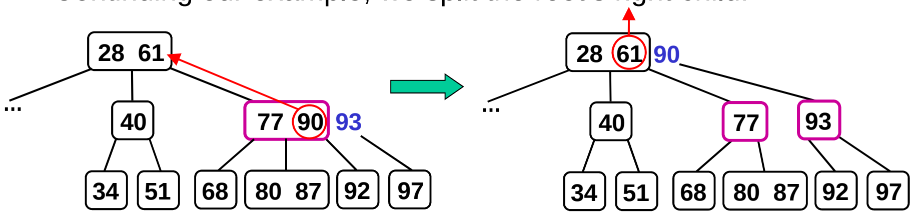
- Split the leaf node, and send up the middle of 92, 93, 97 and insert it the leaf node's parent:



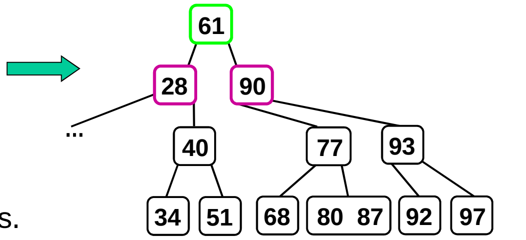
- In this case, the leaf node's parent is also a 3-node, so we need to split is as well…

Splitting the Root Node

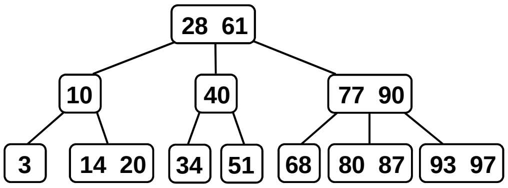
- If an item propagates up to the root node, and the root is a 3-node, we split the root node and create a new, 2-node root containing the middle of the three items.
- Continuing our example, we split the root's right child:



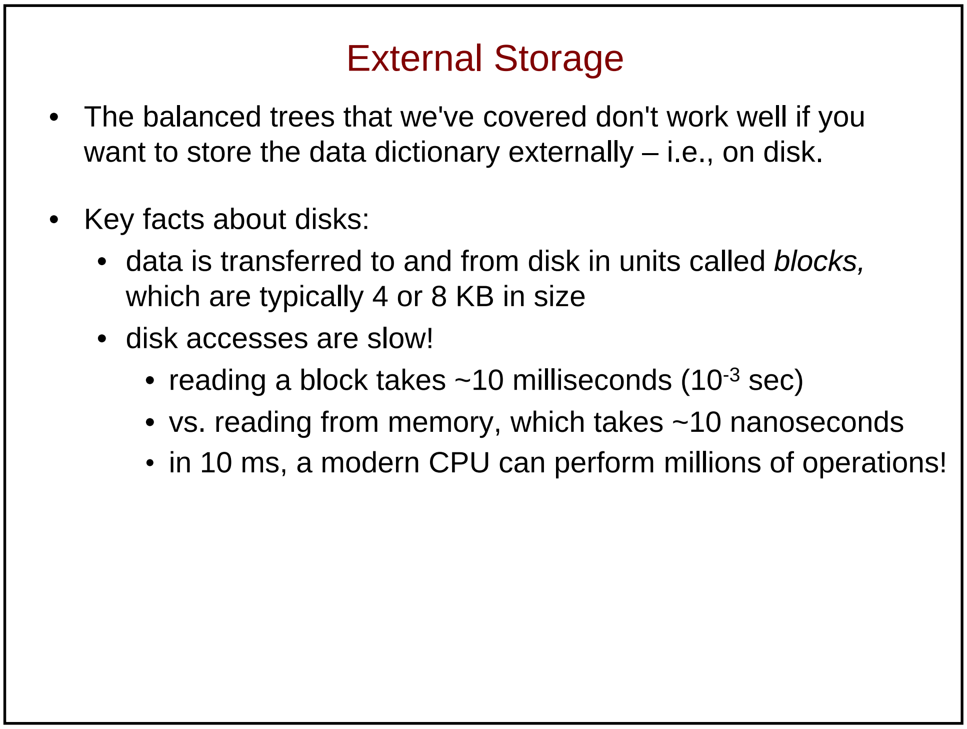
- Then we split the root, which increases the tree's height by 1, but the tree is still balanced.
- This is only case in which the tree's height increases.



Efficiency of 2-3 Trees



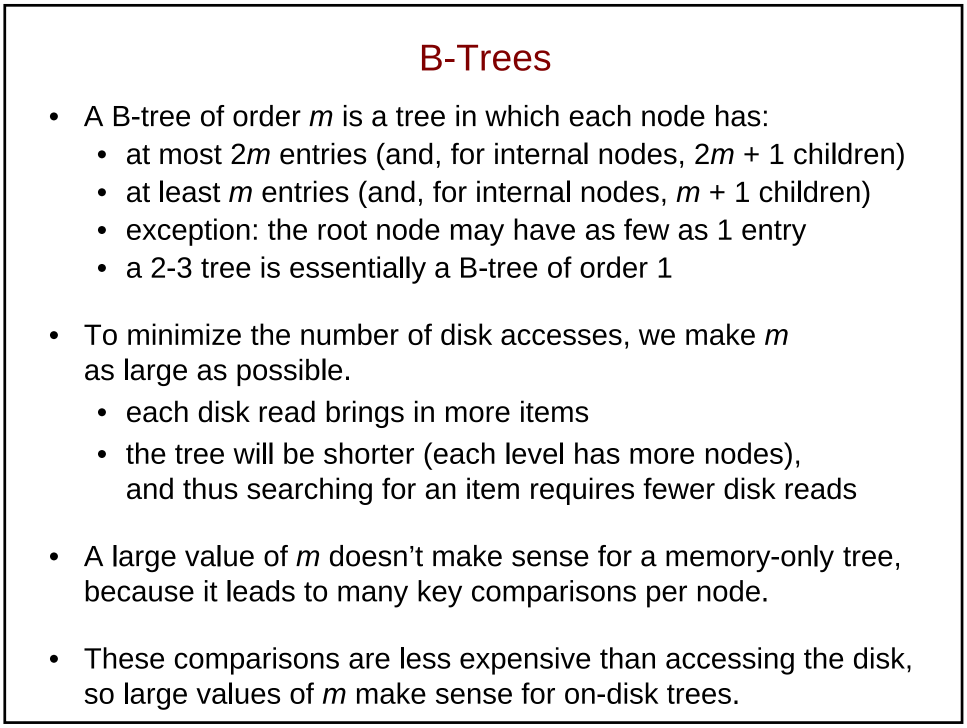
- A 2-3 tree containing n items has a height <= $\log_2 n$.
- Thus, search and insertion are both $O(\log n)$.
  - a search visits at most $\log_2 n$ nodes
  - an insertion begins with a search; in the worst case, it goes all the way back up to the root performing splits, so it visits at most $2\log_2 n$ nodes
- Deletion is tricky – you may need to coalesce nodes! However, it also has a time complexity of $O(\log n)$.
- Thus, we can use 2-3 trees for a $O(\log n)$-time data dictionary.



External Storage

- The balanced trees that we've covered don't work well if you want to store the data dictionary externally – i.e., on disk.

- Key facts about disks:
  - data is transferred to and from disk in units called *blocks,* which are typically 4 or 8 KB in size
  - disk accesses are slow!
    - reading a block takes ~10 milliseconds (10^{-3} sec)
    - vs. reading from memory, which takes ~10 nanoseconds
    - in 10 ms, a modern CPU can perform millions of operations!



B-Trees

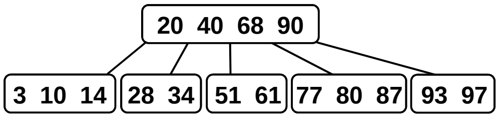
- A B-tree of order *m* is a tree in which each node has:
  - at most 2*m* entries (and, for internal nodes, 2*m* + 1 children)
  - at least *m* entries (and, for internal nodes, *m* + 1 children)
  - exception: the root node may have as few as 1 entry
  - a 2-3 tree is essentially a B-tree of order 1

- To minimize the number of disk accesses, we make *m* as large as possible.
  - each disk read brings in more items
  - the tree will be shorter (each level has more nodes), and thus searching for an item requires fewer disk reads

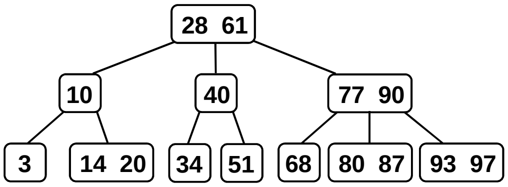
- A large value of *m* doesn't make sense for a memory-only tree, because it leads to many key comparisons per node.

- These comparisons are less expensive than accessing the disk, so large values of *m* make sense for on-disk trees.

Example: a B-Tree of Order 2



- Order 2: at most 4 data items per node (and at most 5 children)
- The above tree holds the same keys as one of our earlier 2-3 trees, which is shown again below:

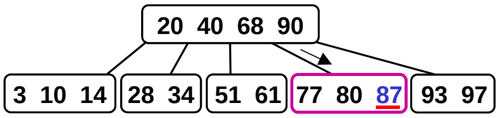


- We used the same order of insertion to create both trees:

  51, 3, 40, 77, 20, 10, 34, 28, 61, 80, 68, 93, 90, 97, 87, 14
- For extra practice, see if you can reproduce the trees!

Search in B-Trees

- Similar to search in a 2-3 tree.
- Example: search for 87



Insertion in B-Trees

- Similar to insertion in a 2-3 tree:

  search for the key until you reach a leaf node

  if a leaf node has fewer than 2*m* items, add the item to the leaf node

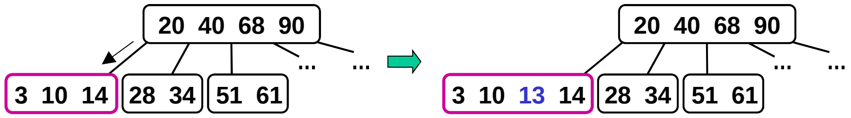
  else split the node, dividing up the 2*m* + 1 items:

    the smallest *m* items remain in the original node

    the largest *m* items go in a new node

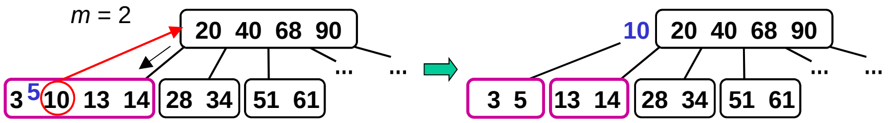
    send the middle entry up and insert it (and a pointer to the new node) in the parent

- Example of an insertion without a split: insert 13

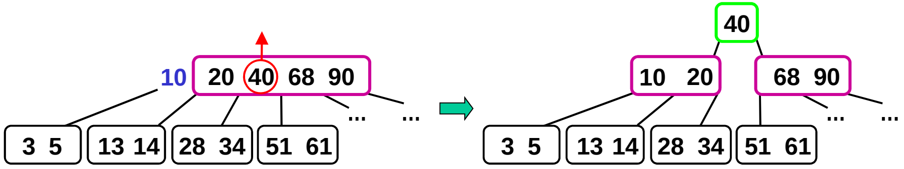


Splits in B-Trees

- Insert 5 into the result of the previous insertion:



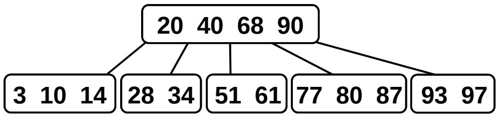
- The middle item (the 10) was sent up to the root. It has no room, so it is split as well, and a new root is formed:



- Splitting the root increases the tree’s height by 1, but the tree is still balanced. This is only way that the tree’s height increases.

- When an internal node is split, its 2*m* + 2 pointers are split evenly between the original node and the new node.

