Chapter 4 – Fundamental Data Types

Dr Kafi Rahman, PhD
Email: kafi@truman.edu
Truman State University

Understanding Classes, Objects, and Encapsulation (cont'd.)

- Method
  - A self-contained block of program code that carries out an action
  - Similar to a procedure
- Encapsulation
  - Conceals internal values and methods from outside sources
  - Provides security
  - Keeps data and methods safe from inadvertent changes

Understanding Inheritance and Polymorphism

- Inheritance
  - An important feature of object-oriented programs
  - Classes share attributes and methods of existing classes but with more specific features
  - Helps you understand real-world objects
- Polymorphism
  - Means "many forms"
  - Allows the same word to be interpreted correctly in different situations based on context

Features of the Java Programming Language

- Java
  - Developed by Sun Microsystems
  - An object-oriented language
  - General-purpose
  - Advantages
    - Security features
    - Architecturally neutral

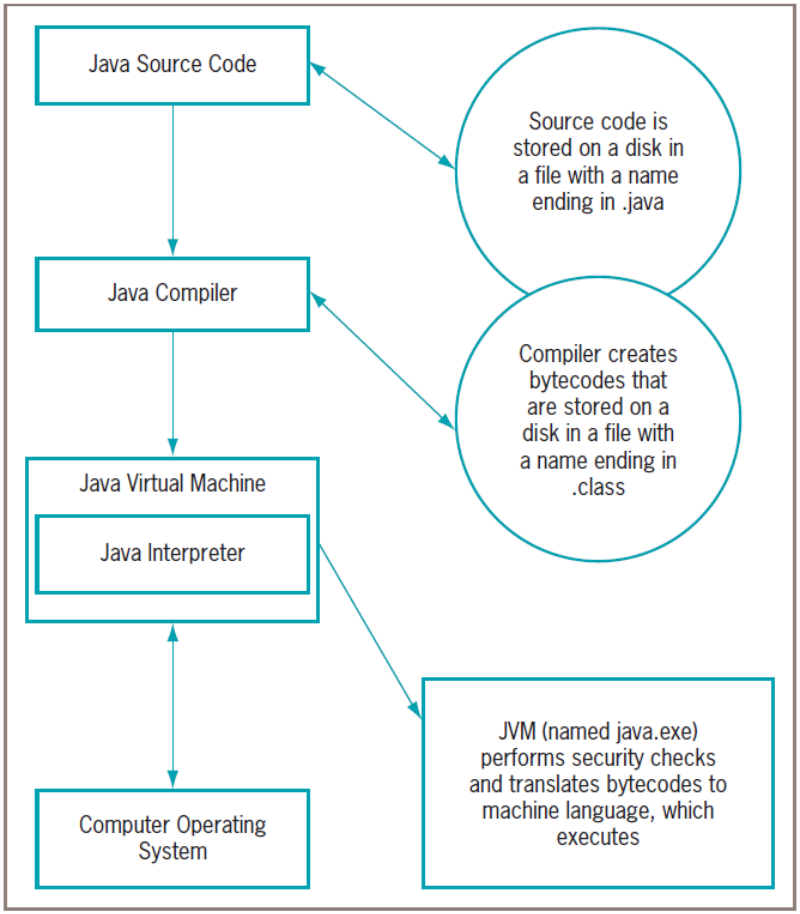
Features of the Java Programming Language (cont'd.)

- Java (cont'd.)
  - Can be run on a wide variety of computers
  - Does not execute instructions on the computer directly
  - Runs on a hypothetical computer known as a Java Virtual Machine (JVM)

Features of the Java Programming Language (cont'd.)

- Development environment
  - A set of tools used to write programs
- Bytecode
  - Statements saved in a file
  - A binary program into which the Java compiler converts source code
- Java interpreter
  - Checks bytecode and communicates with the operating system
  - Executes bytecode instructions line by line within the Java Virtual Machine

