Programming in Java

Boston University
David G. Sullivan, Ph.D.

In CS 112, we assume that all students have already taken a rigorous prior course in programming and computational problem-solving, but that they may not have had experience using Java. We thus spend several weeks reviewing some of the essential features of that language.

This collection of lecture notes is designed for a different type of course—one that introduces Java to students with no prior programming background. We are providing it in the hope that you may find it useful as an additional resource.

Java Basics 2

Procedural Decomposition Using Simple Methods 8

Primitive Data, Types, and Expressions 21

Definite Loops 52

Methods with Parameters and Return Values 80

Using Objects from Existing Classes 103

Conditional Execution 126

Indefinite Loops and Boolean Expressions 149

Arrays 166

Classes as Blueprints: How to Define New Types of Objects 196

Inheritance and Polymorphism 234

Java Basics

Boston 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`

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 Resources page)

- 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 the *terminal* or *console*.

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

Procedural Decomposition

(How to Use Methods to Write Better Programs)

Boston University

David G. Sullivan, Ph.D.

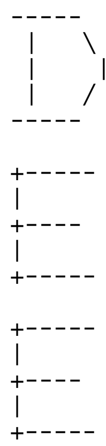
Example Program: Writing Block Letters

- 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:

  methodName`();`

- 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:

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

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

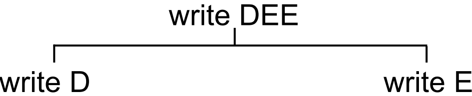
```
public class BlockLetters3 {
    public static void writeD() {
        System.out.println("    -----");
        System.out.println("     |    \\");
        System.out.println("     |     |");
        System.out.println("     |    /");
        System.out.println("    -----");
    }

    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.)

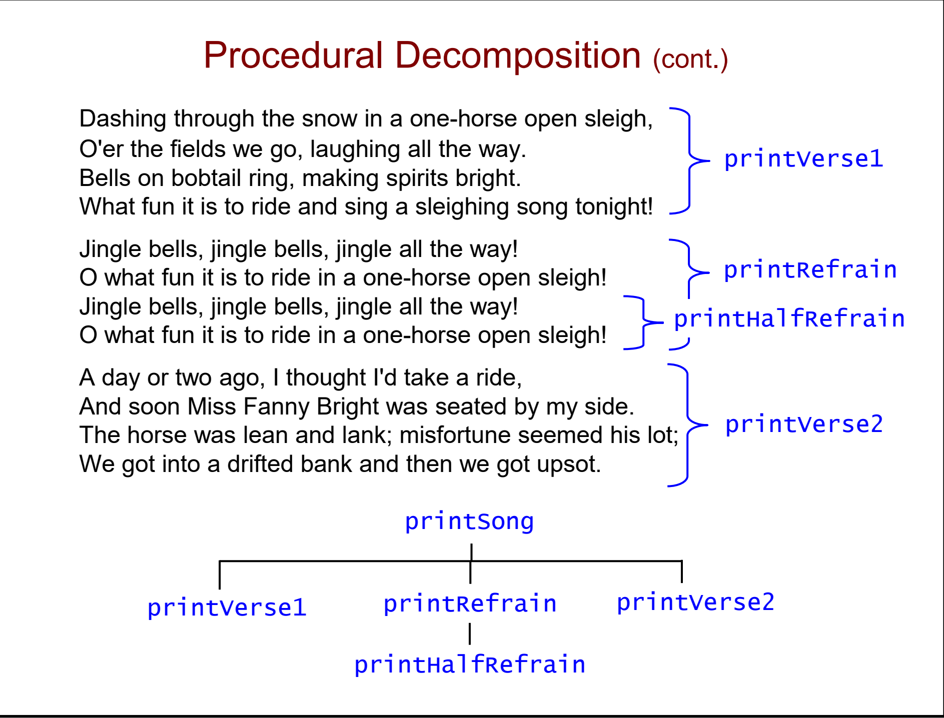
- How could we use procedural decomposition in printing the following lyrics?

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!



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.

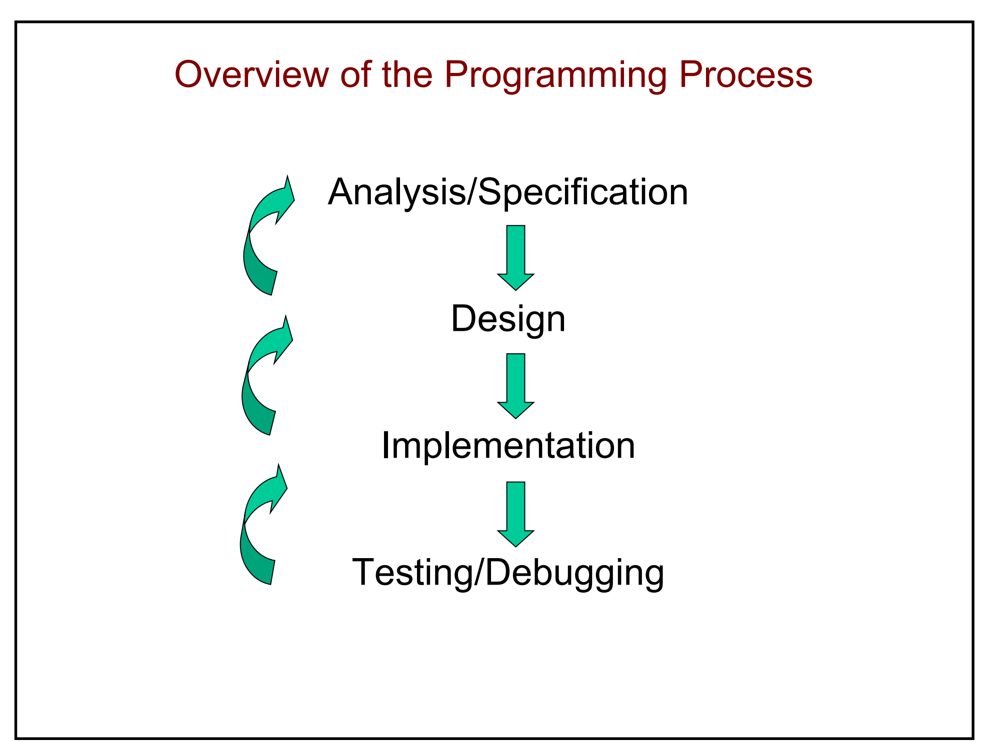
Primitive Data, Variables, and Expressions;

Simple Conditional Execution

Boston University

David G. Sullivan, Ph.D.

Overview of the Programming Process



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

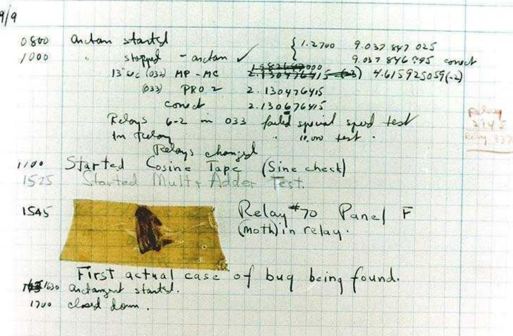
- We need to learn more Java before we can do this!
- Here's a portion or *fragment* of a Java program for computing the value of a particular collection of coins:

```
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

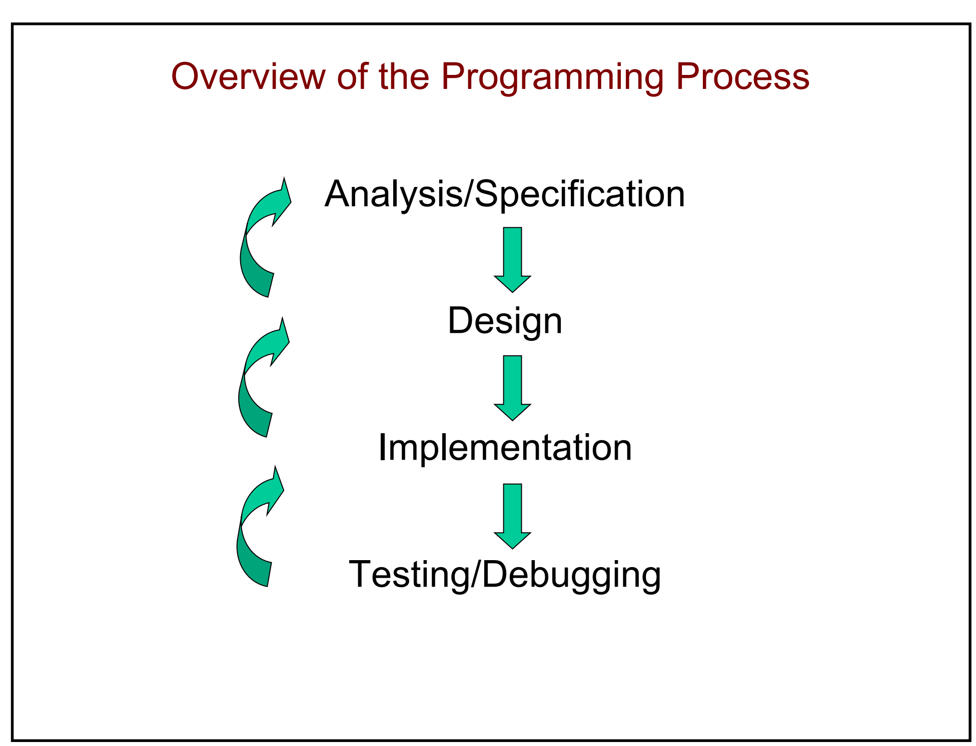
- A *bug* is an error in your program.
- *Debugging* involves finding and fixing the bugs.



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



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.

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`

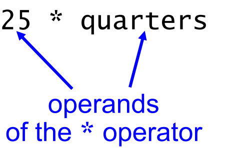
Program Building Blocks: Expressions (cont.)

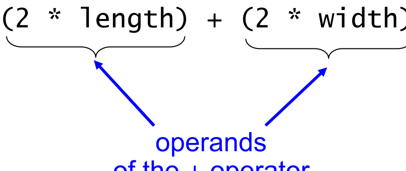
- Numerical operators include:

+	addition
-	subtraction
*	multiplication
/	division
%	modulus or mod: gives the remainder of a division

  example: `11 % 3` evaluates to 2

- Operators are applied to *operands:*





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);
```

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:

    $n \times 10^p$ 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 <u>not</u> create a permanent relationship between variables.
- Example: consider the following code fragment

```
int x = 10;
int y = x + 2;
System.out.println(y);
x = 20;
System.out.println(y);
```

it outputs:

  - 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:

Code	Memory
`int num1;` `int num2 = 120;`	num1 [?] (undefined) 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):

Code	Memory
`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:

expression	value
`5 / 3`	`1`
`11 / 5`	`2`

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

expression	value
`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` (points to `d`); value of type `int` (points to `3`)

  - 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` (points to `i`); value of type `double` (points to `7.5`)

- 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 after 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 in Java:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration

```
public class MyProgram2 {
    public static void main(String[] args) {
        System.out.println("Welcome!");
        System.out.println("Let's do some math!");
        int j = 10;                            } scope of j
        System.out.println(j / 5);
    }
}
```

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

Another Example

```
public class MyProgram3 {
    public static void method1() {
        int i = 5;                                        } scope of method1's version of i
        System.out.println(i * 3);
        int j = 10;                        } scope of j
        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;                                  } scope of main's version of i
        System.out.println(i * 6);
        method1();
    }
}
```

Local Variables vs. Global Variables