Analysis of B-Trees



20 40 68 90

3 10 14

28 34

51 61

77 80 87

93 97

- All internal nodes have at least *m* children (actually, at least *m*+1).
- Thus, a B-tree with n items has a height <= $\log_m n$, and search and insertion are both $O(\log_m n)$.
- As with 2-3 trees, deletion is tricky, but it's still logarithmic.

Search Trees: Conclusions

- Binary search trees can be $O(\log n)$, but they can degenerate to $O(n)$ running time if they are out of balance.
- 2-3 trees and B-trees are *balanced* search trees that guarantee $O(\log n)$ performance.
- When data is stored on disk, the most important performance consideration is reducing the number of disk accesses.
- B-trees offer improved performance for on-disk data dictionaries.

Unit 9, Part 3

Heaps and Priority Queues

Computer Science S-111
Harvard University

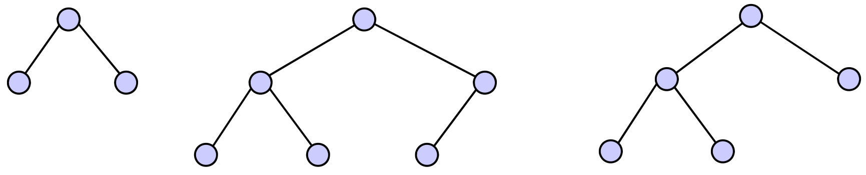
David G. Sullivan, Ph.D.

Priority Queue

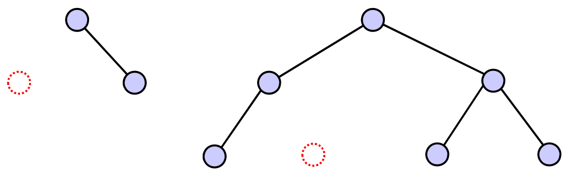
- A *priority queue* is a collection in which each item in the collection has an associated number known as a *priority*.
  - ("Henry Leitner", 10), ("Drew Faust", 15), ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"
- Example: scheduling a shared resource like the CPU
  - give some processes/applications a higher priority, so that they will be scheduled first and/or more often
- Key operations:
  - *insert:* add an item to the priority queue, positioning it according to its priority
  - *remove:* remove the item with the highest priority
- How can we efficiently implement a priority queue?
  - use a type of binary tree known as a *heap*

Complete Binary Trees

- A binary tree of height *h* is *complete* if:
  - levels 0 through *h* – 1 are fully occupied
  - there are no “gaps” to the left of a node in level *h*
- Complete:

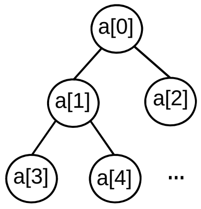


- Not complete (◌ = missing node):

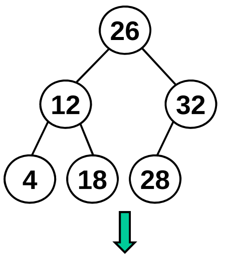


Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.
- The nodes of the tree are stored in the array in the order in which they would be visited by a level-order traversal (i.e., top to bottom, left to right).

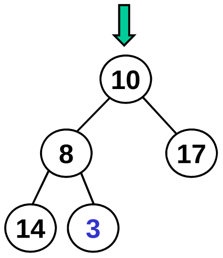


- Examples:



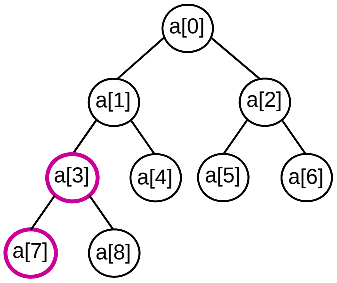
26	12	32	4	18	28

10	8	17	14	3



Navigating a Complete Binary Tree in Array Form

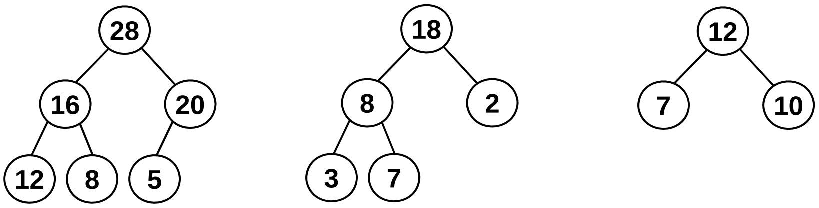
- The root node is in `a[0]`
- Given the node in `a[i]`:
  - its left child is in `a[2*i + 1]`
  - its right child is in `a[2*i + 2]`
  - its parent is in `a[(i – 1)/2]` (using integer division)



- Examples:
  - the left child of the node in `a[1]` is in `a[2*1 + 1]` = `a[3]`
  - the right child of the node in `a[3]` is in `a[2*3 + 2]` = `a[8]`
  - the parent of the node in `a[4]` is in `a[(4-1)/2]` = `a[1]`
  - the parent of the node in `a[7]` is in `a[(7-1)/2]` = `a[3]`

Heaps

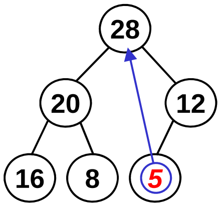
- Heap: a complete binary tree in which each interior node is greater than or equal to its children
- Examples:



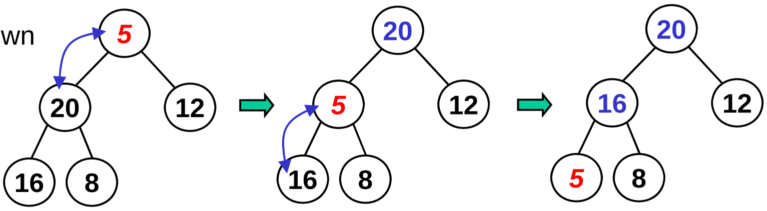
- The largest value is always at the root of the tree.
- The smallest value can be in *any* leaf node – there's no guarantee about which one it will be.
- Strictly speaking, the heaps that we will use are *max-at-top* heaps. You can also define a *min-at-top* heap, in which every interior node is less than or equal to its children.

Removing the Largest Item from a Heap

- Remove and return the item in the root node.
- In addition, we need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children
- Algorithm:
  1. make a copy of the largest item
  2. move the last item in the heap to the root
  3. “sift down” the new root item until it is >= its children (or it’s a leaf)
  4. return the largest item



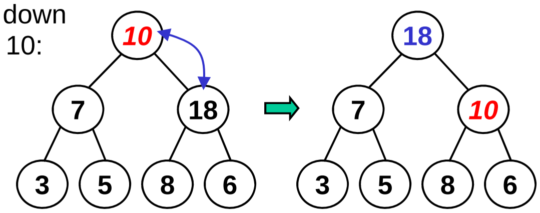
sift down the 5:



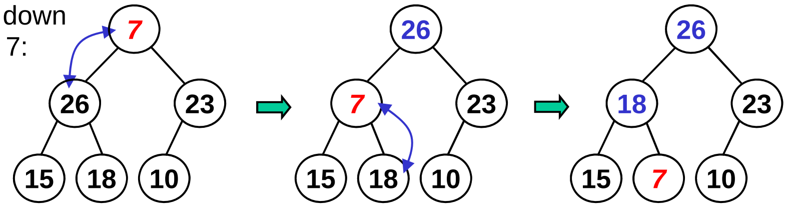
Sifting Down an Item

- To sift down item *x* (i.e., the item whose key is *x*):
  1. compare *x* with the larger of the item’s children, *y*
  2. if *x* < *y*, swap *x* and *y* and repeat
- Other examples:

sift down the 10:



sift down the 7:

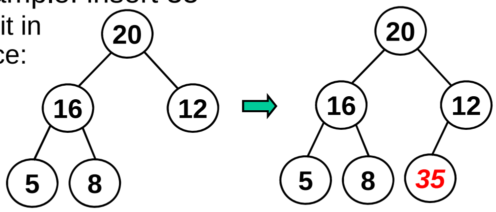


Inserting an Item in a Heap

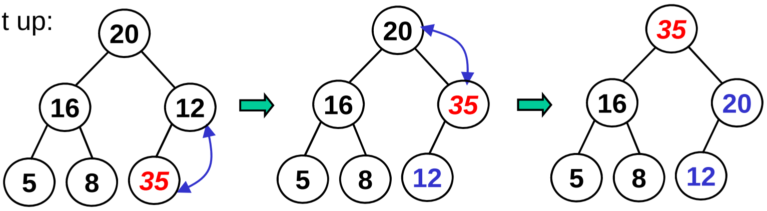
- Algorithm:
  1. put the item in the next available slot (grow array if needed)
  2. “sift up” the new item
     until it is <= its parent (or it becomes the root item)

- Example: insert 35

put it in place:



sift it up:

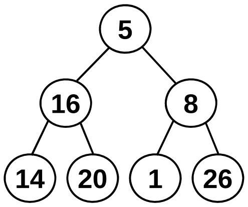


Converting an Arbitrary Array to a Heap

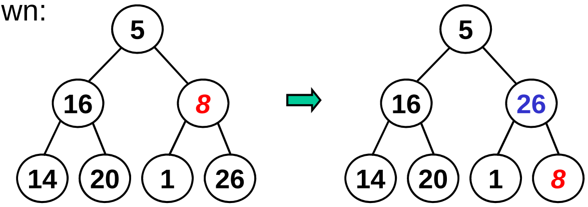
- To convert an array (call it `contents`) with n items to a heap:
  1. start with the parent of the last element:
     `contents[i]`, where `i = ((n – 1) – 1)/2 = (n – 2)/2`
  2. sift down `contents[i]` and all elements to its left

- Example:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

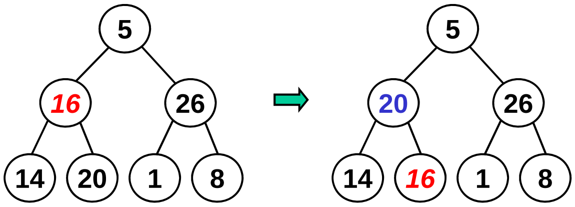


- Last element’s parent = `contents[(7 – 2)/2]` = `contents[2]`.
  Sift it down:

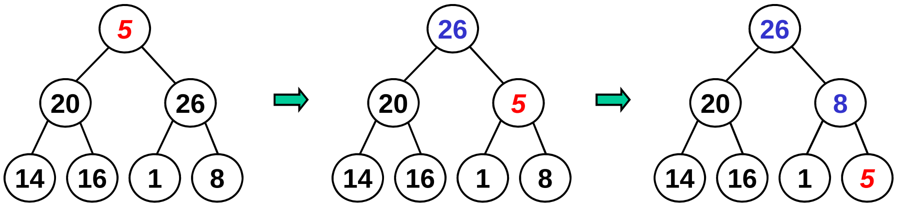


Converting an Array to a Heap (cont.)

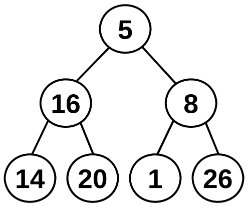
- Next, sift down `contents[1]`:



- Finally, sift down `contents[0]`:



Time Complexity of a Heap



- A heap containing n items has a height <= $\log_2 n$.
- Thus, removal and insertion are both *O*(log n).
  - remove: go down at most $\log_2 n$ levels when sifting down from the root, and do a constant number of operations per level
  - insert: go up at most $\log_2 n$ levels when sifting up to the root, and do a constant number of operations per level
- This means we can use a heap for a *O*(log n)-time priority queue.
- Time complexity of creating a heap from an array?