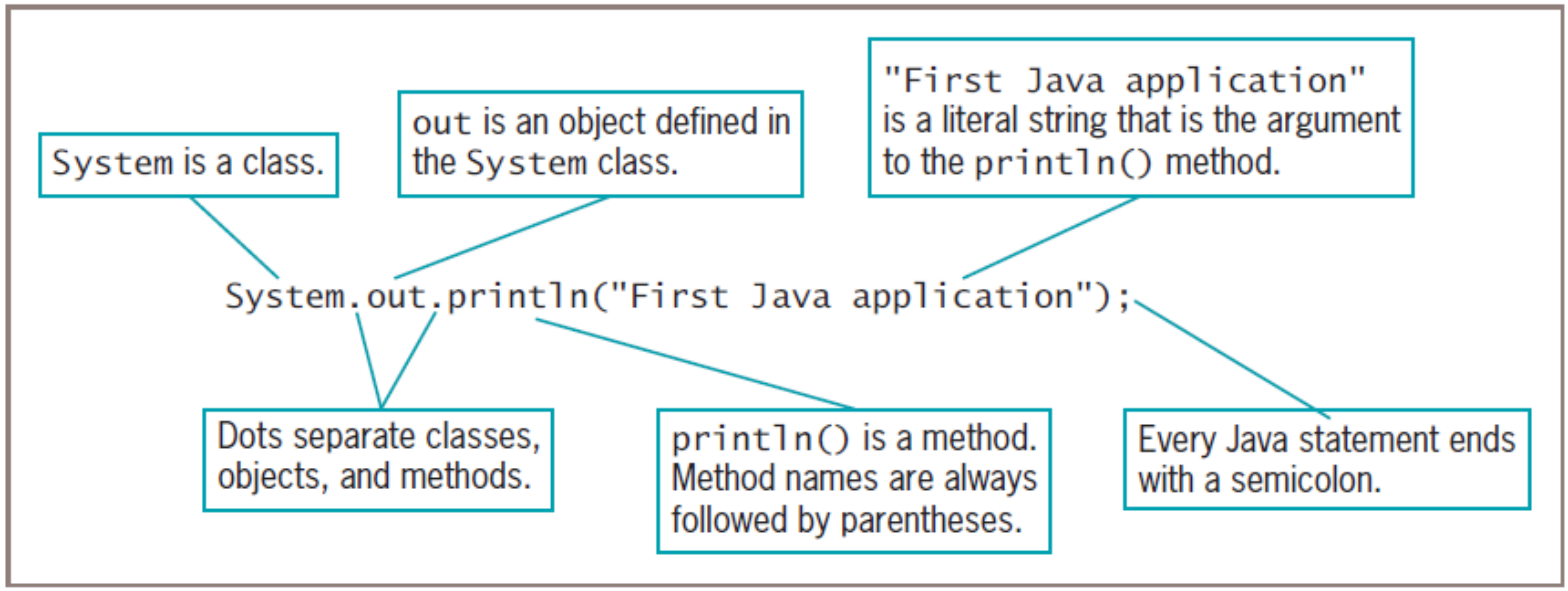
Features of the Java Programming Language (cont'd.)