```
public class MyProgram {
    static int x = 10;     // a global variable

    public static void method1() {
        int y = 5;         // a local variable
        System.out.println(x + y);
        ...
```

- Variables that are declared inside a method are *local variables.*
  - they cannot be used outside that method.
- In theory, we can define *global variables* that are available throughout the program.
  - they are declared outside of any method, using the keyword `static`
- 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.

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.

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.
  - docs.oracle.com/javase/tutorial/java/data/numberformat.html

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?
  - identifiers?
  - expressions?
  - statements?

Definite Loops

Boston 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>

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

- 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.*

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

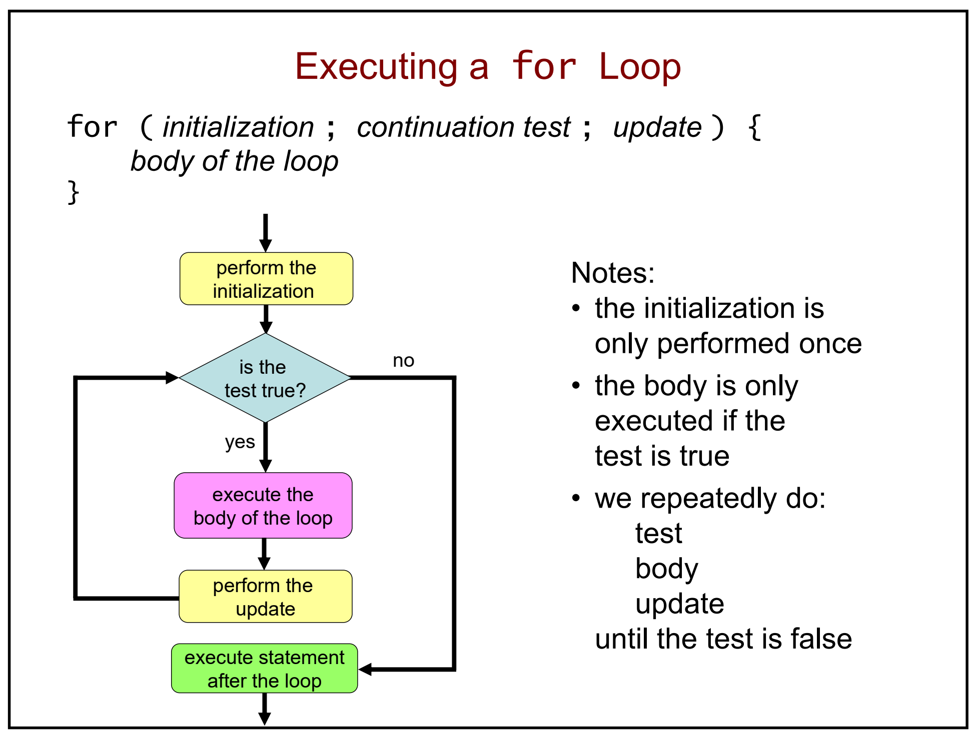
- Syntax:

for (*initialization* ; *continuation test* ; *update*) {
    one or more statements
}

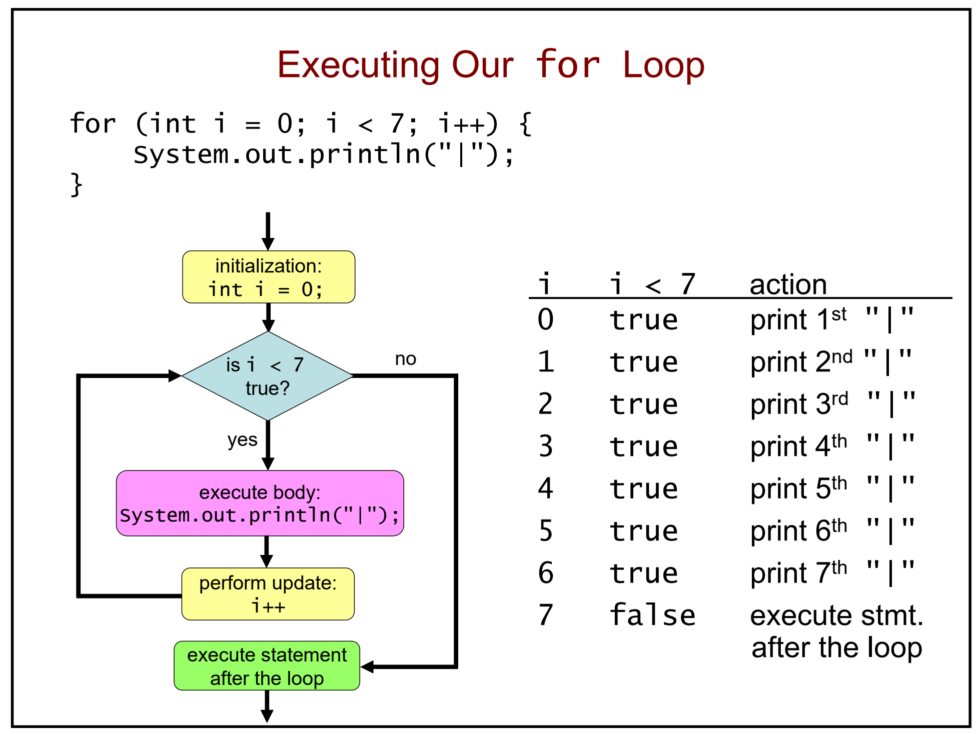
- In our example:


```
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("|");
}
initialization:
int i = 0;
is i < 7 true?
no
yes
execute body:
System.out.println("|");
perform update:
i++
execute statement after the loop
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 <u>n lines</u> 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.
- 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 in Java:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration

```
public class MyProgram2 {
    public static void main(String[] args) {
        System.out.println("Welcome!");
        System.out.println("Let's do some math!");
        int j = 10;                              } scope of j
        System.out.println(j / 5);
    }
}
```

Special Case: `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;                       } scope of i
        System.out.println(j);
    }
    // the following line won't compile
    System.out.print(i);
    System.out.println(" values were printed.");
}
```

Special Case: `for` Loops and Variable Scope (cont.)

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

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

scope of i

Special Case: `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 Crimson!");
    }
}
```

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

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

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

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

- What's the initial pseudocode for this task?

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

```
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

```
 2*2   2*2
   ()
  (())        2*2
 ((()))
(((())))
========
   4*2
|::::::|      2
 |::::|
```

- 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();
    }
}
```

```
 2*2  2*2
   ()
  (())      2*2
 ((()))
(((())))
```

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

- Pseudocode for default size:

- Java code for default size:

```
public static void drawHandle() {



}
```

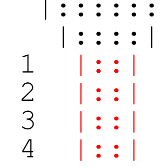
```
    ()
   (())
  ((()))
 (((())))
========
 |::::::|
  |::::|
1  |::|
2  |::|
3  |::|
4  |::|
   +==+
```

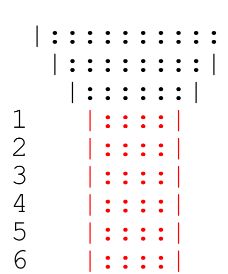
Practice: Making the Handle Scaleable

- We again compare two different sizes.

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

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





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();
}
```

Methods with Parameters and Return Values

Boston 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();
    }
}
```

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 in Java:
  - begins at the point at which it is declared
  - ends at the end of the innermost block that encloses the declaration

```
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);
}
```

scope of sum

scope of avg

Special Case: Parameters and Variable Scope

- What about the parameters of a method?
  - they do *not* follow the default scope rules!
  - their scope is limited to their method

```
public class MyClass {
    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);
    }

    static int c = a + b;     // does not compile!
}
```

scope of a and b

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 returnType 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`
- For other examples, use the Java API on the Resources page.

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:
  ClassName.*methodName*(*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;
}
```

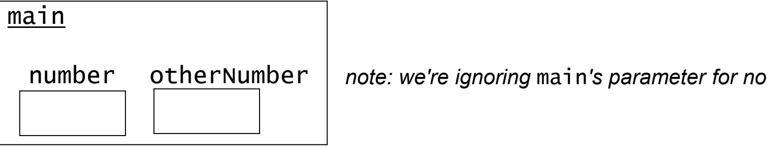
- Here's some code that uses it correctly:

```
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.)

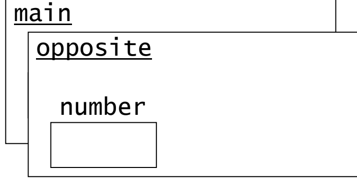
- When you make a method call, the Java runtime sets aside a block of memory known as the *frame* of that method call.



- 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.

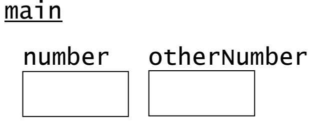
Keeping Track of Variables (cont.)

- 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;
    }
}
```

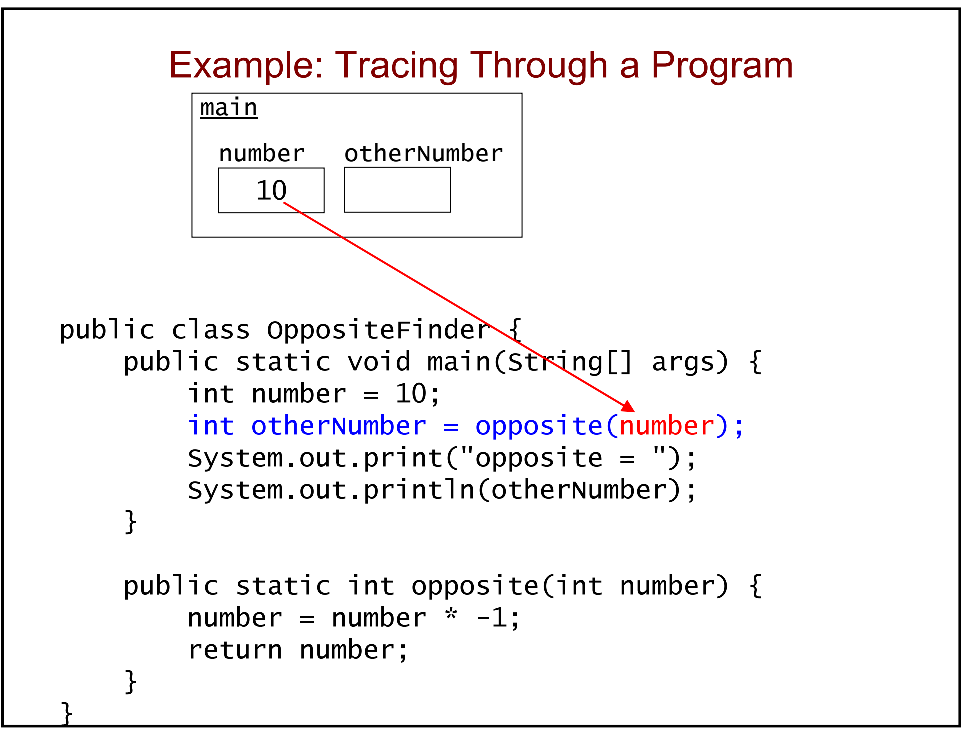
Example: Tracing Through a Program

main

number otherNumber

10