Using a Heap to Sort an Array

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

0	1	2	3	4	5	6
5	**16**	**8**	**14**	**20**	**1**	**26**

0	1	2	3	4	5	6
1	**16**	**8**	**14**	**20**	**5**	**26**

0	1	2	3	4	5	6
1	***5***	**8**	**14**	**20**	**16**	**26**

…

- It isn't efficient ($O(n^2)$), because it performs a linear scan to find the smallest remaining element ($O(n)$ steps per scan).
- Heapsort is a sorting algorithm that repeatedly finds the *largest* remaining element and puts it in place.
- It *is* efficient ($O(n \log n)$), because it turns the array into a heap, which means that it can find and remove the largest remaining element in $O(\log n)$ steps.

Heapsort

- Pseudocode:

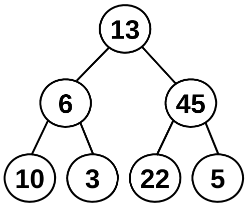
```
heapSort(arr) {
    // Turn the array into a max-at-top heap.
    heap = new Heap(arr);

    endUnsorted = arr.length - 1;
    while (endUnsorted > 0) {
        // Get the largest remaining element and put it
        // at the end of the unsorted portion of the array.
        largestRemaining = heap.remove();
        arr[endUnsorted] = largestRemaining;

        endUnsorted--;
    }
}
```

Heapsort Example

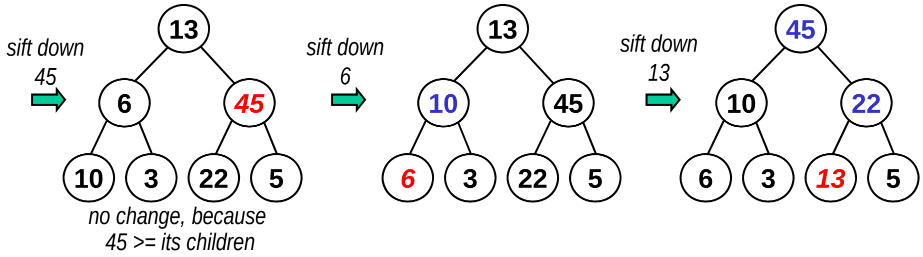
- Sort the following array:

0	1	2	3	4	5	6
13	6	45	10	3	22	5

- Here's the corresponding complete tree:

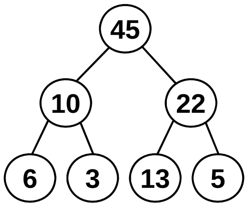


- Begin by converting it to a heap:



Heapsort Example (cont.)

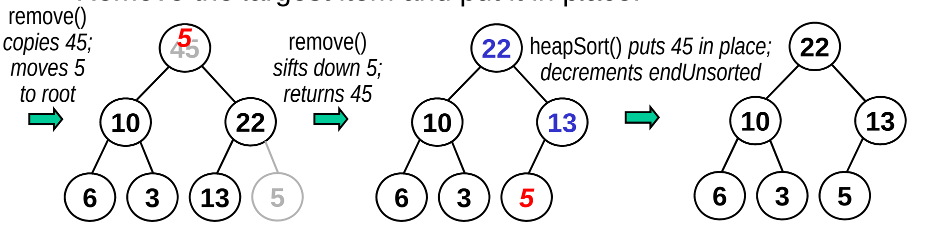
- Here's the heap in both tree and array forms:



0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: **6**

- Remove the largest item and put it in place:


remove() copies 45; moves 5 to root
remove() sifts down 5; returns 45
heapSort() puts 45 in place; decrements endUnsorted
5
45
10
22
6
3
13
5
22
10
13
6
3
5
22
10
13
6
3
5

toRemove: **45**

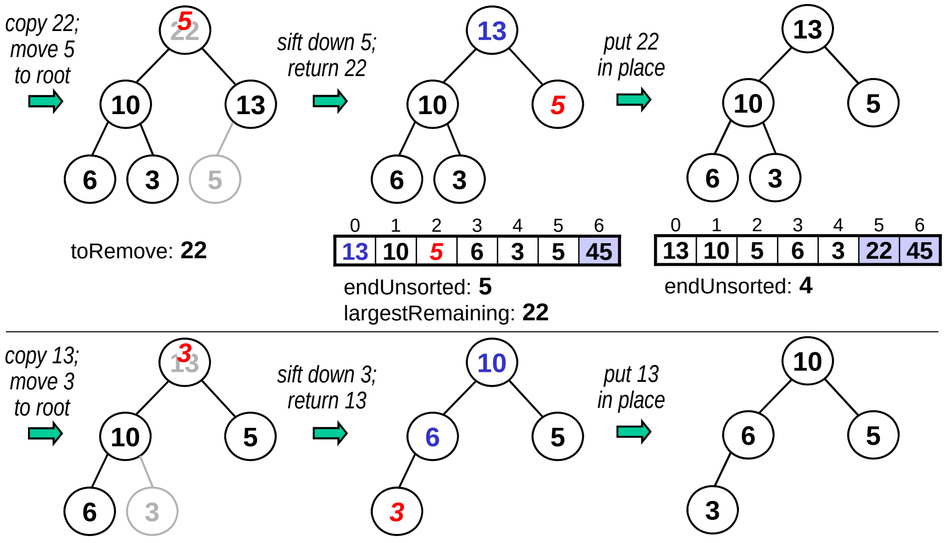
0	1	2	3	4	5	6
22	10	13	6	3	5	5

endUnsorted: **6**
largestRemaining: **45**

0	1	2	3	4	5	6
22	10	13	6	3	5	45

endUnsorted: **5**

Heapsort Example (cont.)



copy 22; move 5 to root

toRemove: **22**

sift down 5; return 22

0	1	2	3	4	5	6
13	10	5	6	3	5	45

endUnsorted: **5**
largestRemaining: **22**

put 22 in place

0	1	2	3	4	5	6
13	10	5	6	3	22	45

endUnsorted: **4**

copy 13; move 3 to root

toRemove: **13**

sift down 3; return 13

0	1	2	3	4	5	6
10	6	5	3	3	22	45

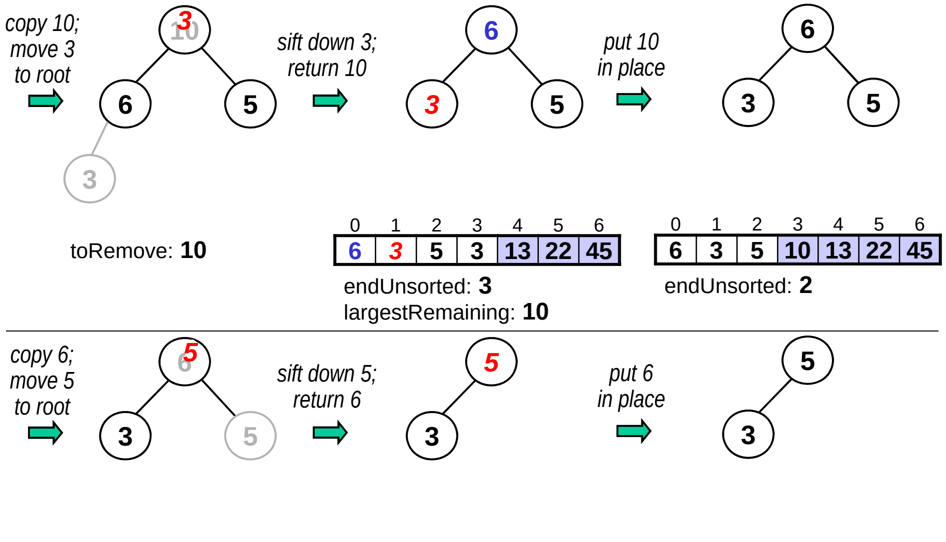
endUnsorted: **4**
largestRemaining: **13**

put 13 in place

0	1	2	3	4	5	6
10	6	5	3	13	22	45

endUnsorted: **3**

Heapsort Example (cont.)



copy 10; move 3 to root

toRemove: **10**

sift down 3; return 10

0	1	2	3	4	5	6
6	3	5	3	13	22	45

endUnsorted: **3**
largestRemaining: **10**

put 10 in place

0	1	2	3	4	5	6
6	3	5	10	13	22	45

endUnsorted: **2**

copy 6; move 5 to root

toRemove: **6**

sift down 5; return 6

0	1	2	3	4	5	6
5	3	5	10	13	22	45

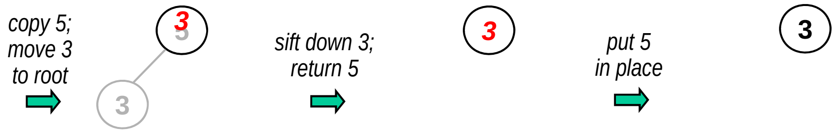
endUnsorted: **2**
largestRemaining: **6**

put 6 in place

0	1	2	3	4	5	6
5	3	6	10	13	22	45

endUnsorted: **1**

Heapsort Example (cont.)



toRemove: **5**

0	1	2	3	4	5	6
3	**3**	**6**	**10**	**13**	**22**	**45**

endUnsorted: **1**
largestRemaining: **5**

0	1	2	3	4	5	6
3	**5**	**6**	**10**	**13**	**22**	**45**

endUnsorted: **0**

How Does Heapsort Compare?

algorithm	best case	avg case	worst case	extra memory
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Shell sort	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(1)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n\log n)$	$O(n)$
heapsort	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n\log n)}$	$\boldsymbol{O(1)}$

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
    - heapsort will scramble an almost sorted array before sorting it
- Quicksort is still typically fastest in the average case.

Unit 9, Part 4

Hash Tables

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Data Dictionary Revisited

- We've considered several data structures that allow us to store and search for data items using their keys fields:

data structure	***searching for an item***	***inserting an item***
a list implemented using an array	*O*(log n) using binary search	*O*(n)
a list implemented using a linked list	*O*(n) using linear search	*O*(n)
binary search tree		
balanced search trees (2-3 tree, B-tree, others)		

- Today, we'll look at hash tables, which allow us to do better than *O*(log n).

Ideal Case: Searching = Indexing

- The optimal search and insertion performance is achieved when we can treat the key as an index into an array.
- Example: storing data about members of a sports team
  - key = jersey number (some value from 0-99).
  - class for an individual player's record:

    ```
    public class Player {
        private int jerseyNum;
        private String firstName;
        …
    }
    ```
  - store the player records in an array:

    ```
    Player[] teamRecords = new Player[100];
    ```
- In such cases, we can perform both search and insertion in *O*(1) time. For example:

  ```
  public Player search(int jerseyNum) {
      return teamRecords[jerseyNum];
  }
  ```

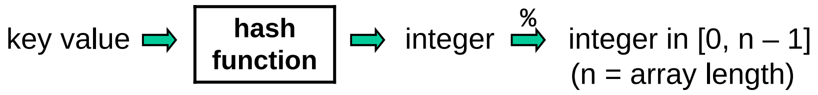
Hashing: Turning Keys into Array Indices

- In most real-world problems, indexing is not as simple as it is in the sports-team example. Why?
  -
  -
  -
- To handle these problems, we perform *hashing*:
  - use a *hash function* to convert the keys into array indices

    `"Sullivan" → 18`
  - use techniques to handle cases in which multiple keys are assigned the same hash value
- The resulting data structure is known as a *hash table*.

Hash Functions

- A hash function defines a mapping from the set of possible keys to the set of integers.
- We then use the modulus operator to get a valid array index.

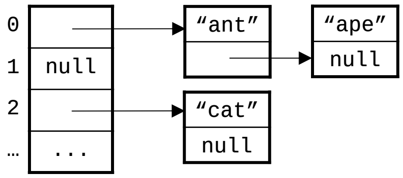


key value ⇨ **hash function** ⇨ integer $\xrightarrow{\%}$ integer in [0, n – 1] (n = array length)

- Here's a very simple hash function for keys of lower-case letters:
  h(key) = Unicode value of first char – Unicode value of 'a'
  - examples:
    h("ant") = Unicode for 'a' – Unicode for 'a' = 0
    h("cat") = Unicode for 'c' – Unicode for 'a' = 2
- h(key) is known as the key's *hash code.*
- A *collision* occurs when items with different keys are assigned the same hash code.