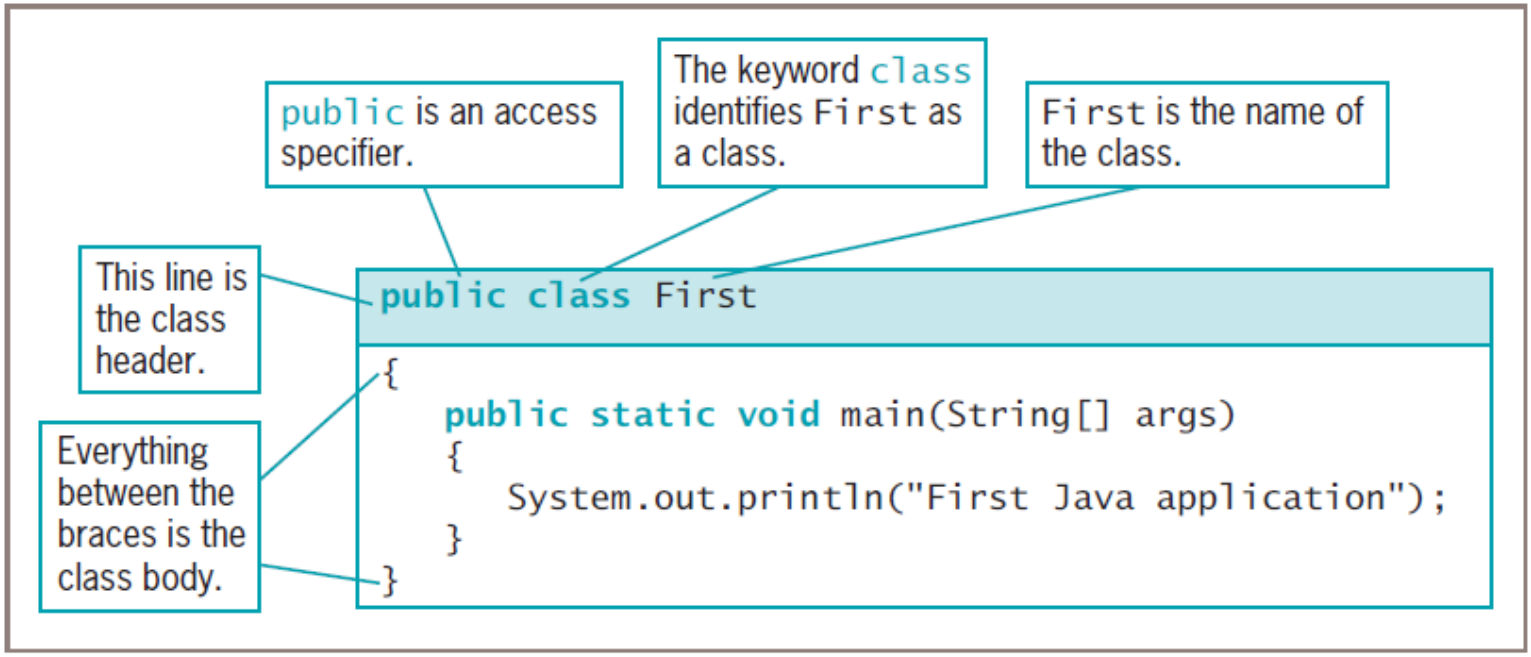
Java Program Types

- Applets
  - Programs embedded in a Web page
- Java applications
  - Called Java stand-alone programs
  - Console applications
    - Support character output
  - Windowed applications
    - Menus
    - Toolbars
    - Dialog boxes

Understanding the Statement that Produces the Output



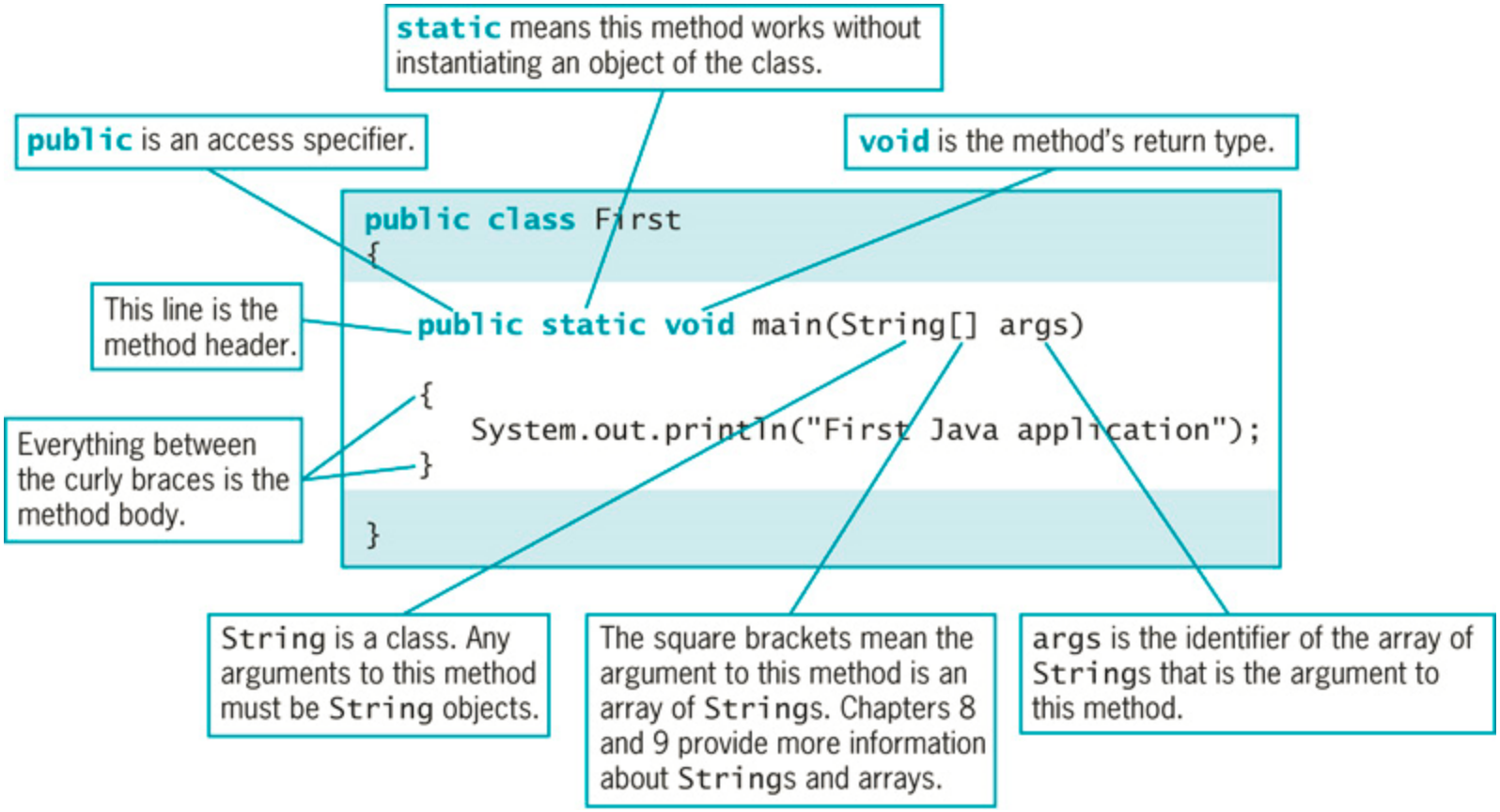
Understanding the First Class (cont'd.)