```
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

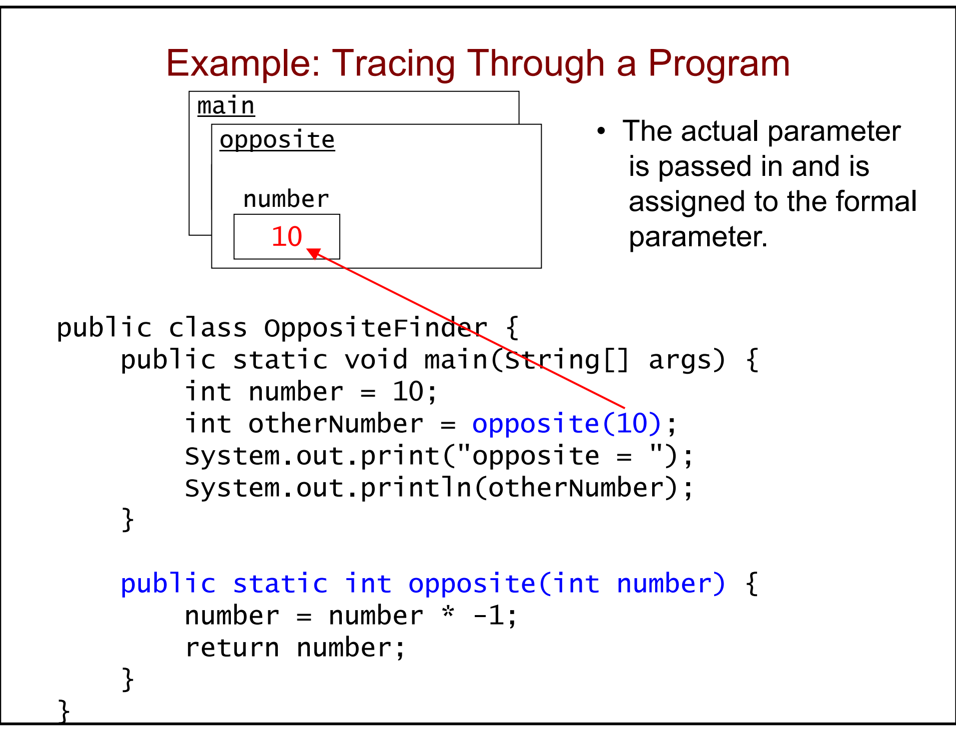
main

opposite

number

- 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

main

opposite

number

10

```
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;
    }
}
```

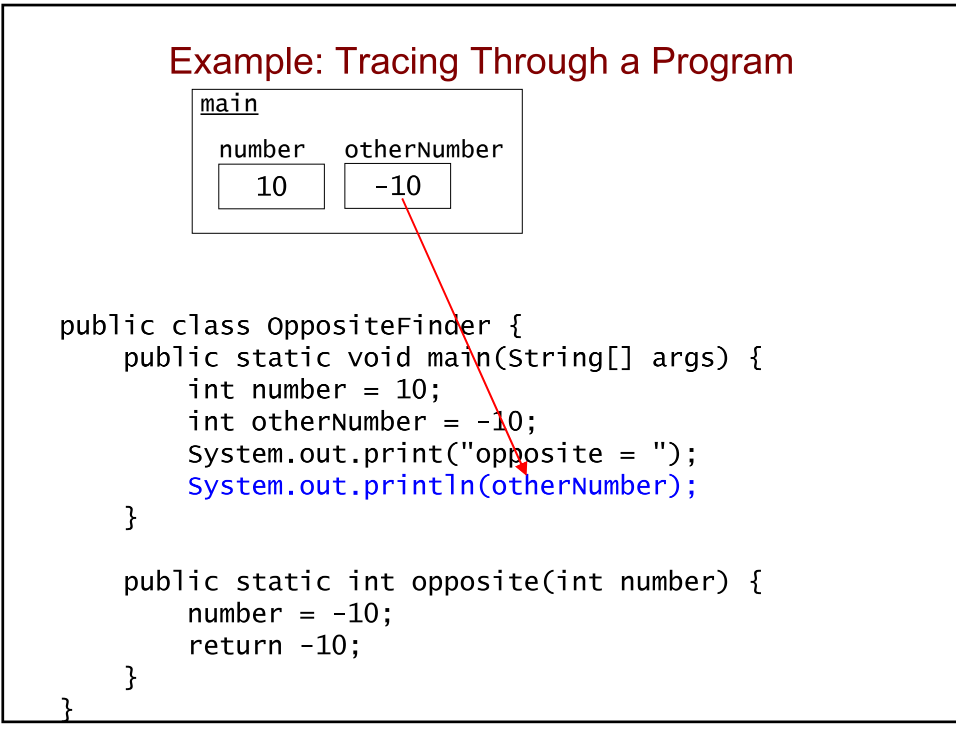
Example: Tracing Through a Program

- 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


```
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(_______________________) {




    }
}
```

Using Objects from Existing Classes

Boston University

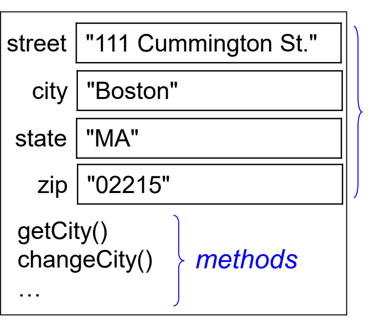
David G. Sullivan, Ph.D.

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"

Classes as Blueprints

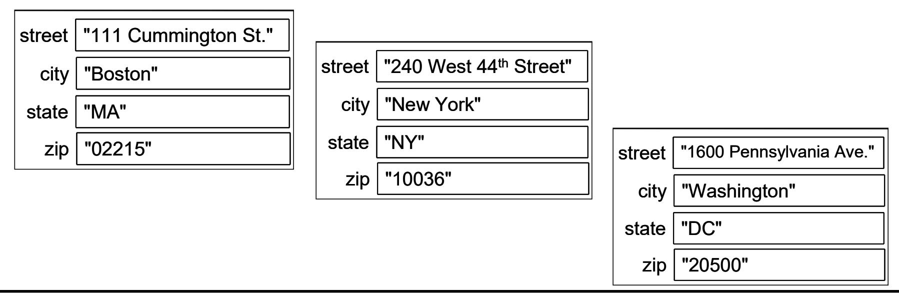
- We've been using classes as containers for our programs.
- A class can also serve as a blueprint – as the definition of a new type of object.
- The objects of a given class are built according to its blueprint.
- Another analogy:
  - class = cookie cutter
    objects = cookies
- The objects of a class 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:



Using Objects from Existing Classes

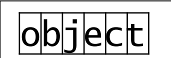
- Later in the course, you'll learn how to create your own classes that act as blueprints for objects.
- For now, we'll focus on learning how to use objects from existing classes.

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) → `String`; *variable name* → `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 `String`s 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.

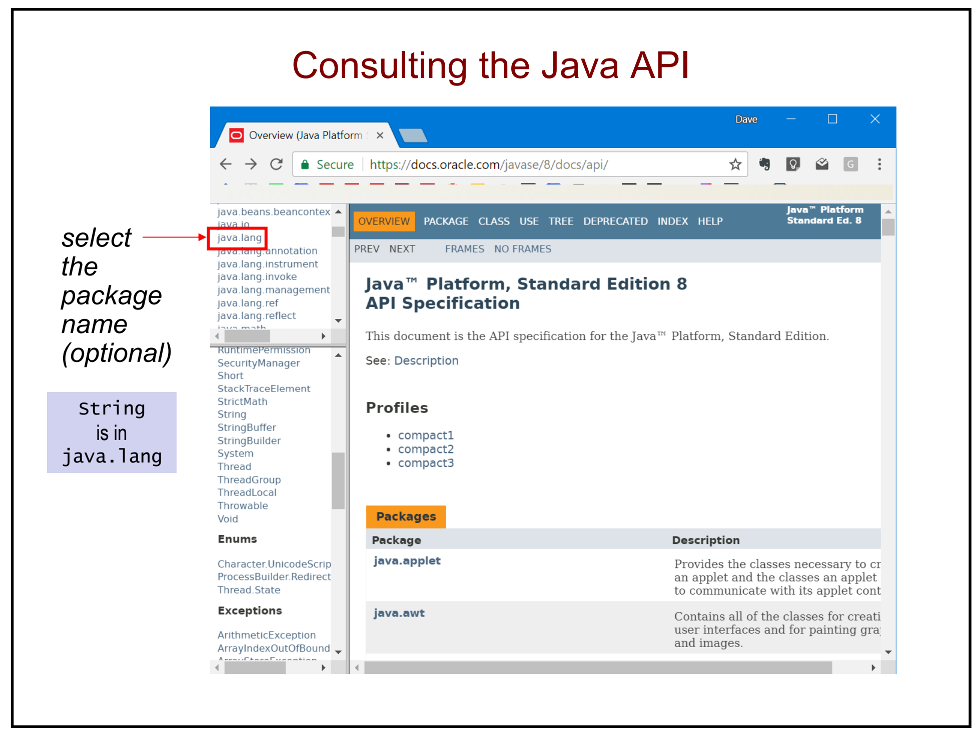
The API of a Class

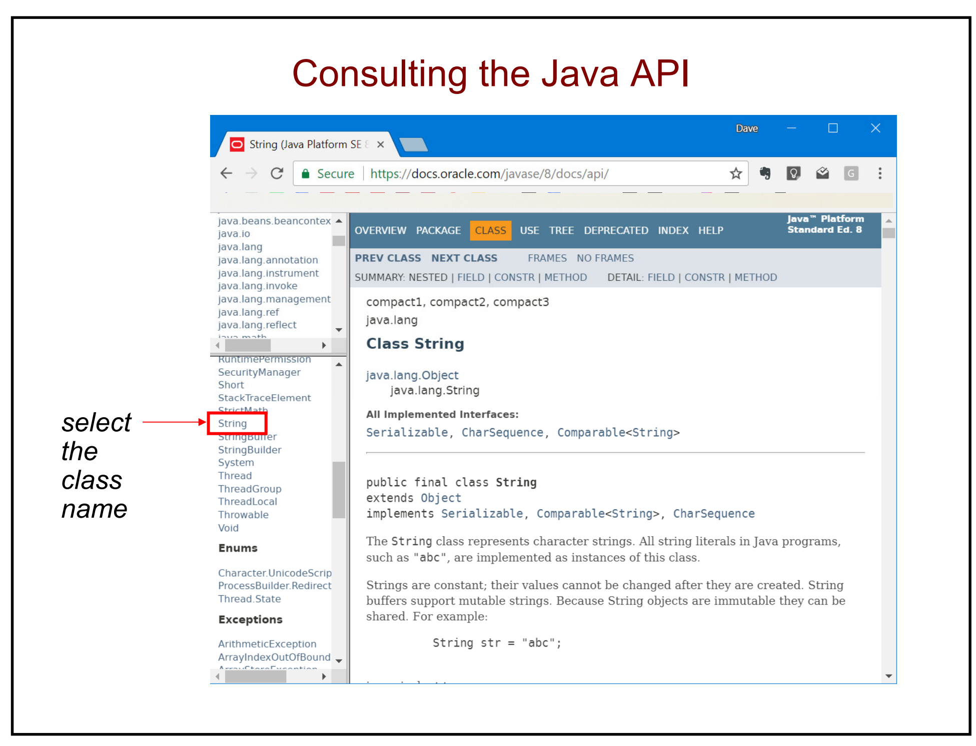
- The methods defined within a class are known as the *API* of that class.
  - API = application programming interface
- We can consult the API of an existing class to determine which operations are supported.
- The API of all classes that come with Java is available here:
  `https://docs.oracle.com/javase/8/docs/api/`
  - there's a link on the resources page of the course website

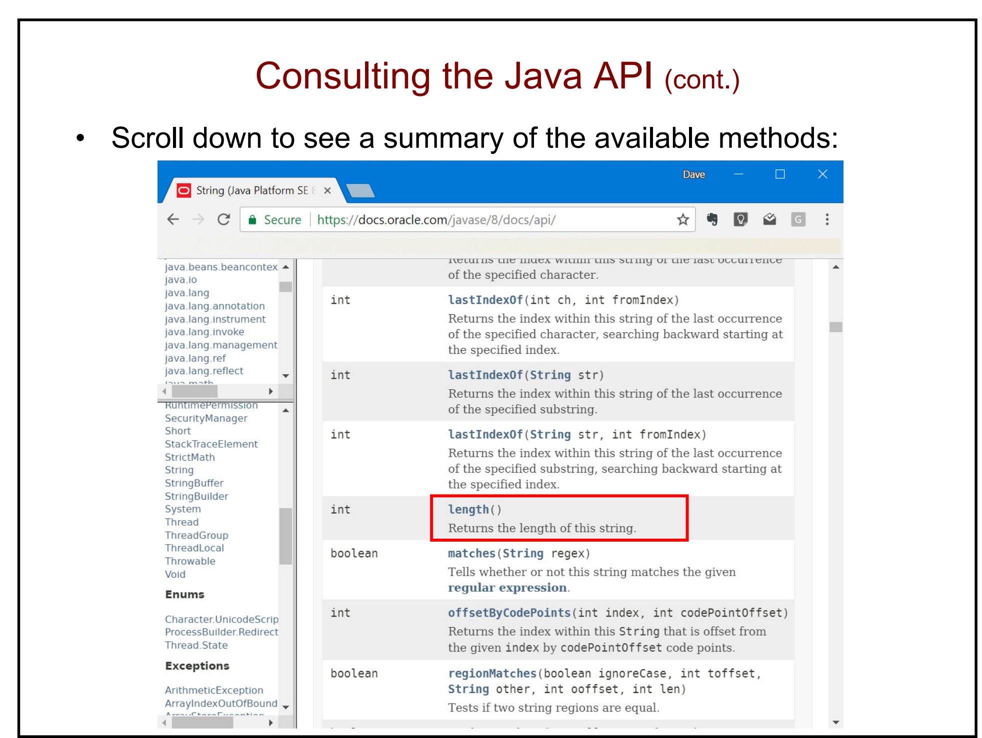
Consulting the Java API

select the package name (optional)

`String` is in `java.lang`

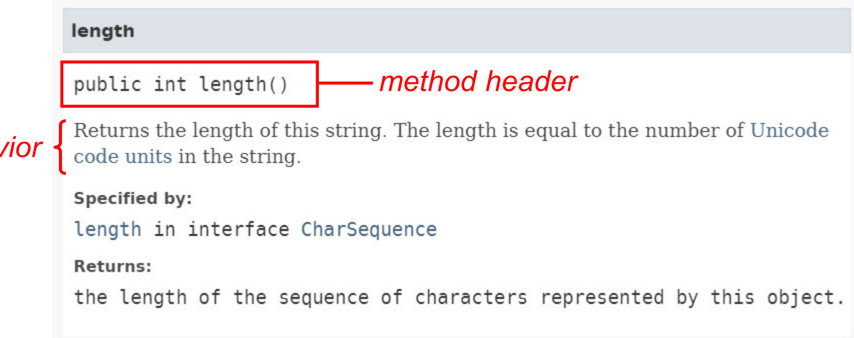



Consulting the Java API
select the class name
String (Java Platform SE
https://docs.oracle.com/javase/8/docs/api/
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Class String
java.lang.Object
java.lang.String
All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>
public final 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";


Consulting the Java API (cont.)
• Scroll down to see a summary of the available methods:
int lastIndexOf(int ch, int fromIndex)
Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int lastIndexOf(String str)
Returns the index within this string of the last occurrence of the specified substring.
int lastIndexOf(String str, int fromIndex)
Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int length()
Returns the length of this string.
boolean matches(String regex)
Tells whether or not this string matches the given regular expression.
int offsetByCodePoints(int index, int codePointOffset)
Returns the index within this String that is offset from the given index by codePointOffset code points.
boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
Tests if two string regions are equal.