Dealing with Collisions I: Separate Chaining

- If multiple items are assigned the same hash code, we "chain" them together.
- Each position in the hash table serves as a *bucket* that is able to store multiple data items.
- Two implementations:
  1. each bucket is itself an array
     - disadvantages:
       - large buckets can waste memory
       - a bucket may become full; *overflow* occurs when we try to add an item to a full bucket
  2. each bucket is a linked list
     - disadvantage:
       - the references in the nodes use additional memory



Dealing with Collisions II: Open Addressing

- When the position assigned by the hash function is occupied, find another open position.
- Example: “wasp” has a hash code of 22, but it ends up in position 23, because position 22 is occupied.
- We will consider three ways of finding an open position – a process known as *probing*.
- The hash table also performs probing to search for an item.
    - example: when searching for “wasp”, we look in position 22 and then look in position 23
    - we can only stop a search when we reach an empty position

0	“ant”
1	
2	“cat”
3	
4	“emu”
5	
6	
7	
…	...
22	“wolf”
23	**“wasp”**
24	“yak”
25	“zebra”

Linear Probing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, …, wrapping around as necessary.
- Examples:
    - “ape” (h = 0) would be placed in position 1, because position 0 is already full.
    - “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
    - where would “zebu” end up?
- Advantage: if there is an open position, linear probing will eventually find it.
- Disadvantage: “clusters” of occupied positions develop, which tends to increase the lengths of subsequent probes.
    - probe length = the number of positions considered during a probe

0	“ant”
1	“ape”
2	“cat”
3	“bear”
4	“emu”
5	
6	
7	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Quadratic Probing

- Probe sequence: h(key), h(key) + 1, h(key) + 4, h(key) + 9, …, wrapping around as necessary.
  - the offsets are perfect squares: h + 1^2, h + 2^2, h + 3^2, …
- Examples:
  - “ape” (h = 0): try 0, 0 + 1 – open!
  - “bear” (h = 1): try 1, 1 + 1, 1 + 4 – open!
  - “zebu”?
- Advantage: reduces clustering
- Disadvantage: it may fail to find an existing open position. For example:

table size = 10
x = occupied

trying to insert a
key with h(key) = 0

offsets of the probe
sequence in italics

0	x		5	x	*25*
1	x	*1 81*	6	x	*16 36*
2			7		
3			8		
4	x	*4 64*	9	x	*9 49*

0	“ant”
1	“ape”
2	“cat”
3	
4	“emu”
5	“bear”
6	
7	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Double Hashing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - probe sequence: h1, h1 + h2, h1 + 2*h2, …
- Examples:
  - h1 = our previous h
  - h2 = number of characters in the string
  - “ape” (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - “bear” (h1 = 1, h2 = 4): try 1 – open!
  - “zebu”?
- Combines the good features of linear and quadratic probing:
  - reduces clustering
  - will find an open position if there is one, provided the table size is a prime number

0	“ant”
1	“bear”
2	“cat”
3	“ape”
4	“emu”
5	
6	
7	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

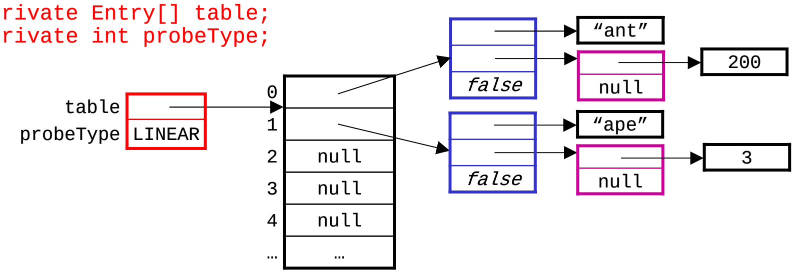
Removing Items Under Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item, but “bear” is further down in the table
- When we remove an item from a position, we need to leave a special value in that position to indicate that an item was removed.
- Three types of positions: occupied, empty, “removed”.
- We stop probing when we encounter an empty position, but not when we encounter a removed position.
- We can insert items in either empty or removed positions.

0	“ant”
1	
2	“cat”
3	“bear”
4	“emu”
5	
…	...
22	“wolf”
23	“wasp”
24	“yak”
25	“zebra”

Implementation

```
public class HashTable {
    private class Entry {
        private String key;
        private LLList valueList;
        private boolean hasBeenRemoved;
        …
    }
    …
    private Entry[] table;
    private int probeType;
}
```



- We use a private inner class for the entries in the hash table.
- To handle duplicates, we maintain a list of values for each key.
- When we remove a key and its values, we set the `Entry`’s `hasBeenRemoved` field to `true`; this indicates that the position is a removed position.

Probing Using Double Hashing

```
private int probe(String key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function

    // keep probing until we get an empty position or match
    // (write this together)




    return i;
}
```

- We'll assume that removed positions have a key of null.
  - thus, for non-empty positions, it's always okay to compare the probe key with the key in the `Entry`

Avoiding an Infinite Loop

- The while loop in our probe method could lead to an infinite loop.

  ```
  while (table[i] != null && !key.equals(table[i].key)) {
      i = (i + h2) % table.length;
  }
  ```

- When would this happen?

- We can stop probing after checking n positions (n = table size), because the probe sequence will just repeat after that point.
  - for quadratic probing:

    $(h1 + n^2) \% n = h1 \% n$

    $(h1 + (n+1)^2) \% n = (h1 + n^2 + 2n + 1) \% n = (h1 + 1) \% n$
  - for double hashing:

    $(h1 + n*h2) \% n = h1 \% n$

    $(h1 + (n+1)*h2) \% n = (h1 + n*h2 + h2) \% n = (h1 + h2) \% n$

Avoiding an Infinite Loop (cont.)

```
private int probe(String key) {
    int i = h1(key);    // first hash function
    int h2 = h2(key);   // second hash function
    int positionsChecked = 1;

    // keep probing until we get an
    // empty position or a match
    while (table[i] != null && !key.equals(table[i].key)) {
        if (positionsChecked == table.length)
            return -1;
        i = (i + h2) % table.length;
        positionsChecked++;
    }

    return i;
}
```

Search and Removal

- Both of these methods begin by probing for the key.

```
public LLList search(String key) {
    int i = probe(key);
    if (i == -1 || table[i] == null)
        return null;
    else
        return table[i].valueList;
}

public void remove(String key) {
    int i = probe(key);
    if (i == -1 || table[i] == null)
        return;

    table[i].key = null;
    table[i].valueList = null;
    table[i].hasBeenRemoved = true;
}
```

Insertion

- We begin by probing for the key.
- Several cases:
  1. the key is already in the table (we're inserting a duplicate)
     → add the value to the valueList in the key's Entry
  2. the key is not in the table: three subcases:
     a. encountered 1 or more removed positions while probing
        → put the (key, value) pair in the *first* removed position that we encountered while searching for the key.
        why does this make sense?

     b. no removed position; reached an empty position
        → put the (key, value) pair in the empty position

     c. no removed position or empty position encountered
        → overflow; throw an exception

Tracing Through Some Examples

- Start with the hashtable at right with:
  - double hashing
  - our earlier hash functions h1 and h2
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - delete "emu"
  - search "eel"
  - insert "bee"

0	“ant”
1	
2	“cat”
3	
4	"emu"
5	"fox"
6	
7	
8	
9	
10	

Dealing with Overflow

- Overflow = can't find a position for an item
- When does it occur?
  - linear probing:
  - quadratic probing:
    -
    -
  - double hashing:
    - if the table size is a prime number: same as linear
    - if the table size is not a prime number: same as quadratic
- To avoid overflow (and reduce search times), grow the hash table when the percentage of occupied positions gets too big.
  - problem: if we're not careful, we can end up needing to rehash **all** of the existing items
  - approaches exist that limit the number of rehashed items

Implementing the Hash Function

- Characteristics of a good hash function:
  1) efficient to compute
  2) uses the entire key
     - changing any char/digit/etc. should change the hash code
  3) distributes the keys more or less uniformly across the table
  4) must be a function!
     - a key must always get the same hash code
- In Java, every object has a `hashCode()` method.
  - the version inherited from `Object` returns a value based on an object’s memory location
  - classes can override this version with their own

Hash Functions for Strings: version 1

- h_a = the sum of the characters' Unicode values
- Example: h_a("eat") = 101 + 97 + 116 = 314
- All permutations of a given set of characters get the same code.
  - example: h_a("tea") = h_a("eat")
  - could be useful in a Scrabble game
    - allow you to look up all words that can be formed from a given set of characters
- The range of possible hash codes is very limited.
  - example: hashing keys composed of 1-5 lower-case char's (padded with spaces)
  - 26*27*27*27*27 = over 13 million possible keys
  - smallest code = h_a("a   ") = 97 + 4*32 = 225
    largest code = h_a("zzzzz") = 5*122 = 610

    } 610 – 225
    = 385 codes

Hash Functions for Strings: version 2

- Compute a *weighted* sum of the Unicode values:

  $$h_b = a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b + a_{n-1}$$

  where a_i = Unicode value of the ith character
  b = a constant
  n = the number of characters

- Multiplying by powers of b allows the *positions* of the characters to affect the hash code.
  - different permutations get different codes
- We may get arithmetic overflow, and thus the code may be negative. We adjust it when this happens.
- Java uses this hash function with b = 31 in the `hashCode()` method of the `String` class.

Hash Table Efficiency

- In the best case, search and insertion are *O*(1).
- In the worst case, search and insertion are linear.
  - open addressing: *O*(m), where m = the size of the hash table
  - separate chaining: *O*(n), where n = the number of keys
- With good choices of hash function and table size, complexity is generally better than *O*(log n) and approaches *O*(1).
- *load factor* = # keys in table / size of the table. To prevent performance degradation:
  - open addressing: try to keep the load factor < 1/2
  - separate chaining: try to keep the load factor < 1
- Time-space tradeoff: bigger tables have better performance, but they use up more memory.

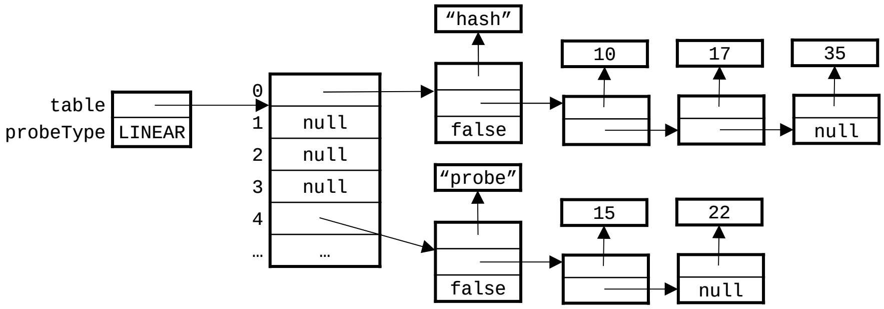
Hash Table Limitations

- It can be hard to come up with a good hash function for a particular data set.
- The items are not ordered by key. As a result, we can’t easily:
  - print the contents in sorted order
  - perform a range search
  - perform a rank search – get the kth largest item

  We *can* do all of these things with a search tree.