Understanding the main() Method

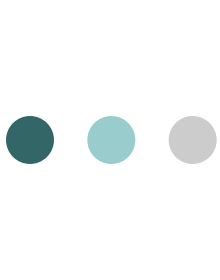
- static
  - A reserved keyword
  - Means the method is accessible and usable even though no objects of the class exist
- void
  - Use in the main() method header
  - Does not indicate the main() method is empty
  - Indicates the main() method does not return a value when called
  - Does not mean that main() doesn't produce output

Understanding the main() Method (cont'd.)



Understanding the main() Method (cont'd.)

```
public class AnyClassName
{
   public static void main(String[] args)
   {
      /******/
   }
}
```



Saving a Java Class

- Saving a Java class
  - Save the class in a file with exactly the same name and .java extension
    - For public classes, class name and filename must match exactly

Adding Comments to a Java Class (cont'd.)

- Types of Java comments (cont'd.)
  - Javadoc comments
    - A special case of block comments
    - Begin with a slash and two asterisks (/**)
    - End with an asterisk and a forward slash (*/)
    - Use to generate documentation

Adding Comments to a Java Class (cont'd.)

```
// Demonstrating comments
/* This shows
   that these comments
   don't matter */
System.out.println("Hello"); // This line executes
   // up to where the comment started
/* Everything but the println()
     is a comment */
```

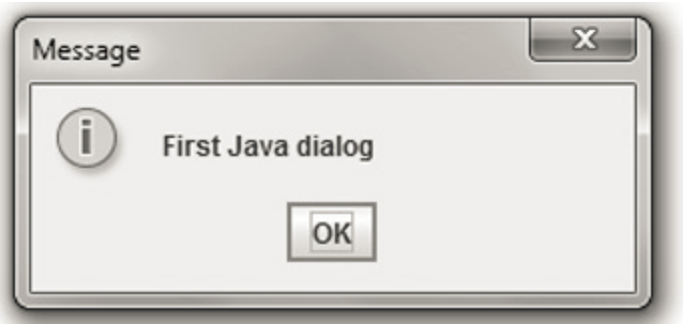
Creating a Java Application that Produces GUI Output

- JOptionPane
  - Produces dialog boxes
- Dialog box
  - A GUI object resembling a window
  - Messages placed for display
- import statement
  - Use to access a built-in Java class
- Package
  - A group of classes

Creating a Java Application that Produces GUI Output (cont'd.)

```
import javax.swing.JOptionPane;
public class FirstDialog
{
   public static void main(String[] args)
   {
      JOptionPane.showMessageDialog(null, "First Java dialog");
   }
}
```

Creating a Java Application that Produces GUI Output (cont'd.)