Consulting the Java API (cont.)

- Clicking on a method name gives you more information:



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

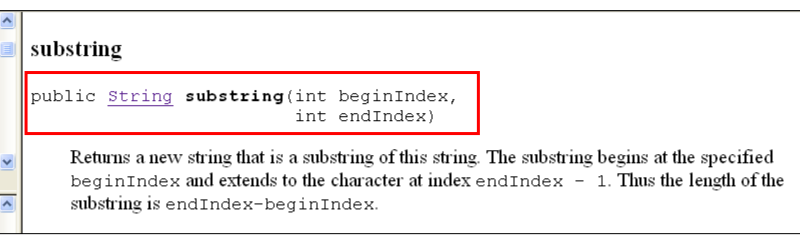
Numbering the Characters in a `String`

- 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`.

`String substring(int beginIndex, int endIndex)`

- 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";
     ```

How should we fill in the blank?

charAt

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`.

```
String s = "Strings have methods inside them!";
int len = s.length();
__________________  // get the last character in s
```

charAt Method

charAt

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from 0 to length() - 1.

- The charAt() method that we use for indexing returns a char, not a String.
- We have to be careful when we use its return value!
  - example: what does this print?

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

charAt Method

- Here's how we can fix this:

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

⬇

```
System.out.println('P' + "" +
   'S');
```

⬇

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

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.

`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 (applies to `substring(int beginIndex, int endIndex)`)

- 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 on **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?

```
import java.util.*;

public class LengthConverter {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        int cm = (int)(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 inches;
    }
}
```

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

Conditional Execution

Boston University

David G. Sullivan, Ph.D.

Review: Simple Conditional Execution in Java

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

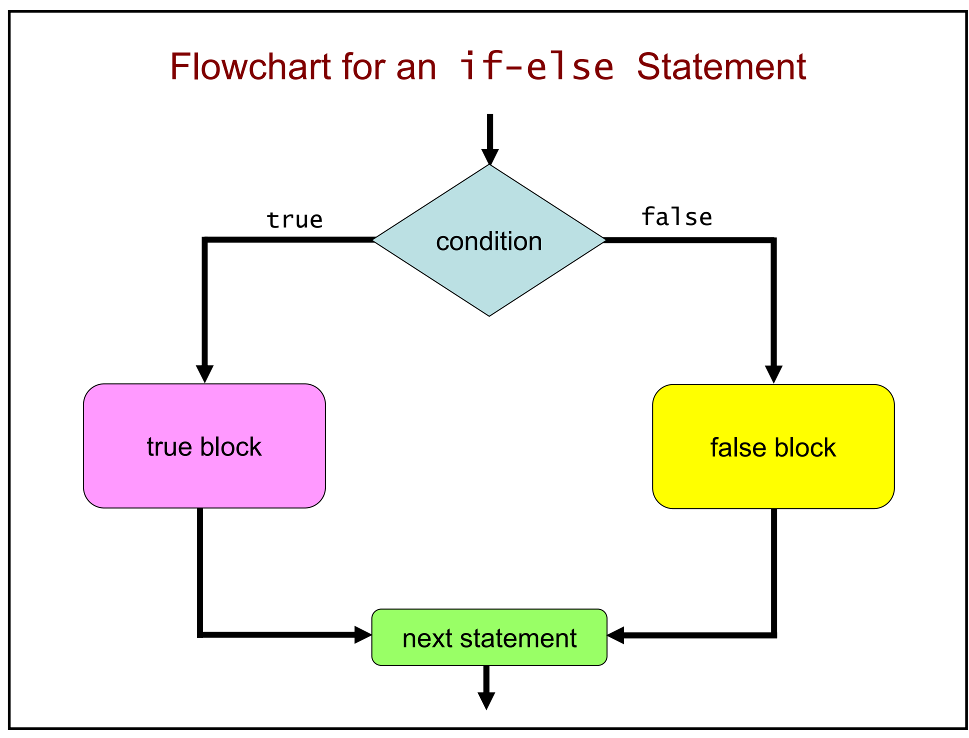
```
if (condition) {
    true block
}
```

- 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.

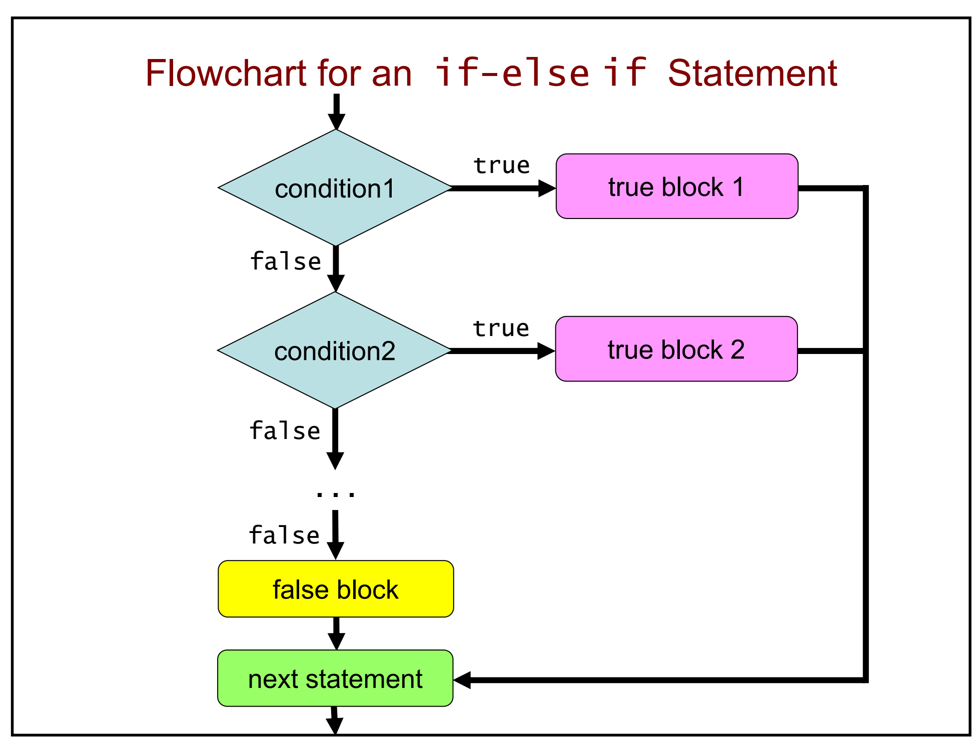
`if-else if` Statements

- 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(_____________________) {




}
```

Indefinite Loops and Boolean Expressions

Boston 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

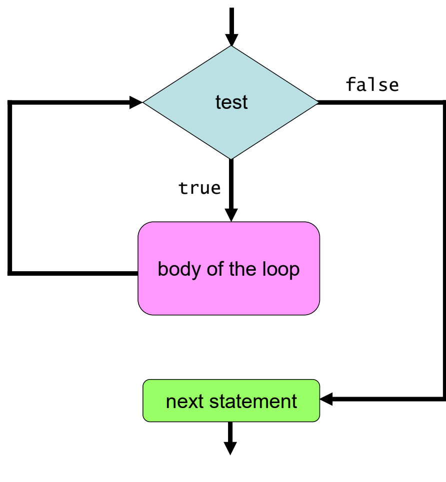
```
while (test) {
    one or more statements
}
```

- 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`:

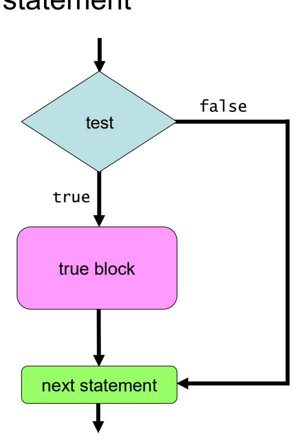
```
int mult = num;
while (mult < 100) {
    System.out.print(mult + " ");
    mult = mult + num;
}
```

	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

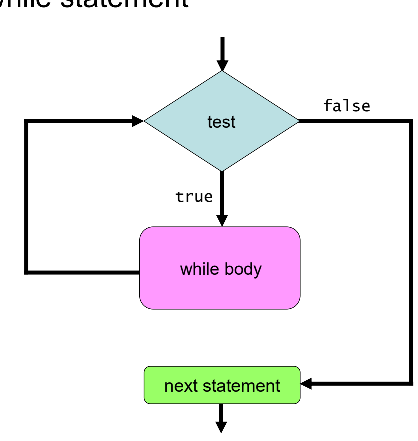
and now `(mult < 100)` is `false`, so we exit the loop

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);
```

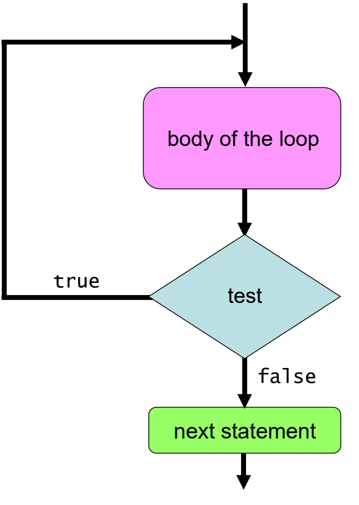
- Note the need for a semi-colon after the condition.
- We do *not* need a semi-colon after the condition in a `while` loop.
  - beware of using one – it can actually create an infinite loop!

Evaluating a `do-while` Loop

Steps:

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

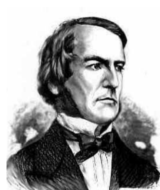
- What does the following loop output?

```
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) {
    ...
```

Arrays

Boston 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*

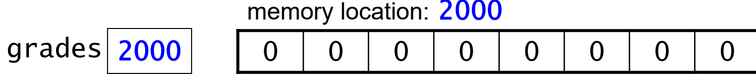
Arrays and References

- 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:

memory location: 2000



- We usually use an arrow to represent a reference:

grades → | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

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.

What is the output of the full program?

```
import java.util.*;

public class FunWithArrays {
    public static void main(String[] args) {
        int[] temps = {51, 50, 36, 29, 30};
        int first = temps[0];
        int numTemps = temps.length;
        int last = temps[numTemps - 1];

        temps[2] = 40;
        temps[3] += 5;
        System.out.println(temps[3]);
        System.out.println(Arrays.toString(temps));
    }
}
```

`temps`

`first`

`numTemps`

`last`

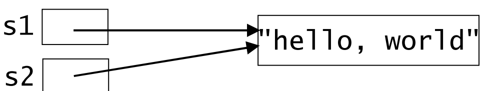
output:

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:

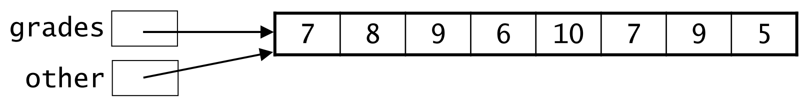
```
String s1 = "hello, world";
String s2 = s1;
```



Copying References (cont.)