Application of Hashing: Indexing a Document

- Read a text document from a file and create an index of the line numbers on which each word appears.
- Use a hash table to store the index:
  - key = word
  - values = line numbers in which the word appears



- See `WordIndex.java`

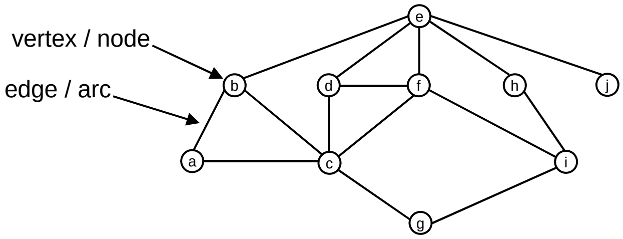
Unit 10

Graphs

Computer Science S-111
Harvard University

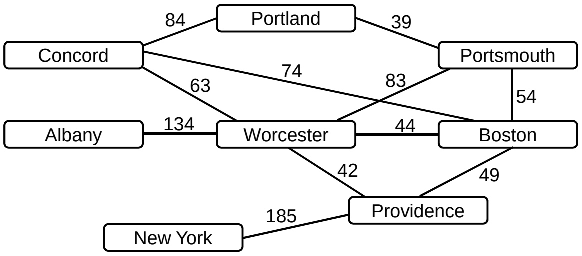
David G. Sullivan, Ph.D.

What is a Graph?



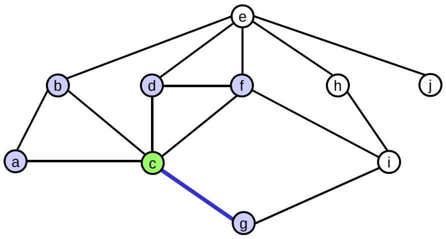
- A graph consists of:
  - a set of *vertices* (also known as *nodes*)
  - a set of *edges* (also known as *arcs*), each of which connects a pair of vertices

Example: A Highway Graph



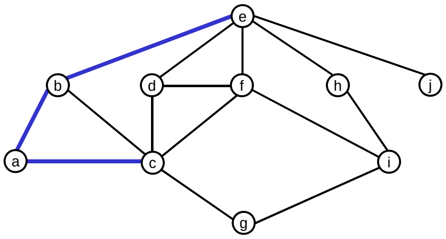
- Vertices represent cities.
- Edges represent highways.
- This is a *weighted* graph, because it has a *cost* associated with each edge.
  - for this example, the costs denote mileage
- We'll use graph algorithms to answer questions like "What is the shortest route from Portland to Providence?"

Relationships Among Vertices

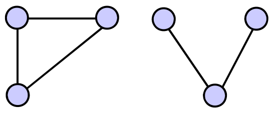


- Two vertices are *adjacent* if they are connected by a single edge.
  - *ex:* c and g are adjacent, but c and i are not
- The collection of vertices that are adjacent to a vertex v are referred to as v's *neighbors.*
  - *ex:* c's neighbors are a, b, d, f, and g

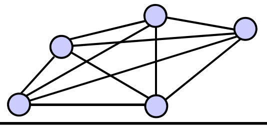
Paths in a Graph



- A *path* is a sequence of edges that connects two vertices.
  - *ex:* the path highlighted above connects c and e



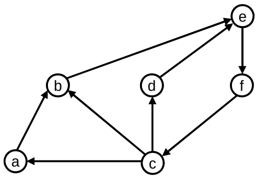
- A graph is *connected* if there is a path between any two vertices.
  - *ex:* the six vertices at right are part of a graph that is *not* connected



- A graph is *complete* if there is an edge between every pair of vertices.
  - *ex:* the graph at right is complete

Directed Graphs

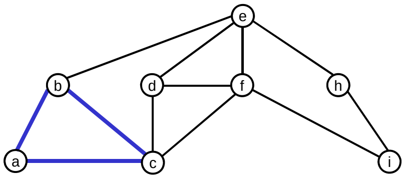
- A *directed* graph has a direction associated with each edge, which is depicted using an arrow:



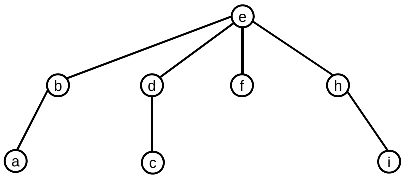
- Edges in a directed graph are often represented as ordered pairs of the form (start vertex, end vertex).
  - *ex:* (a, b) is an edge in the graph above, but (b, a) is not.
- A path in a directed graph is a sequence of edges in which the end vertex of edge i must be the same as the start vertex of edge i + 1.
  - *ex:* { (a, **b**), (**b**, **e**), (**e**, f) } is a valid path.
{ (a, **b**), (**c**, **b**), (**c**, a) } is not.

Trees vs. Graphs

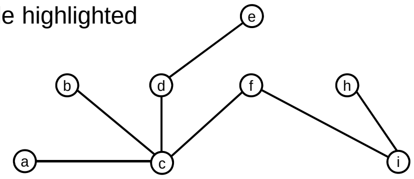
- A tree is a special type of graph.
  - it is connected and undirected
  - it is *acyclic:* there is no path containing distinct edges that starts and ends at the same vertex
  - we usually single out one of the vertices to be the root of the tree, although graph theory does not require this



a graph that is *not* a tree,
with one cycle highlighted



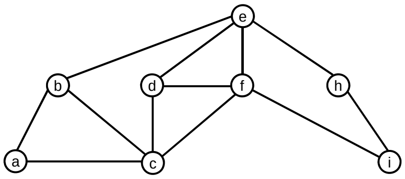
a tree using the same nodes



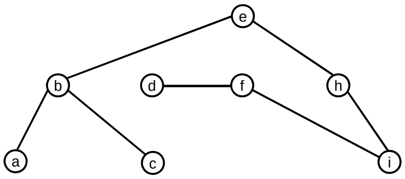
another tree using the same nodes

Spanning Trees

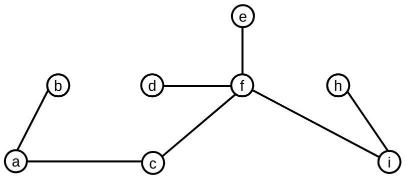
- A spanning tree is a subset of a connected graph that contains:
  - all of the vertices
  - a subset of the edges that form a tree
- The trees on the previous page were examples of spanning trees for the graph on that page. Here are two others:



the original graph



another spanning tree for this graph

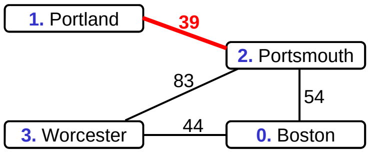


another spanning tree for this graph

Representing a Graph Using an Adjacency Matrix

- Adjacency matrix = a two-dimensional array that is used to represent the edges and any associated costs
  - edge[r][c] = the cost of going from vertex r to vertex c
- Example:

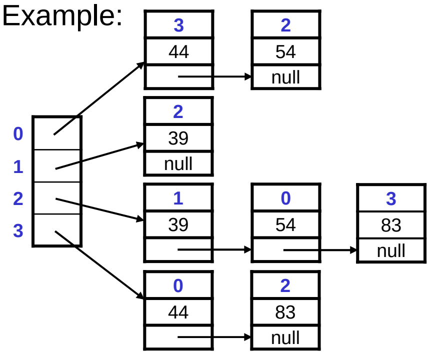
	0	1	2	3
0	▨	▨	54	44
1	▨	▨	39	▨
2	54	39	▨	83
3	44	▨	83	▨

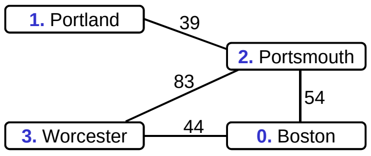


- Use a special value to indicate that you can't go from r to c.
  - either there's no edge between r and c, or it's a directed edge that goes from c to r
  - this value is shown as a shaded cell in the matrix above
  - we can't use 0, because we may have actual costs of 0
- This representation is good if a graph is *dense* – if it has many edges per vertex – but wastes memory if the graph is *sparse* – if it has few edges per vertex.

Representing a Graph Using an Adjacency List

- Adjacency list = a list (either an array or linked list) of linked lists that is used to represent the edges and any associated costs
- Example:

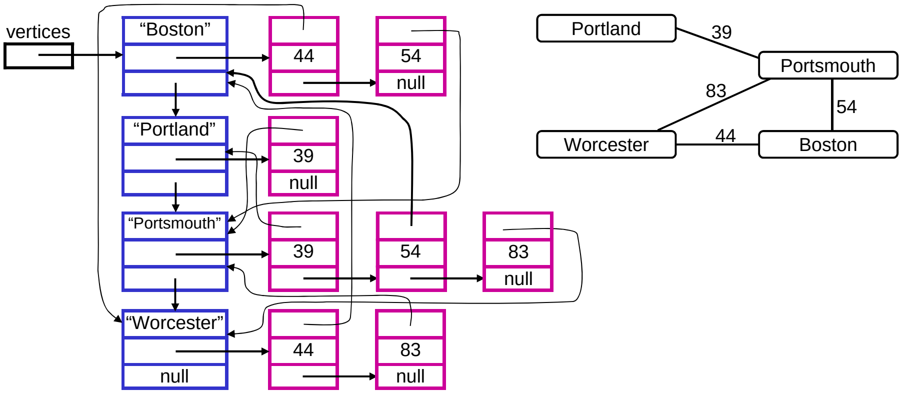




- No memory is allocated for non-existent edges, but the references in the linked lists use extra memory.
- This representation is good if a graph is sparse, but wastes memory if the graph is dense.

Our Graph Representation

- Use a linked list of linked lists for the adjacency list.
- Example:



`vertices` is a reference to a linked list of `Vertex` objects.
Each `Vertex` holds a reference to a linked list of `Edge` objects.
Each `Edge` holds a reference to the `Vertex` that is the end vertex.

Graph Class

```
public class Graph {
    private class Vertex {
        private String id;
        private Edge edges;          // adjacency list
        private Vertex next;
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }
    private class Edge {
        private Vertex start;
        private Vertex end;
        private double cost;
        private Edge next;
        …
    }
    private Vertex vertices;
    …
}
```

The highlighted fields are shown in the diagram on the previous page.

Traversing a Graph

- Traversing a graph involves starting at some vertex and visiting all of the vertices that can be reached from that vertex.
  - visiting a vertex = processing its data in some way
    - example: print the data
  - if the graph is connected, all of the vertices will be visited
- We will consider two types of traversals:
  - **depth-first**: proceed as far as possible along a given path before backing up
  - **breadth-first**: visit a vertex
visit all of its neighbors
visit all unvisited vertices 2 edges away
visit all unvisited vertices 3 edges away, etc.
- Applications:
  - determining the vertices that can be reached from some vertex
  - web crawler (vertices = pages, edges = links)

Depth-First Traversal

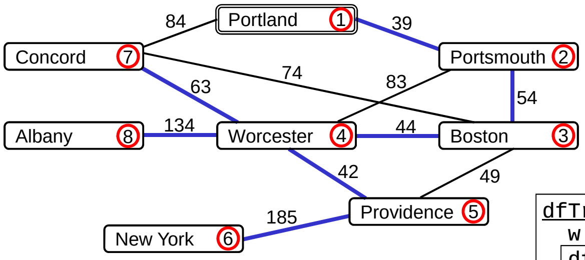
- Visit a vertex, then make recursive calls on all of its yet-to-be-visited neighbors:

```
dfTrav(v, parent)
    visit v and mark it as visited
    v.parent = parent
    for each vertex w in v’s neighbors
        if (w has not been visited)
            dfTrav(w, v)
```

- Java method:

```
private static void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id);    // visit v
    v.done = true;
    v.parent = parent;

    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;
        if (!w.done)
            dfTrav(w, v);
        e = e.next;
    }
}
```