- An example involving an array:

```
int[] grades = {7, 8, 9, 6, 10, 7, 9, 5};
int[] other = grades;
```



- Given the lines of code above, what will the lines below print?

```
other[2] = 4;
System.out.println(grades[2] + " " + other[2]);
```

Changing the Internals vs. Changing a Variable

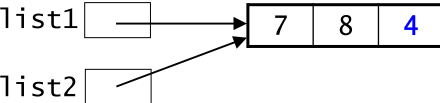
- When two variables hold a reference to the same array...

```
int[] list1 = {7, 8, 9};
int[] list2 = list1;
```

list1 → 7 | 8 | 9
list2 → (same array)

- ...if we change *the internals* of the array, both variables will "see" the change:

```
list2[2] = 4;
System.out.println(Arrays.toString(list1));
```



output of println:

Changing the Internals vs. Changing a Variable (cont.)

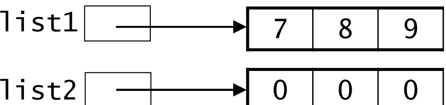
- When two variables hold a reference to the same array...

```
int[] list1 = {7, 8, 9};
int[] list2 = list1;
```

list1 → 7 | 8 | 9
list2 → (same array)

- ...if we change one of the variables *itself*, that does *not* change the other variable:

```
list2 = new int[3];
System.out.println(Arrays.toString(list1));
```



output of println:

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`

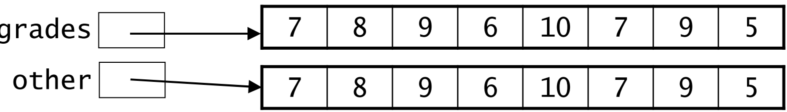
- Attempting to use a null reference to access an object produces a `NullPointerException`.
  - "pointer" is another name for reference
  - examples:

```
int[] grades = null;
String s = 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;
}
```

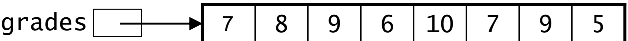
Testing If An Array Meets Some Condition (cont.)

- Here's a similar problem: write a method that determines if all of the grades are perfect (assume perfect = 100).

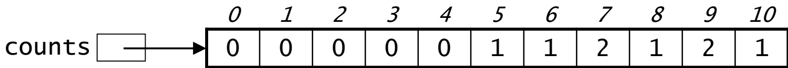
```
public static boolean allPerfect(int[] grades) {









}
```

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



we would have this array of counts:



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

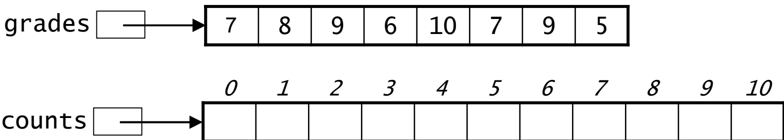
- The size of the `counts` array should be one more than the maximum value being counted:

```
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.)



- 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

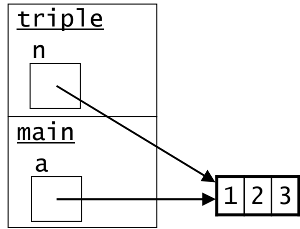
- We can write a method to create and return the array of counts:

```
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);
    ...
}
```

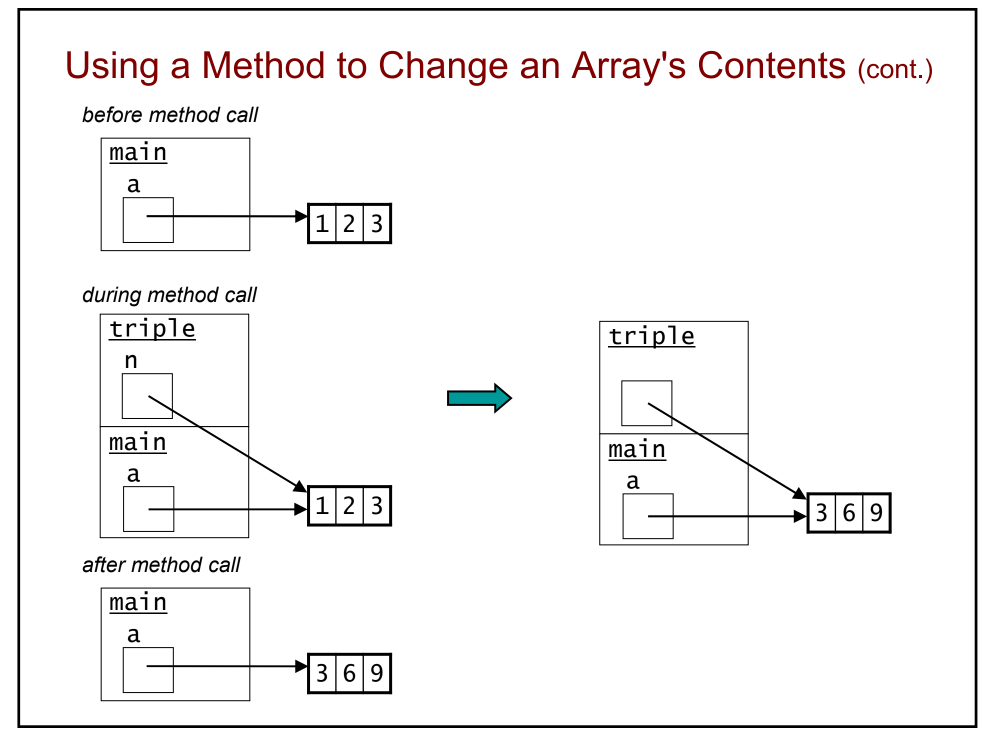
Using a Method to Change an Array's Contents

```
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 copy of the reference, *not* a copy of the array.
- If the method changes the internals of the array, those changes will be there after the method returns.



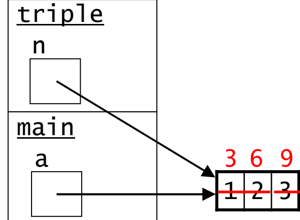
Using a Method to Change an Array's Contents (cont.)



Changing the Internals vs. Changing a Variable

```
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;    // changes internals
    }
}
```

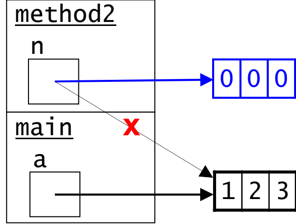
- If the method changes the *internals* of the array, those changes will be there after the method returns.



Changing the Internals vs. Changing a Variable (cont.)

```
public static void main(String[] args) {
    int[] a = {1, 2, 3};
    triple(a);
    System.out.println(Arrays.toString(a));
}

public static void method2(int[] n) {
    n = new int[3];     // changes the variable
}
```

- However, if the method changes its *variable* for the array, that change does *not* affect the original array.
- Changing what's in one variable doesn't affect any other variable!



Swapping Elements in an Array

- We sometimes need to be able to swap two elements in an array.
- Example:

arr

0	1	2	3	4	5	6	7
35	6	19	23	3	47	9	15

arr

35	6	*47*	23	3	*19*	9	15

- What's wrong with this code for swapping the two values?

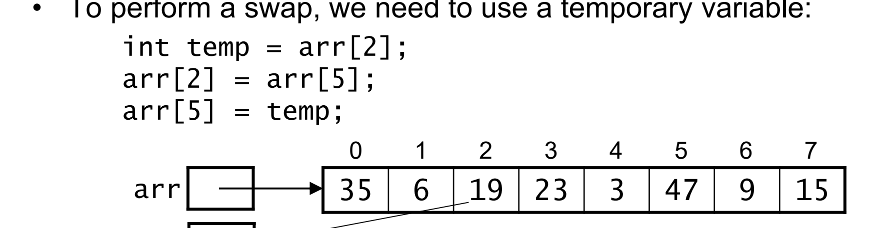
```
arr[2] = arr[5];
arr[5] = arr[2];
```

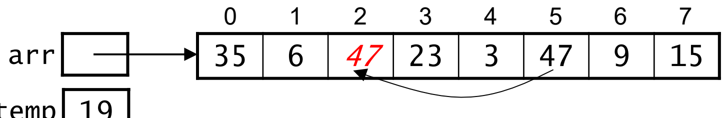
  - it gives this:

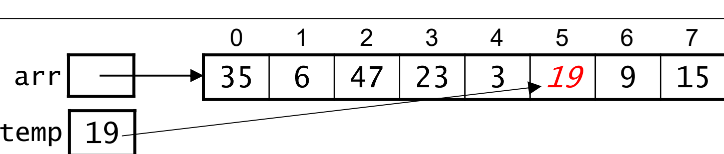
Swapping Elements in an Array (cont.)

- To perform a swap, we need to use a temporary variable:

```
int temp = arr[2];
arr[2] = arr[5];
arr[5] = temp;
```







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 internals of 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?

Recall: 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);
    ...
}
```

An Alternative Approach for the Array of Counts

- Create the array ahead of time and pass it into the method:

```
public static void getCounts(int[] grades, int[] counts) {

    for (int i = 0; i < grades.length; i++) {
        counts[grades[i]]++;
    }


}

public static void main(String[] args) {
    ... // main method begins as in the earlier versions
    int max = maxGrade(grades);
    int[] counts = new int[max];
    getCounts(grades, counts);
    ...
}
```

- Because the method changes the internals of the array, those changes will be there after the method returns.

Shifting Values in an Array

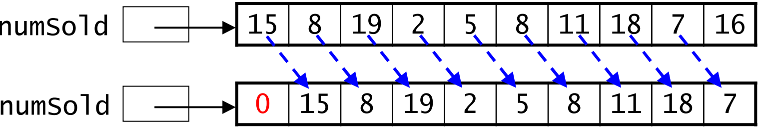
- Let's say a small business is using an array to store the number of items sold over a 10-day period.

`numSold` → | 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.



  - 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

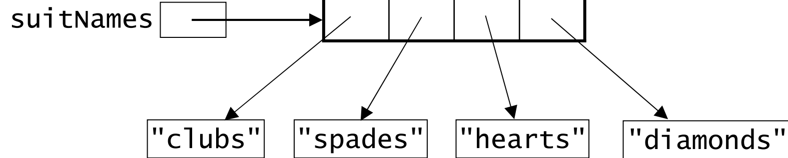
- Once we have created an array, we can't increase its size.
- Instead, we need to do the following:
  - create a new, larger array (use a temporary variable)
  - copy the contents of the original array into the new array
  - assign the new array to the original array variable
- Example for our `grades` 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"};
```



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];
```

number of rows ↗ ↖ number of columns

- 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:

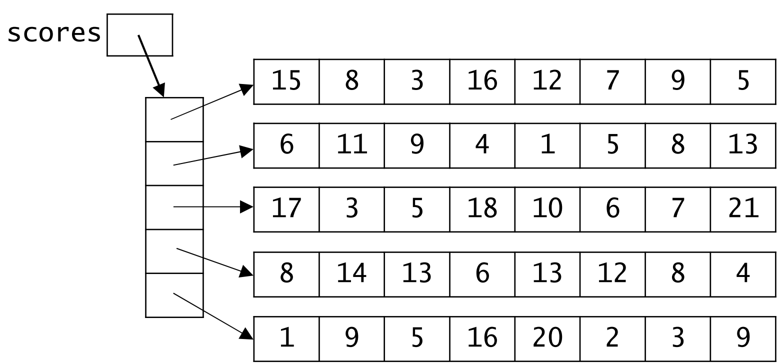
```
char[][] board = {{' ', ' ', ' '},
                  {' ', ' ', ' '},
                  {' ', ' ', ' '}};
```

- 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!



- `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}};
```


- We can also create arrays of higher dimensions.
  - example: a three-dimensional matrix:

```
double[][][] matrix = new double[2][5][4];
```

Classes as Blueprints: How to Define New Types of Objects

Boston 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();
```

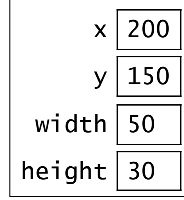
- Using an instance method is like sending a message to an object, asking it to perform an operation.
- We refer to the object on which the method is invoked as either:
  - the *called object*
  - the *current object*

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:



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 {
    ...
```

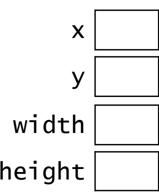
- As always, the class definition goes in an appropriately named text file.
  - in this case: Rectangle.java

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



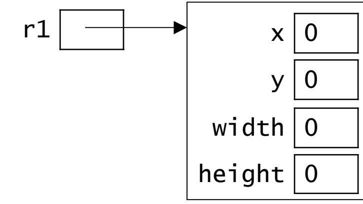
- Another name for a field is an *instance variable.*

Using Fields to Capture an Object's State (cont.)