Example: Depth-First Traversal from Portland

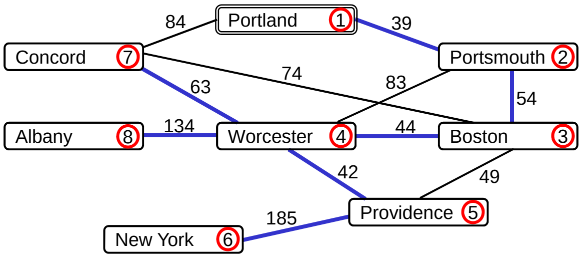


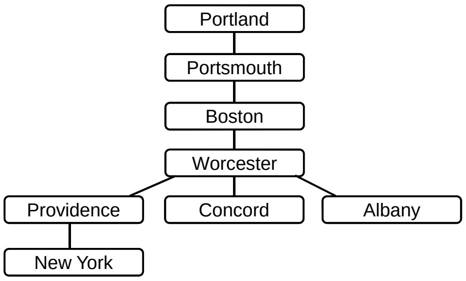
For the examples, we'll assume that the edges in each vertex's adjacency list are sorted by increasing edge cost.

```
void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id);
    v.done = true;
    v.parent = parent;
    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;
        if (!w.done)
            dfTrav(w, v);
        e = e.next;
    }
}
```

```
dfTrav(Ptl, null)
  w = Pts
  dfTrav(Pts, Ptl)
    w = Ptl, Bos
    dfTrav(Bos, Pts)
      w = Wor
      dfTrav(Wor, Bos)
        w = Pro
        dfTrav(Pro, Wor)
          w = Wor, Bos, NY
          dfTrav(NY, Pro)
            w = Pro
            return
          no more neighbors
          return
        w = Bos, Con
        dfTrav(Con, Wor)
          …
```

Depth-First Spanning Tree



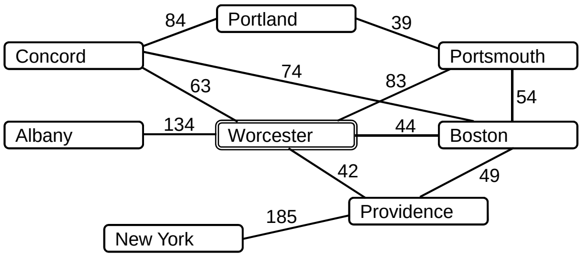


The edges obtained by following the `parent` references form a spanning tree with the origin of the traversal as its root.

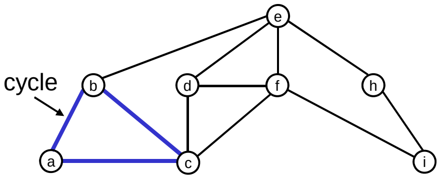
From any city, we can get to the origin by following the roads in the spanning tree.

Another Example: Depth-First Traversal from Worcester

- In what order will the cities be visited?
- Which edges will be in the resulting spanning tree?



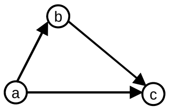
Checking for Cycles in an Undirected Graph



- To discover a cycle in an undirected graph, we can:
  - perform a depth-first traversal, marking the vertices as visited
  - when considering neighbors of a visited vertex, if we discover one already marked as visited, there must be a cycle

- If no cycles found during the traversal, the graph is acyclic.

- This doesn't work for directed graphs:
  - c is a neighbor of both a and b
  - there is no cycle

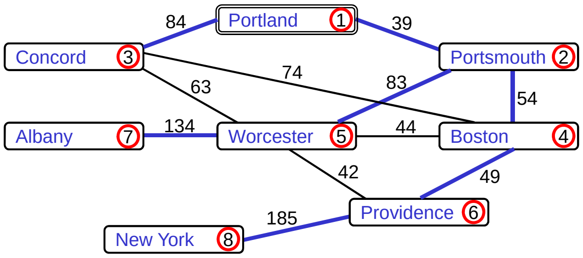


Breadth-First Traversal

- Use a queue, as we do for level-order tree traversal:

```
private static void bfTrav(Vertex origin) {
    origin.encountered = true;
    origin.parent = null;
    Queue<Vertex> q = new LLQueue<Vertex>();
    q.insert(origin);

    while (!q.isEmpty()) {
        Vertex v = q.remove();
        System.out.println(v.id);         // Visit v.

        // Add v's unencountered neighbors to the queue.
        Edge e = v.edges;
        while (e != null) {
            Vertex w = e.end;
            if (!w.encountered) {
                w.encountered = true;
                w.parent = v;
                q.insert(w);
            }
            e = e.next;
        }
    }
}
```

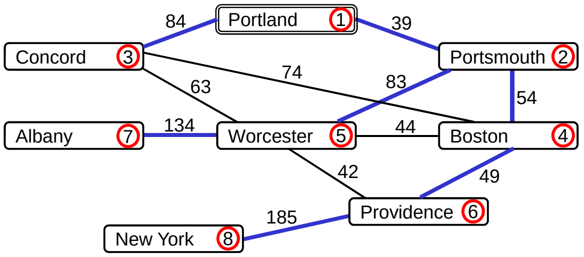
Example: Breadth-First Traversal from Portland



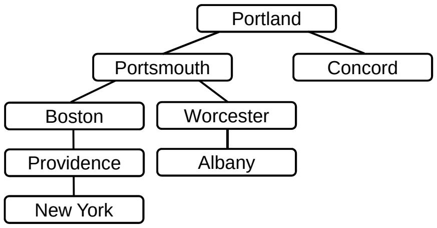
Evolution of the queue:

remove	insert	queue contents
	Portland	Portland
Portland	Portsmouth, Concord	Portsmouth, Concord
Portsmouth	Boston, Worcester	Concord, Boston, Worcester
Concord	*none*	Boston, Worcester
Boston	Providence	Worcester, Providence
Worcester	Albany	Providence, Albany
Providence	New York	Albany, New York
Albany	*none*	New York
New York	*none*	*empty*

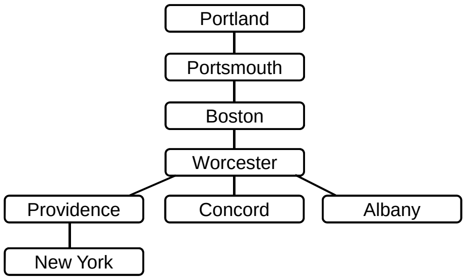
Breadth-First Spanning Tree



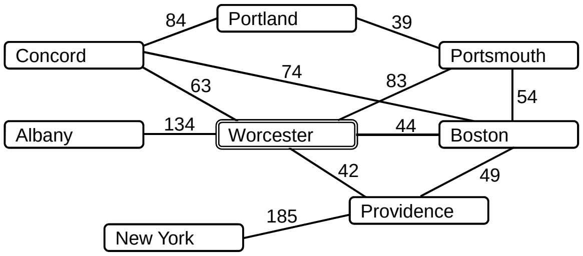
breadth-first spanning tree:



depth-first spanning tree:



Another Example: Breadth-First Traversal from Worcester



Evolution of the queue:

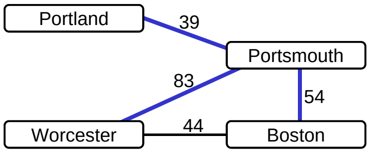
remove	insert	queue contents

Time Complexity of Graph Traversals

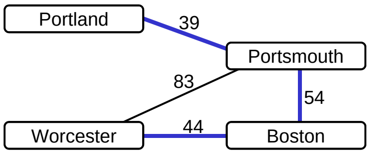
- let V = number of vertices in the graph
  E = number of edges
- If we use an adjacency matrix, a traversal requires *O*(V^2) steps.
  - why?
- If we use an adjacency list, a traversal requires *O*(V + E) steps.
  - visit each vertex once
  - traverse each vertex's adjacency list at most once
    - the total length of the adjacency lists is at most 2E = *O*(E)
  - *O*(V + E) << *O*(V^2) for a sparse graph
  - for a dense graph, E = O(V^2), so both representations are *O*(V^2)
- In our implementations of the remaining algorithms, we'll assume an adjacency-list implementation.

Minimum Spanning Tree

- A minimum spanning tree (MST) has the smallest total cost among all possible spanning trees.
  - *example:*



one possible spanning tree
(total cost = 39 + 83 + 54 = 176)

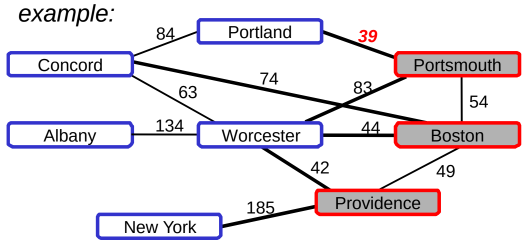


the minimal-cost spanning tree
(total cost = 39 + 54 + 44 = 137)

- If no two edges have the same cost, there is a unique MST. If two or more edges have the same cost, there may be more than one MST.
- Finding an MST could be used to:
  - determine the shortest highway system for a set of cities
  - calculate the smallest length of cable needed to connect a network of computers

Building a Minimum Spanning Tree

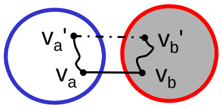
- Key insight: if you divide the vertices into two disjoint subsets A and B, then the lowest-cost edge joining a vertex in A to a vertex in B – call it (v_a, v_b) – must be part of the MST.
    - *example:*



The 6 bold edges each join an unshaded vertex to a shaded vertex.

The one with the lowest cost (Portland to Portsmouth) must be in the MST.

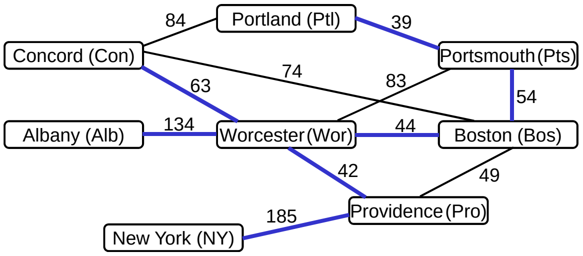
- Proof by contradiction:
  - assume we can create an MST (call it T) that doesn't include edge (v_a, v_b)
  - T must include a path from v_a to v_b, so it must include one of the other edges (v_a', v_b') that spans subsets A and B, such that (v_a', v_b') is part of the path from v_a to v_b
  - adding (v_a, v_b) to T introduces a cycle
  - removing (v_a', v_b') gives a spanning tree with lower cost, which contradicts the original assumption.



Prim's MST Algorithm

- Begin with the following subsets:
    - A = any one of the vertices
    - B = all of the other vertices
- Repeatedly do the following:
    - select the lowest-cost edge (v_a, v_b) connecting a vertex in A to a vertex in B
    - add (v_a, v_b) to the spanning tree
    - move vertex v_b from set A to set B
- Continue until set A contains all of the vertices.

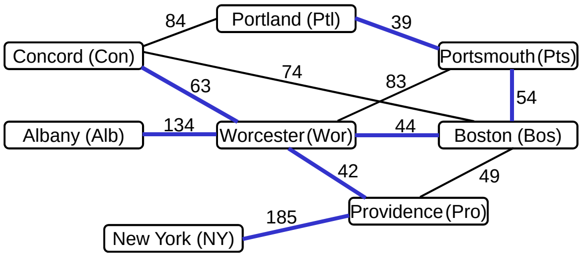
Example: Prim's Starting from Concord



- Tracing the algorithm:

edge added	set A	set B
	{Con}	{Alb, Bos, NY, Ptl, Pts, Pro, Wor}
(Con, Wor)	{Con, Wor}	{Alb, Bos, NY, Ptl, Pts, Pro}
(Wor, Pro)	{Con, Wor, Pro}	{Alb, Bos, NY, Ptl, Pts}
(Wor, Bos)	{Con, Wor, Pro, Bos}	{Alb, NY, Ptl, Pts}
(Bos, Pts)	{Con, Wor, Pro, Bos, Pts}	{Alb, NY, Ptl}
(Pts, Ptl)	{Con, Wor, Pro, Bos, Pts, Ptl}	{Alb, NY}
(Wor, Alb)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb}	{NY}
(Pro, NY)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb, NY}	{}

MST May Not Give Shortest Paths



- The MST is the spanning tree with the minimal *total* edge cost.
- It does not necessarily include the minimal cost path between a pair of vertices.
- Example: shortest path from Boston to Providence is along the single edge connecting them
  - that edge is not in the MST

Implementing Prim's Algorithm in our `Graph` class

- Use the `done` field to keep track of the sets.
  - if `v.done == true`, v is in set A
  - if `v.done == false`, v is in set B

- Repeatedly scan through the lists of vertices and edges to find the next edge to add.