- For now, we'll create `Rectangle` objects like this:

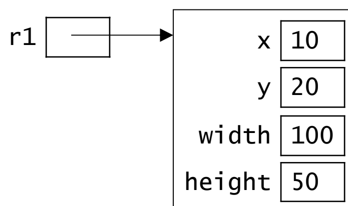
```
Rectangle r1 = new Rectangle();
```

- 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

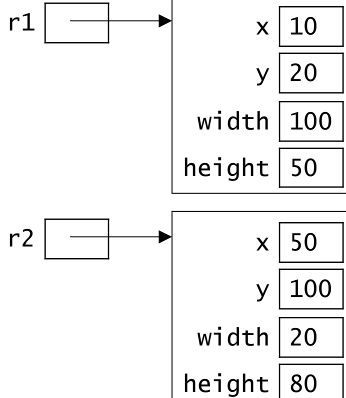
- Here's an instance method for getting the area of a `Rectangle`:

```
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
- Examples of accessors:
  - `area()` in our `Rectangle` class
  - `String` methods: `length()`, `substring()`, `charAt()`

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;
    }
}
```

Which method call increases r's height by 5?

```
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;
    }
}
```

- Consider this client code:

```
Rectangle r = new Rectangle();
r.width = 10;
r.height = 15;
______???______;
```

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.

```
public ______ moveRight(___________________) {


}
```

- Add an accessor method that determines if the rectangle is a square (true or false).

```
public __________ isSquare(____________) {



}
```

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 ClassName(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

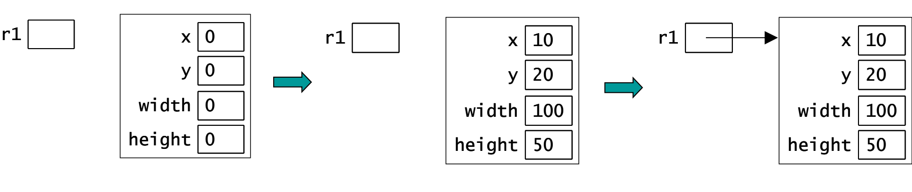
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        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 Only 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;
}
```

- Throwing an exception ends the method early.
- If the caller of the method doesn't handle the exception, it will crash.

Allowing Only 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);
}
```

  - we use `this` to call another method in the same object
  - those other methods perform the necessary error-checking

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.
- 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

```
public _________ scale(___________________) {


}
```

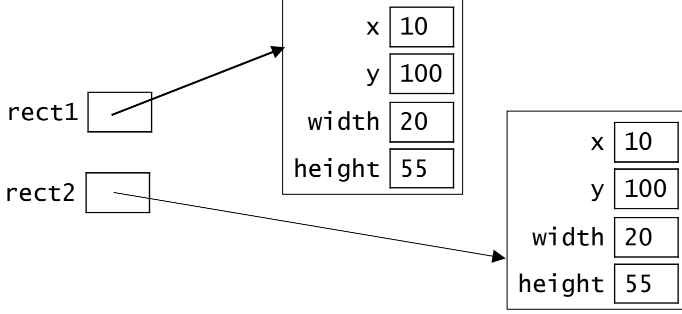
- Add an accessor method that gets the perimeter of a `Rectangle` object.

```
public _________ perimeter(___________________) {


}
```

Testing for Equivalent Objects

- Let's say that we have two different `Rectangle` objects, both of which represent equivalent rectangles:

```
Rectangle rect1 = new Rectangle(10, 100, 20, 55);
Rectangle rect2 = new Rectangle(10, 100, 20, 55);
```



- What is the value of the following condition?

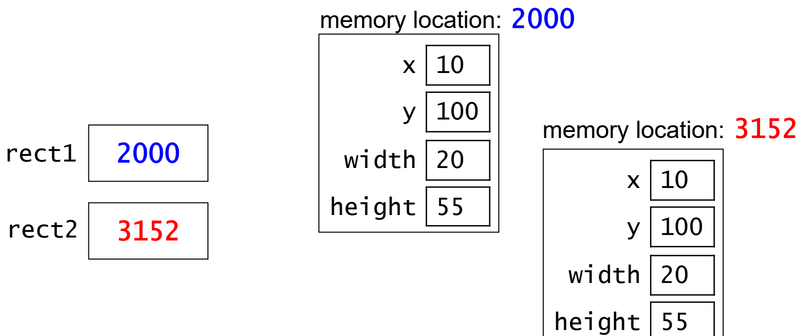
```
rect1 == rect2
```

Testing for Equivalent Objects (cont.)

- The condition

```
rect1 == rect2
```

compares the *references* stored in `rect1` and `rect2`.



- 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

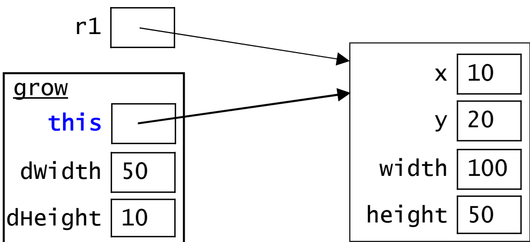
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);

        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)`



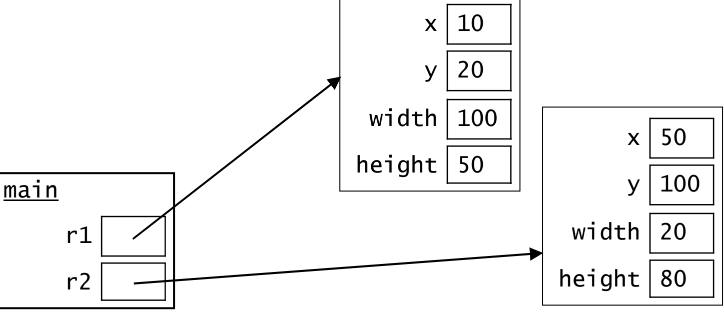
- The method uses `this` to access the fields in the called object.
  - even if the code doesn't explicitly use it

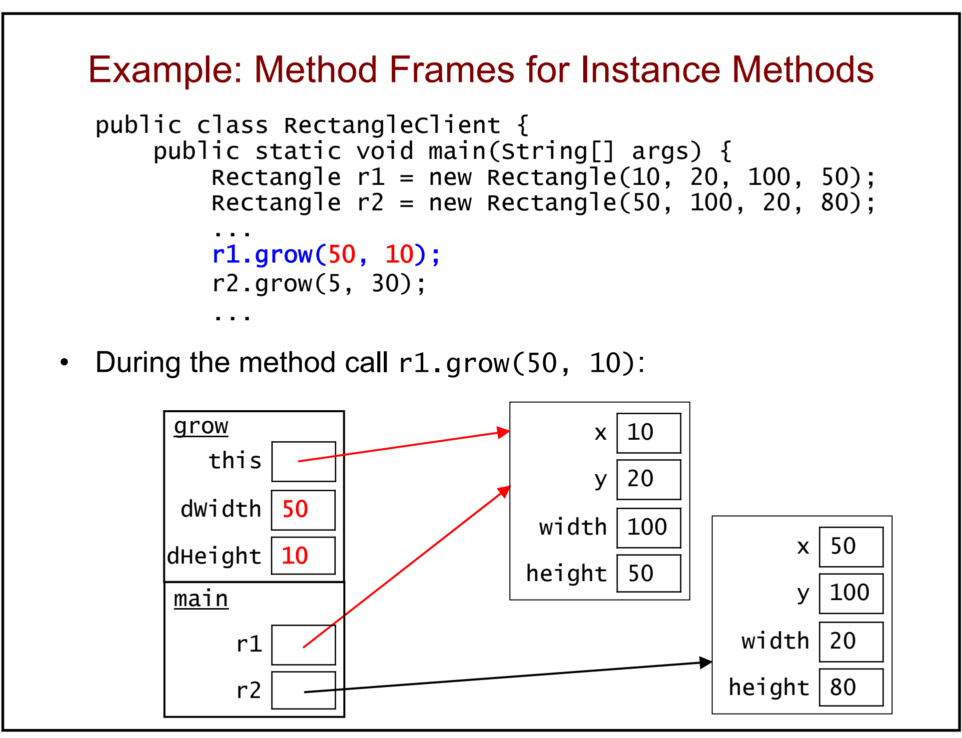
```
width += dWidth;            this.width += dWidth;
height += dHeight;          this.height += dHeight;
```

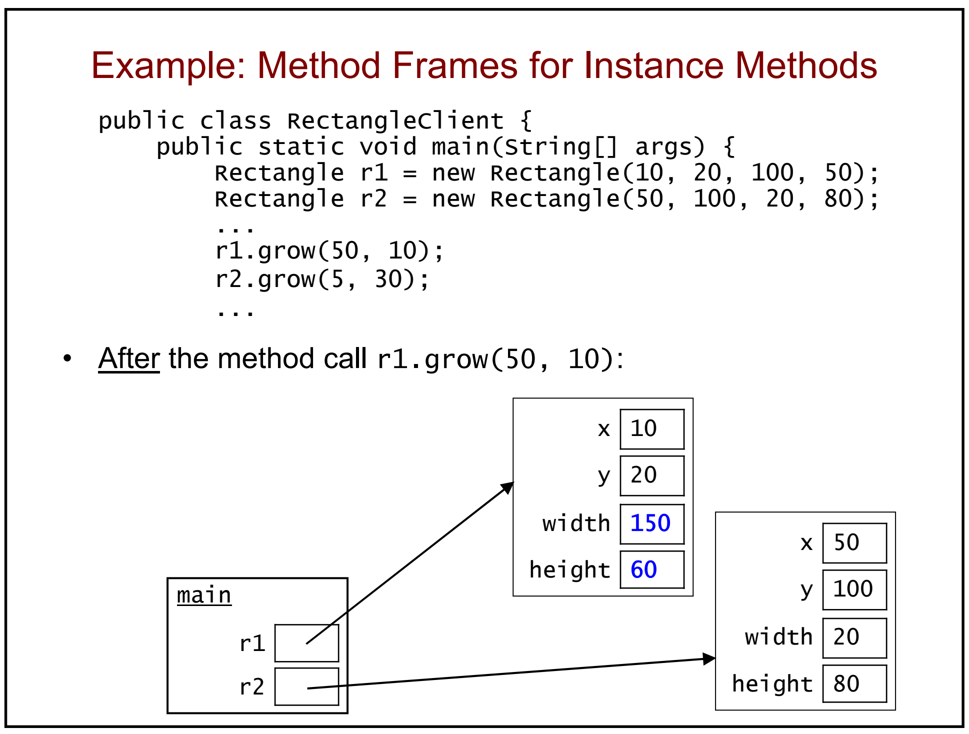
Example: Method Frames for Instance Methods

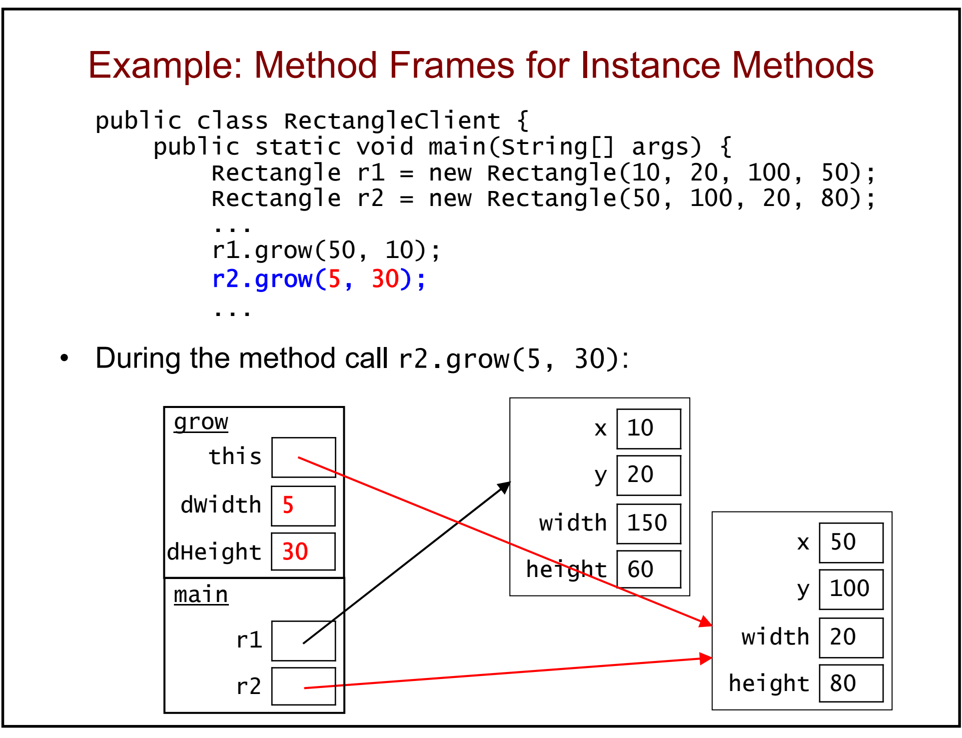
```
public class RectangleClient {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle(10, 20, 100, 50);
        Rectangle r2 = new Rectangle(50, 100, 20, 80);
        ...
        r1.grow(50, 10);
        r2.grow(5, 30);
        ...
```