  ➔ *O*(EV)

- We can do better!
  - use a heap-based priority queue to store the vertices in set B
  - priority of a vertex x = –1 `*` cost of the lowest-cost edge connecting x to a vertex in set A
    - why multiply by –1?

  - somewhat tricky: need to update the priorities over time

  ➔ *O*(E log V)

The Shortest-Path Problem

- It's often useful to know the shortest path from one vertex to another – i.e., the one with the minimal total cost
  - example application: routing traffic in the Internet

- For an *unweighted* graph, we can simply do the following:
  - start a breadth-first traversal from the origin, v
  - stop the traversal when you reach the other vertex, w
  - the path from v to w in the resulting (possibly partial) spanning tree is a shortest path

- A breadth-first traversal works for an unweighted graph because:
  - the shortest path is simply one with the fewest edges
  - a breadth-first traversal visits cities in order according to the number of edges they are from the origin.

- Why might this approach fail to work for a *weighted* graph?

Dijkstra's Algorithm

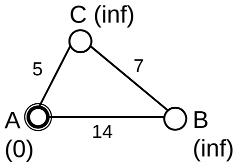
- One algorithm for solving the shortest-path problem for weighted graphs was developed by E.W. Dijkstra.

- It allows us to find the shortest path from a vertex v (the origin) to *all other vertices* that can be reached from v.

- Basic idea:
  - maintain estimates of the shortest paths from the origin to every vertex (along with their costs)
  - gradually refine these estimates as we traverse the graph

- Initial estimates:

	path	cost
the origin itself:	stay put!	0
all other vertices:	unknown	infinity

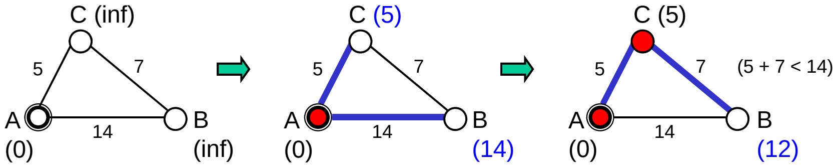


Dijkstra's Algorithm (cont.)

- We say that a vertex w is *finalized* if we have found the shortest path from v to w.

- We repeatedly do the following:
  - find the unfinalized vertex w with the lowest cost estimate
  - mark w as finalized (shown as a filled circle below)
  - examine each unfinalized neighbor x of w to see if there is a shorter path to x that passes through w
    - if there is, update the shortest-path estimate for x

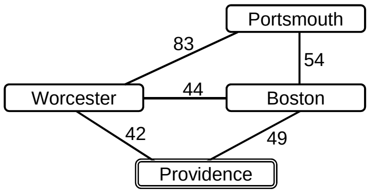
- Example:



Another Example: Shortest Paths from Providence

- Initial estimates:

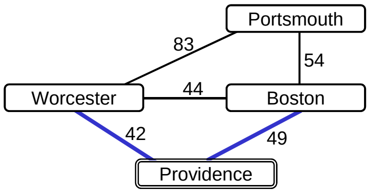
Boston	infinity
Worcester	infinity
Portsmouth	infinity
Providence	***0***



- Providence has the smallest unfinalized estimate, so we finalize it.

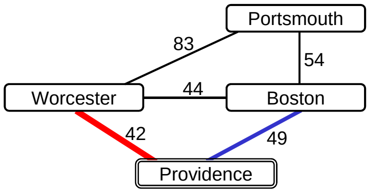
- We update our estimates for its neighbors:

Boston	*49* (< infinity)
Worcester	*42* (< infinity)
Portsmouth	infinity
Providence	0



Shortest Paths from Providence (cont.)

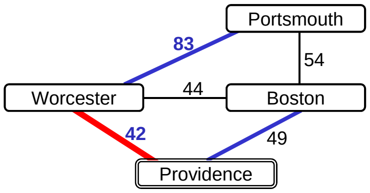
Boston	49
Worcester	***42***
Portsmouth	infinity
Providence	0



- Worcester has the smallest unfinalized estimate, so we finalize it.
  - any other route from Prov. to Worc. would need to go via Boston, and since (Prov → Worc) < (Prov → Bos), we can't do better.

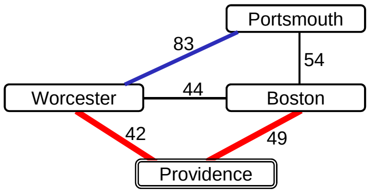
- We update our estimates for Worcester's unfinalized neighbors:

Boston	49 (no change)
Worcester	42
Portsmouth	*125* (42 + 83 < infinity)
Providence	0



Shortest Paths from Providence (cont.)

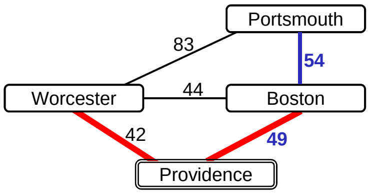
Boston	***49***
Worcester	42
Portsmouth	125
Providence	0



- Boston has the smallest unfinalized estimate, so we finalize it.
  - we'll see later why we can safely do this!

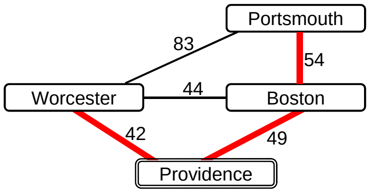
- We update our estimates for Boston's unfinalized neighbors:

Boston	49
Worcester	42
Portsmouth	*103* (49 + 54 < 125)
Providence	0



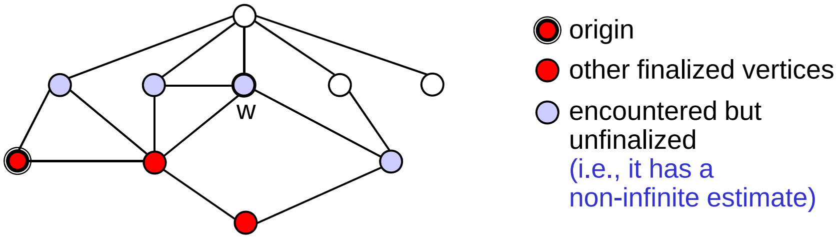
Shortest Paths from Providence (cont.)

Boston	49
Worcester	42
Portsmouth	***103***
Providence	0



- Only Portsmouth is left, so we finalize it.

Finalizing a Vertex


w
origin
other finalized vertices
encountered but unfinalized
(i.e., it has a non-infinite estimate)

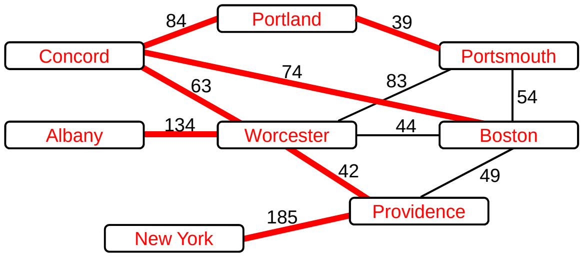
- Let w be the unfinalized vertex with the smallest cost estimate. Why can we finalize w, before seeing the rest of the graph?
- We know that w's current estimate is for the shortest path to w that passes through only *finalized* vertices.
- Any shorter path to w would have to pass through one of the other encountered-but-unfinalized vertices, but we know that they're all further away from the origin than w is.
  - their cost estimates may decrease in subsequent stages of the algorithm, but they can't drop below w's current estimate!

Pseudocode for Dijkstra's Algorithm

```
dijkstra(origin)
   origin.cost = 0
   for each other vertex v
        v.cost = infinity;

   while there are still unfinalized vertices with cost < infinity
        find the unfinalized vertex w with the minimal cost
        mark w as finalized

        for each unfinalized vertex x adjacent to w
            cost_via_w = w.cost + edge_cost(w, x)
            if (cost_via_w < x.cost)
                x.cost = cost_via_w
                x.parent = w
```

- At the conclusion of the algorithm, for each vertex v:
  - v.cost is the cost of the shortest path from the origin to v; if v.cost is infinity, there is no path from the origin to v
  - starting at v and following the parent references yields the shortest path
- The Java version is in `Graph.java`

Example: Shortest Paths from Concord

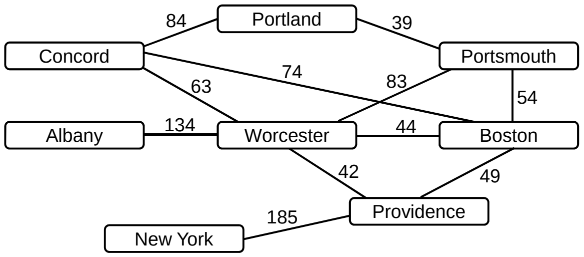


Evolution of the cost estimates (costs in bold have been finalized):

Albany	inf	inf	197	197	197	197	***197***	
Boston	inf	74	***74***					
Concord	***0***							
New York	inf	inf	inf	inf	inf	290	290	***290***
Portland	inf	84	84	***84***				
Portsmouth	inf	inf	146	128	123	***123***		
Providence	inf	inf	105	105	***105***			
Worcester	inf	***63***						

Note that the Portsmouth estimate was improved three times!

Another Example: Shortest Paths from Worcester



Evolution of the cost estimates (costs in bold have been finalized):

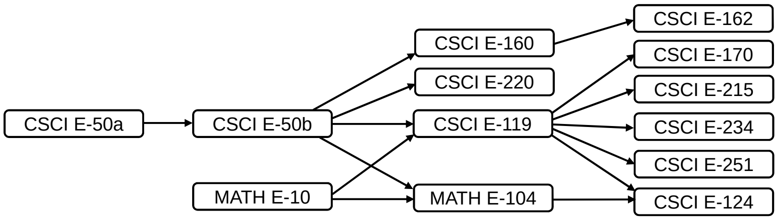
Albany								
Boston								
Concord								
New York								
Portland								
Portsmouth								
Providence								
Worcester								

Implementing Dijkstra's Algorithm

- Similar to the implementation of Prim's algorithm.
- Use a heap-based priority queue to store the unfinalized vertices.
  - priority = ?
- Need to update a vertex's priority whenever we update its shortest-path estimate.
- Time complexity = $O(E\log V)$

Topological Sort

- Used to order the vertices in a directed acyclic graph (a DAG).
- Topological order: an ordering of the vertices such that, if there is directed edge from a to b, a comes before b.
- Example application: ordering courses according to prerequisites



  - a directed edge from a to b indicates that a is a prereq of b
- There may be more than one topological ordering.

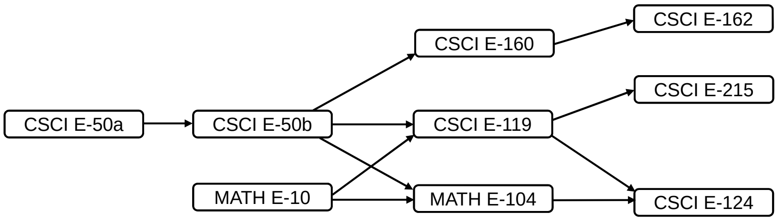
Topological Sort Algorithm

- A *successor* of a vertex v in a directed graph = a vertex w such that (v, w) is an edge in the graph (v•→•w)
- Basic idea: find vertices that have no successors and work backward from them.
  - there must be at least one such vertex. why?
- Pseudocode for one possible approach:

```
topolSort
    S = a stack to hold the vertices as they are visited
    while there are still unvisited vertices
        find a vertex v with no unvisited successors
        mark v as visited
        S.push(v)
    return S
```

- Popping the vertices off the resulting stack gives one possible topological ordering.