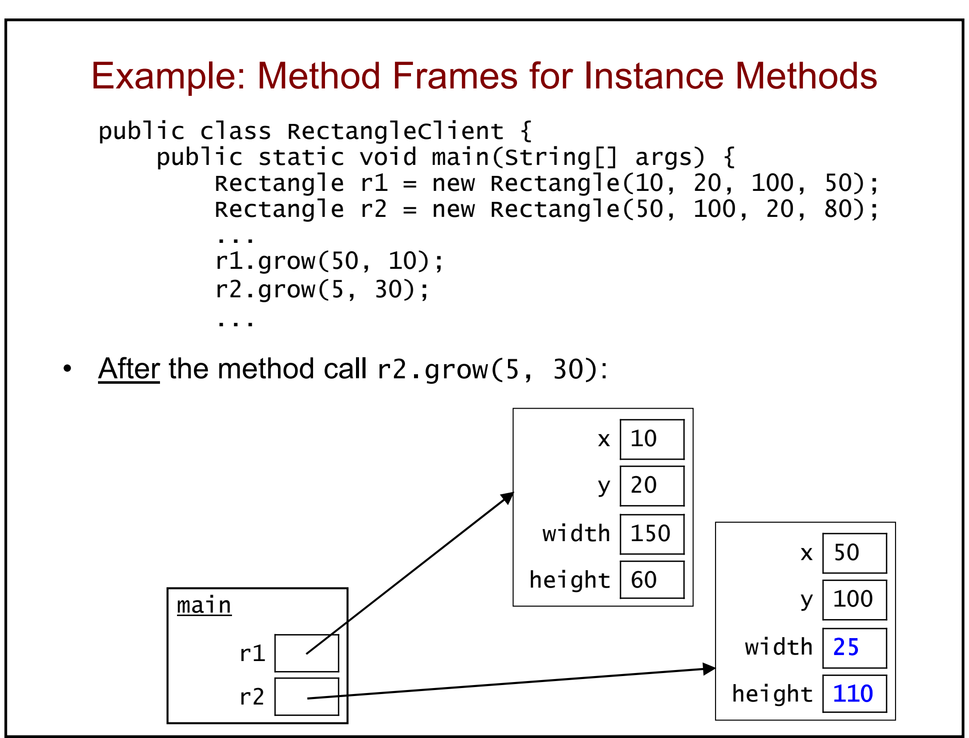
- After the objects are created:




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):
grow
this
dWidth
50
dHeight
10
main
r1
r2
x
10
y
20
width
100
height
50
x
50
y
100
width
20
height
80

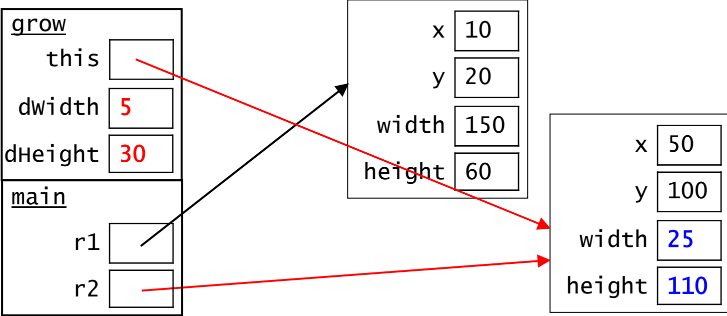

Example: Method Frames for Instance Methods
public class RectangleClient {
public static void main(String[] args) {
Rectangle r1 = new Rectangle(10, 20, 100, 50);
Rectangle r2 = new Rectangle(50, 100, 20, 80);
...
r1.grow(50, 10);
r2.grow(5, 30);
...
After the method call r1.grow(50, 10):
main
r1
r2
x
10
y
20
width
150
height
60
x
50
y
100
width
20
height
80


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):
grow
this
dWidth
5
dHeight
30
main
r1
r2
x
10
y
20
width
150
height
60
x
50
y
100
width
20
height
80


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):
main
r1
r2
x
10
y
20
width
150
height
60
x
50
y
100
width
25
height
110

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 x;

    public static int bar(int b, int c, Foo f) {
        c = c + this.x;          // would not compile
        return 3*b + f.x;        // would compile
    }

    public int boo(int d, Foo f) {
        d = d + this.x + f.x;    // 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 `r`) and an integer (call it `delta`) 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 `delta`
  - prints the new dimensions and area

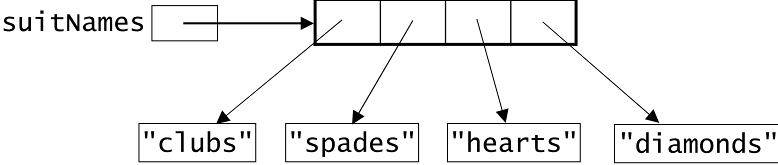
Collections of Data

- 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)
```

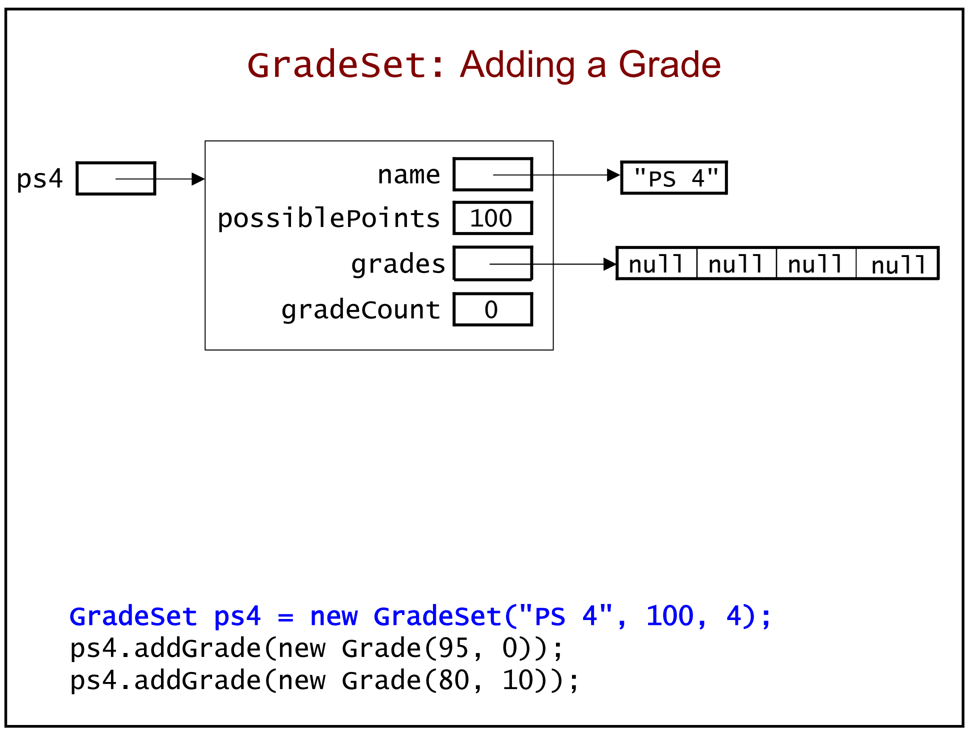
- Accessor methods:
```
String getName()
int getPossiblePoints()
int getGradeCount()
Grade getGrade(int i) // get grade at position i
double averageGrade(boolean includePenalty)
```

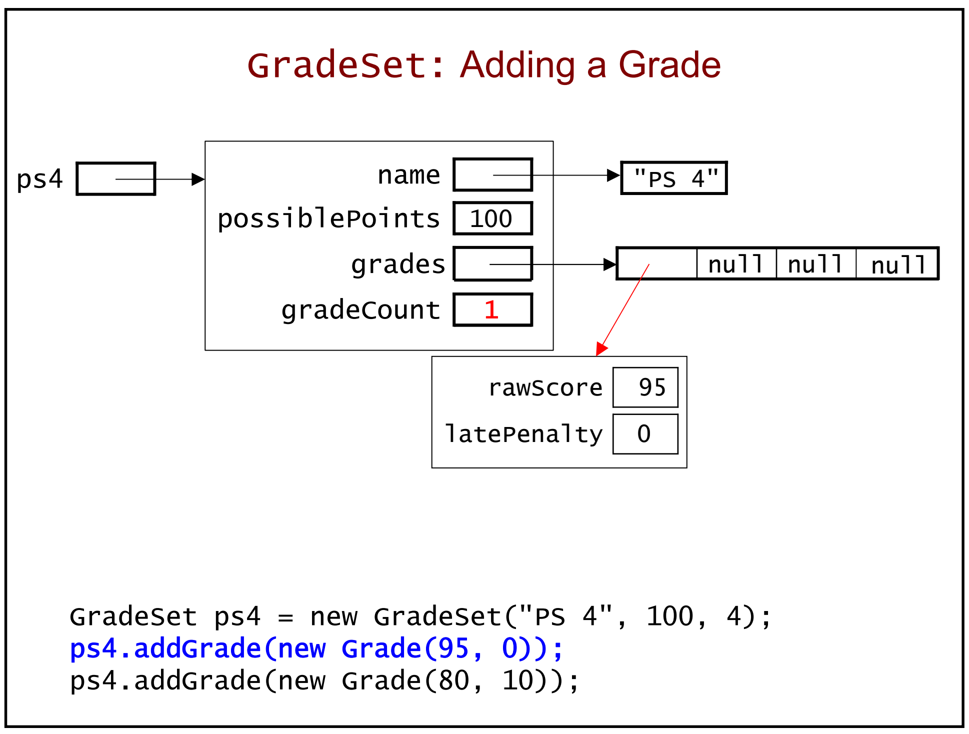
- Mutator methods:
```
void setName(String name)
void setPossiblePoints(int possPoints)
void addGrade(Grade g)
Grade removeGrade(int i) // remove grade at posn i
```

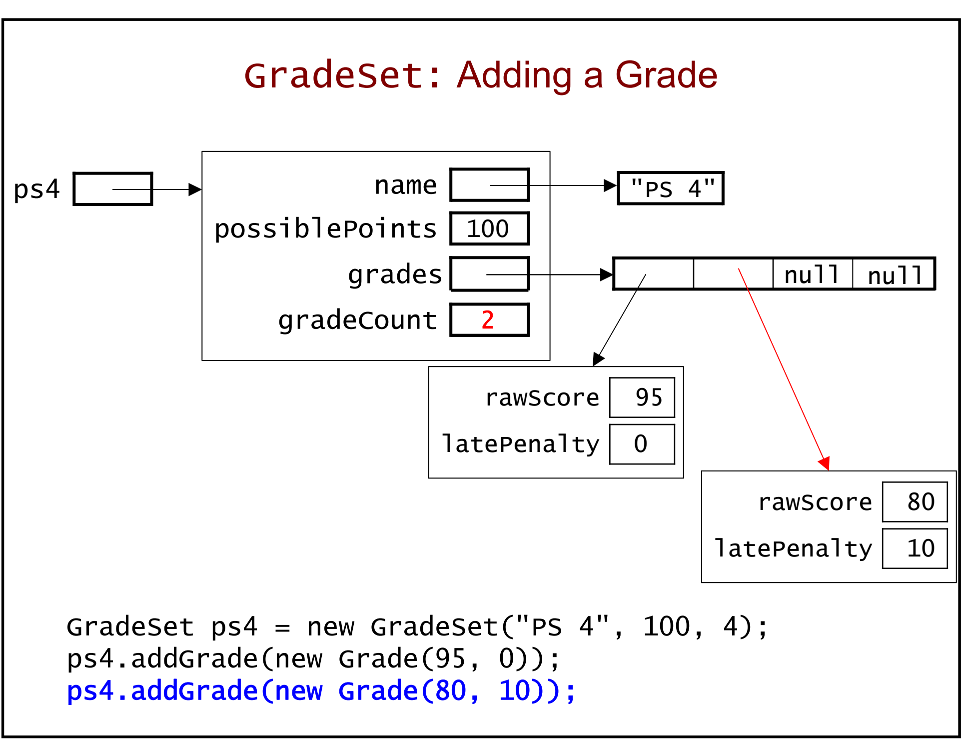
- Let's review the code for these, and write some of them together.

GradeSet Constructor/Methods

GradeSet Constructor/Methods


GradeSet: Adding a Grade
ps4
name
"PS 4"
possiblePoints
100
grades
null
null
null
null
gradeCount
0
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));


GradeSet: Adding a Grade
ps4
name
"PS 4"
possiblePoints
100
grades
null
null
null
gradeCount
1
rawScore
95
latePenalty
0
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));


GradeSet: Adding a Grade
ps4
name
"PS 4"
possiblePoints
100
grades
null
null
gradeCount
2
rawScore
95
latePenalty
0
rawScore
80
latePenalty
10
GradeSet ps4 = new GradeSet("PS 4", 100, 4);
ps4.addGrade(new Grade(95, 0));
ps4.addGrade(new Grade(80, 10));

Inheritance and Polymorphism

Boston 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() {
        String 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 class 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

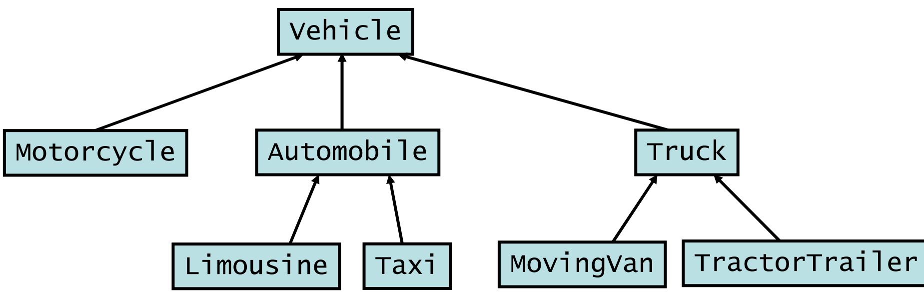
```
public class Vehicle {
    private String make;
    private String model;
    private int year;
    private int mileage;
    private String plateNumber;
    private int numWheels;  // this was not in Automobile

    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

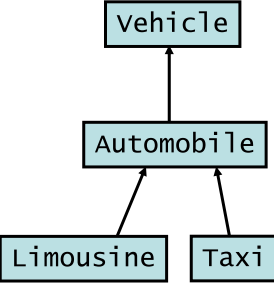
- Inheritance leads classes to be organized in a *hierarchy*:



- 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;
}
```

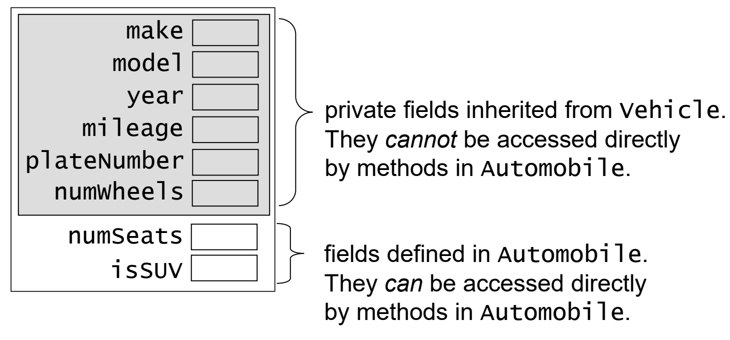
  - it will be inherited by subclasses of `Truck`
  - it won't be available to non-truck subclasses of `Vehicle`

- 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:



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.)

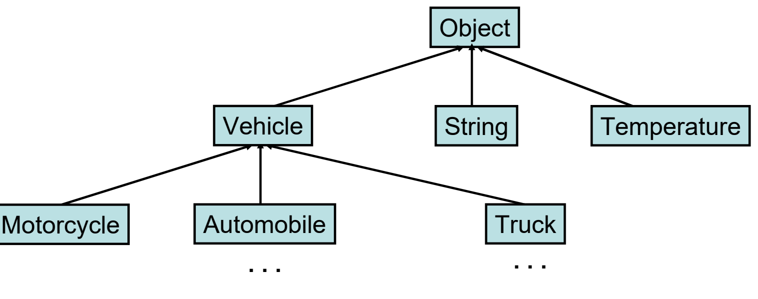
- If a subclass constructor doesn't explicitly invoke a superclass constructor, the compiler tries to insert a call to the superclass constructor with no parameters.
- If there isn't such a constructor, we get a compile-time error.
  - example: this constructor won't compile:

```
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;
}
```

Another Example of Inheritance

- A square is a special type of rectangle.
  - but the width and height must be the same
- Assume that we also want `Square` objects to have a field for the unit of measurement (e.g., `"cm"`).
- `Square` objects should mostly behave like `Rectangle` objects:

```
Rectangle r = new Rectangle(20, 30);
int area1 = r.area();

Square sq = new Square(40, "cm");
int area2 = sq.area();
```

- But there may be differences as well:

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

output:
`20 x 30`

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

output:
`square with 40-cm sides`

Another Example of Inheritance (cont.)

```
public class Rectangle {
   private int width;
   private int height;
   ...
   public Rectangle(int initWidth, int initHeight) {
       ...
   }
   public int getWidth() {
       ...
   }
   ... // other methods
}
```

```
public class Square extends Rectangle {
   private String unit; // inherits other fields
   public Square(int side, String unit) {
       super(side, side);
       this.unit = unit;
   }
   public String toString() {  // overrides
       String s = "square with ";
       s += this.getWidth() + "-";
       s += this.unit + " sides";
       return s;
   } // inherits other methods
}
```

Another Example of Inheritance (cont.)

```
public class Rectangle {
   private int width;
   private int height;
   ...
   public Rectangle(int initWidth, int initHeight) {
       ...
   }
   public int getWidth() {
       ...
   }
   ... // other methods
}
```

```
public class Square extends Rectangle {
   private String unit; // inherits other fields
   public Square(int side, String unit) {
       super(side, side);
       this.unit = unit;
   }
   public String toString() {  // overrides
       String s = "square with ";
       s += this.getWidth() + "-";
       s += this.unit + " sides";
       return s;
   } // inherits other methods
}
```

Another Example of Method Overriding

- The `Rectangle` class has the following mutator method:

```
public void setWidth(int w) {
    if (w <= 0) {
        throw new IllegalArgumentException();
    }
    this.width = w;
}
```

- The `Square` class inherits it. Why should we override it?

Which of these works?

A.

```
// Square version, which overrides
// the version inherited from Rectangle
public void setWidth(int w) {
    this.width = w;
    this.height = w;
}
```

B.

```
// Square version, which overrides
// the version inherited from Rectangle
public void setWidth(int w) {
    this.setWidth(w);
    this.setHeight(w);
}
```

C. either version would work

D. neither version would work

Accessing Methods from the Superclass

- The solution: use `super` to access the inherited version of the method – the one we are overriding:

```
// Square version
public void setWidth(int w) {
    super.setWidth(w);  // call the Rectangle version
    super.setHeight(w);
}
```

- Only use `super` if you want to call a method from the superclass *that has been overridden*.
- If the method has *not* been overridden, use `this` as usual.

Accessing Methods from the Superclass

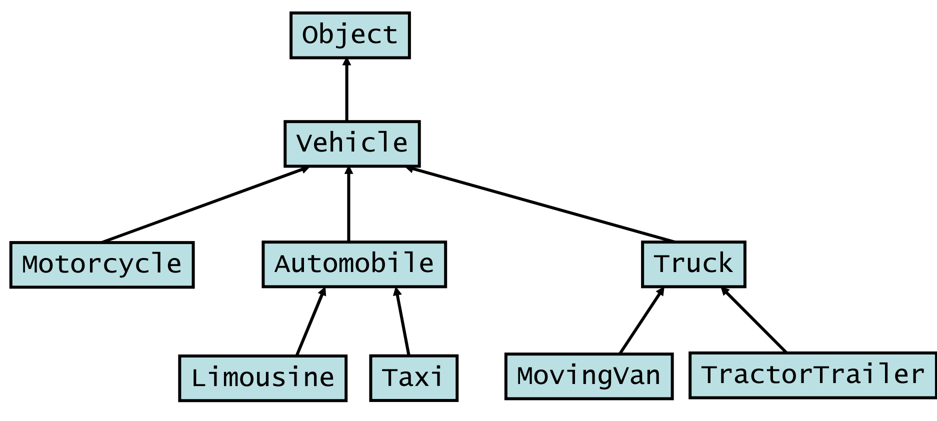
- We need to override *all* of the inherited mutators:

```
// Square versions
public void setWidth(int w) {
    super.setWidth(w);
    super.setHeight(w);
}

public void setHeight(int h) {
    super.setWidth(h);
    super.setHeight(h);
}

public void grow(int dw, int dh) {
    if (dw != dh) {
        throw new IllegalArgumentException();
    }
    super.setWidth(this.getWidth() + dw);
    super.setHeight(this.getHeight() + dh);
}
```

`getWidth()` and `getHeight()` are not overridden, so we use `this`.

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", …);
```

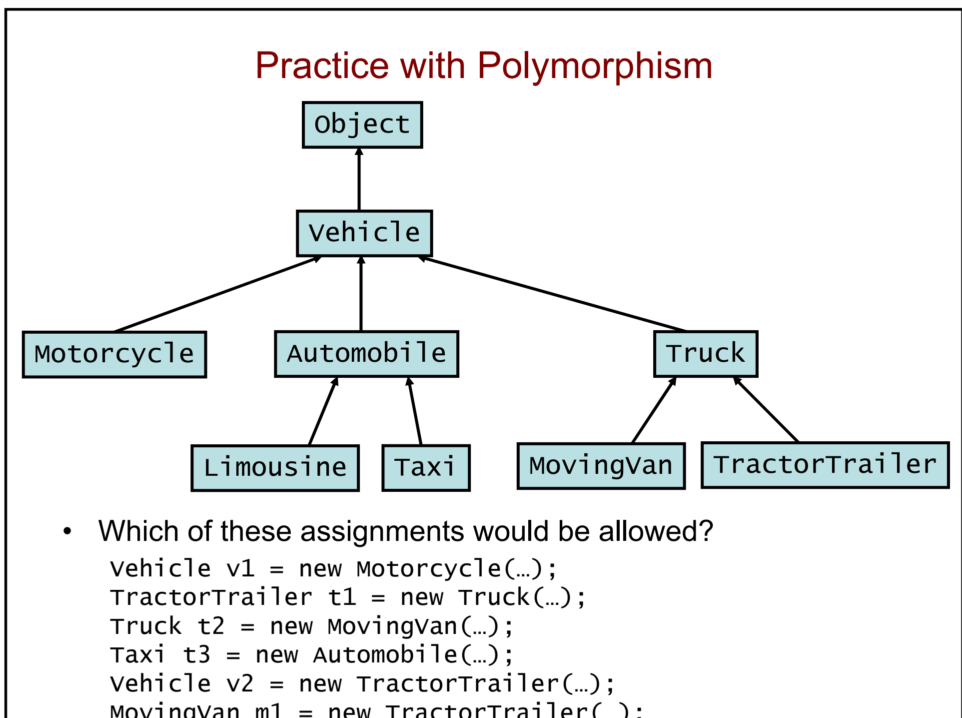
Processing a Collection of Objects

- 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.
  - they are instances of `Vehicle` or a subclass of `Vehicle`
  - thus, they must all have a `getYear()` method

Practice with Polymorphism



- 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:

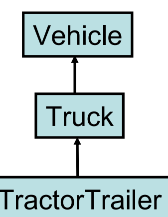
```
Vehicle[] fleet = new Vehicle[5];
fleet[0] = new Automobile("Honda", "Civic", 2005);
fleet[1] = new Motorcycle("Harley", …);
fleet[2] = new TractorTrailer("Mack", …);
```

- 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()
```

Vehicle
getMake

Truck
getNumAxles

MovingVan

TractorTrailer
getNumAxles
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()
```

Vehicle
getMake

Truck
getNumAxles

MovingVan

TractorTrailer
getNumAxles
isSleeper

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();
```

G
method1
method2

E
method2
method3

F
method2

H
method1