Topological Sort Example

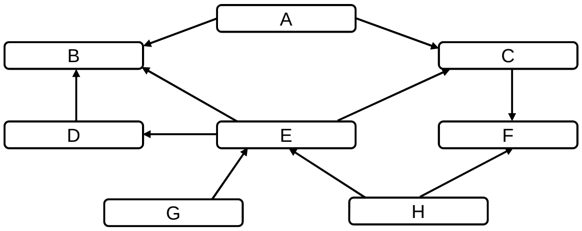


Evolution of the stack:

push	stack contents (top to bottom)
E-124	E-124
E-162	E-162, E-124
E-215	E-215, E-162, E-124
E-104	E-104, E-215, E-162, E-124
E-119	E-119, E-104, E-215, E-162, E-124
E-160	E-160, E-119, E-104, E-215, E-162, E-124
E-10	E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50b	E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50a	**E-50a, E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124**

one possible topological ordering

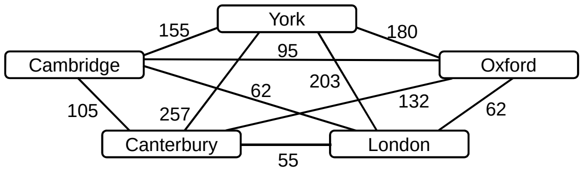
Another Topological Sort Example



Evolution of the stack:

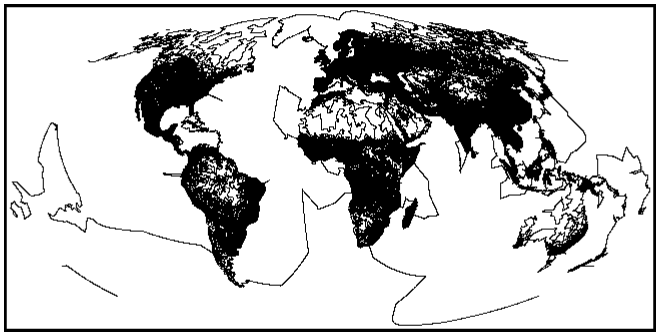
<u>push</u> <u>stack contents (top to bottom)</u>

Traveling Salesperson Problem (TSP)



- A salesperson needs to travel to a number of cities to visit clients, and wants to do so as efficiently as possible.
- As in our earlier problems, we use a weighted graph.
- A *tour* is a path that begins at some starting vertex, passes through every other vertex *once and only once*, and returns to the starting vertex. (The actual starting vertex doesn't matter.)
- TSP: find the tour with the lowest total cost
- TSP algorithms assume the graph is complete, but we can assign infinite costs if there isn't a direct route between two cities.

TSP for Santa Claus



A "world TSP" with 1,904,711 cities.

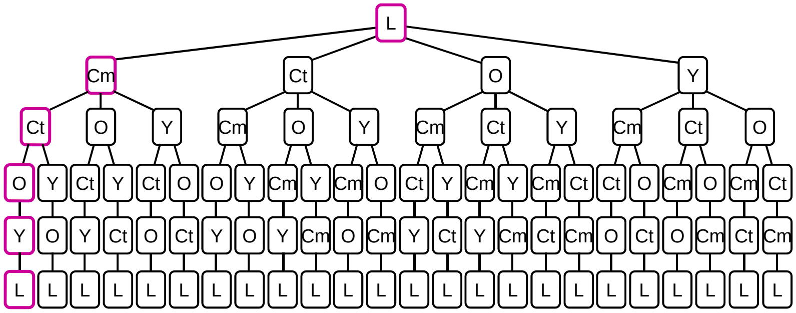
The figure at right shows a tour with a total cost of 7,516,353,779 meters – which is at most 0.068% longer than the optimal tour.

source: http://www.tsp.gatech.edu/world/pictures.html

- Other applications:
  - coin collection from phone booths
  - routes for school buses or garbage trucks
  - minimizing the movements of machines in automated manufacturing processes
  - many others

Solving a TSP: Brute-Force Approach

- Perform an exhaustive search of all possible tours.
- We can represent the set of all possible tours as a tree.


L
Cm Ct O Y
Ct O Y Cm O Y Cm Ct Y Cm Ct O
O Y Ct Y Ct O O Y Cm Y Cm O Ct Y Cm Y Cm Ct Ct O Cm O Cm Ct
Y O Y Ct O Ct Y O Y Cm O Cm Y Ct Y Cm Ct Cm O Ct O Cm Ct Cm
L L L L L L L L L L L L L L L L L L L L L L L L

- The leaf nodes correspond to possible solutions.
  - for n cities, there are (n – 1)! leaf nodes in the tree.
  - half are redundant (e.g., L-Cm-Ct-O-Y-L = L-Y-O-Ct-Cm-L)
- Problem: exhaustive search is intractable for all but small n.
  - example: when n = 14, ((n – 1)!) / 2 = over 3 billion

Solving a TSP: Informed Search

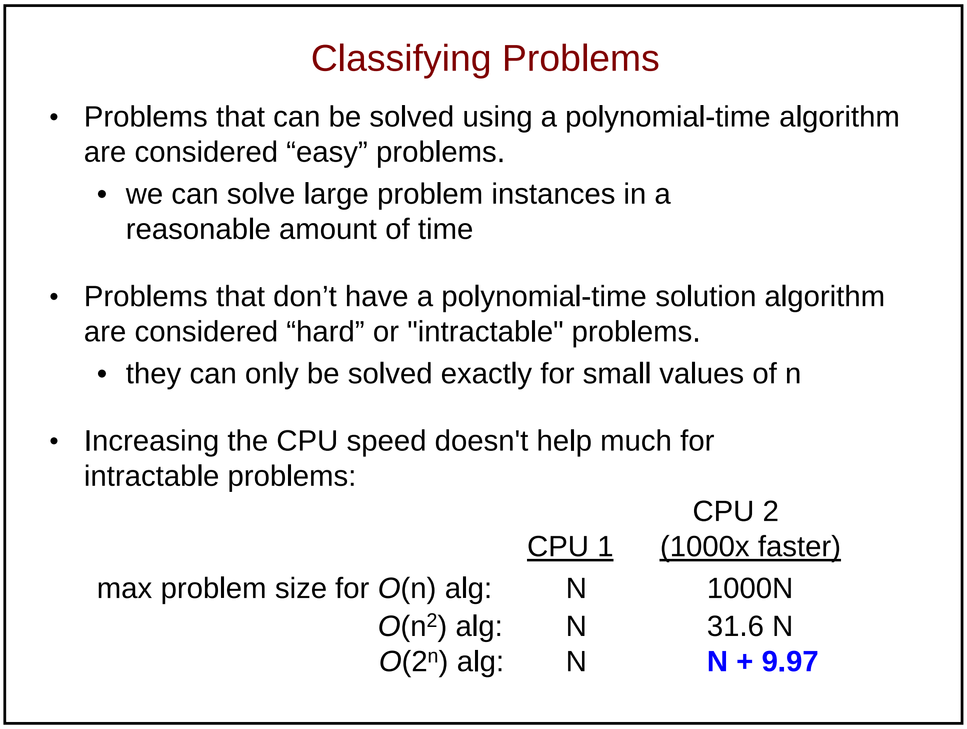
- Focus on the most promising paths through the tree of possible tours.
  - use a domain-specific function that estimates how good a given path is
- Much better than brute force, but it still uses exponential space and time.

Algorithm Analysis Revisited

- Recall that we can group algorithms into classes (n = problem size):

name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
n log n time	$4n\log_2 n$, $n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
n^c $(c > 2)$	$n^3 - 5n$, $2n^5 + 5n^2$	$O(n^c)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$
factorial time	$(n - 1)!/2$, $3n!$	$O(n!)$

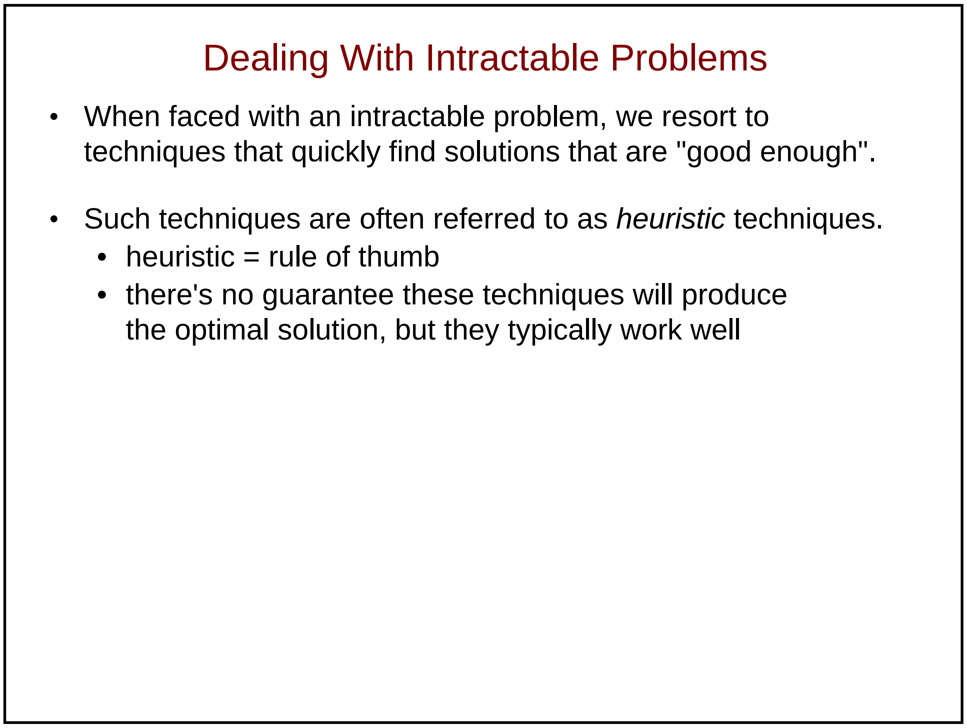
- Algorithms that fall into one of the classes above the dotted line are referred to as *polynomial-time* algorithms.
- The term *exponential-time algorithm* is sometimes used to include *all* algorithms that fall below the dotted line.
  - algorithms whose running time grows as fast or faster than c^n



Classifying Problems

- Problems that can be solved using a polynomial-time algorithm are considered “easy” problems.
  - we can solve large problem instances in a reasonable amount of time
- Problems that don’t have a polynomial-time solution algorithm are considered “hard” or "intractable" problems.
  - they can only be solved exactly for small values of n
- Increasing the CPU speed doesn't help much for intractable problems:

	CPU 1	CPU 2 (1000x faster)
max problem size for *O*(n) alg:	N	1000N
O(n^2) alg:	N	31.6 N
O(2^n) alg:	N	**N + 9.97**



Dealing With Intractable Problems

- When faced with an intractable problem, we resort to techniques that quickly find solutions that are "good enough".
- Such techniques are often referred to as *heuristic* techniques.
  - heuristic = rule of thumb
  - there's no guarantee these techniques will produce the optimal solution, but they typically work well

Take-Home Lessons

- Computer science is the science of solving problems using computers.
- Java is one programming language that we can use when solving problems computationally.
- The key concepts transcend Java:
  - flow of control
  - variables, data types, and expressions
  - conditional execution
  - procedural decomposition
  - definite and indefinite loops
  - recursion
  - console and file I/O
  - memory management (stack, heap, references)

Take-Home Lessons (cont.)

- Object-oriented programming allows us to capture the abstractions in the programs that we write.
  - creates reusable building blocks
  - key concepts: encapsulation, inheritance, polymorphism
- Abstract data types allow us to organize and manipulate collections of data.
  - a given ADT can be implemented in different ways
  - fundamental building blocks: arrays, linked nodes
- Efficiency matters when dealing with large collections of data.
  - some solutions can be *much* faster or more space efficient than others!
  - what’s the best data structure/algorithm for the specific instances of the problem that you expect to see?
    - example: sorting an almost sorted collection