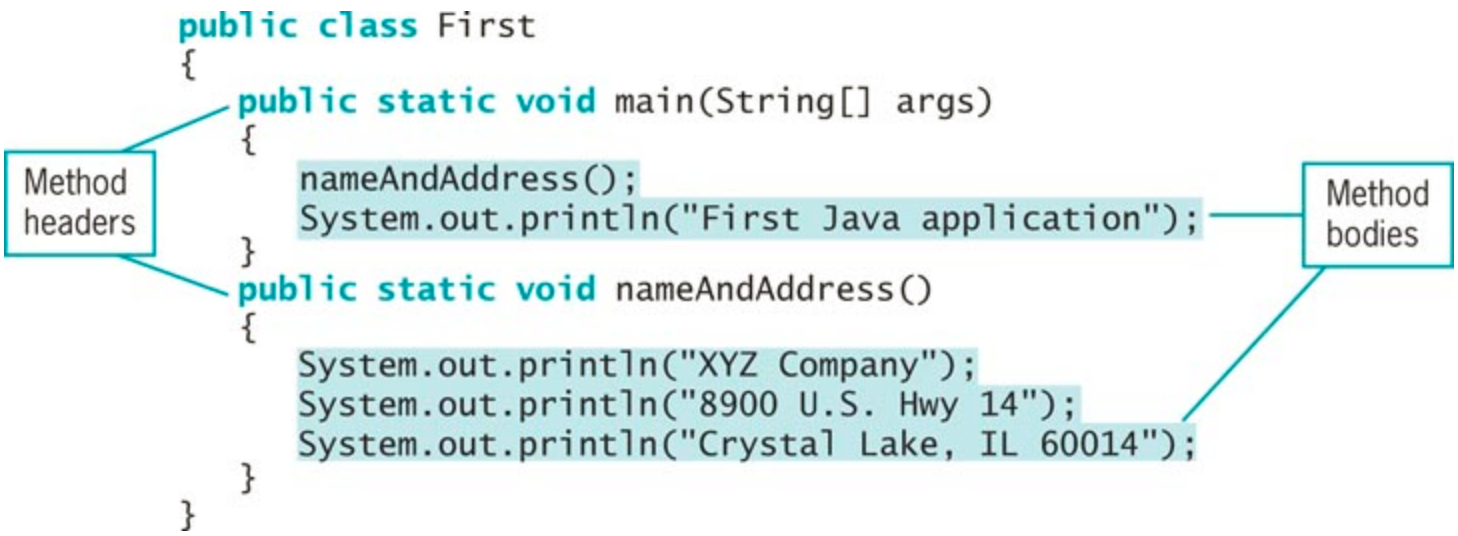
Understanding Method Calls and Placement (cont'd.)

- Main() method executes automatically
- Other methods are called as needed

Understanding Method Calls and Placement (cont'd.)

```
public class First
{
   // You can place additional methods here, before main()
   public static void main(String[] args)
   {
      nameAndAddress();
      System.out.println("First Java application");
   }
   // You can place additional methods here, after main()
}
```

Understanding Method Construction (cont'd.)




```
public class First
{
    public static void main(String[] args)
    {
        nameAndAddress();
        System.out.println("First Java application");
    }
    public static void nameAndAddress()
    {
        System.out.println("XYZ Company");
        System.out.println("8900 U.S. Hwy 14");
        System.out.println("Crystal Lake, IL 60014");
    }
}
```

Creating a Class (cont'd.)

- Extended
  - To be used as a basis for any other class
- Data fields
  - Variables declared within a class but outside of any method
- Instance variables
  - Nonstatic fields given to each object

Creating a Class (cont'd.)

- Private access for fields
  - No other classes can access the field's values
  - Only methods of the same class are allowed to use private variables
- Information hiding
- Most class methods are public

Declaring Objects and Using Their Methods (cont'd.)

- Reference to the object
  - The name for a memory address where the object is held
- Constructor method
  - A method that creates and initializes class objects
  - You can write your own constructor methods
  - Java writes a constructor when you don't write one
  - The name of the constructor is always the same as the name of the class whose objects it constructs

An Introduction to Using Constructors

- Employee chauffeur = new Employee();
  - Actually a calling method named Employee()
- Default constructors
  - Require no arguments
  - Created automatically by a Java compiler
    - For any class
    - Whenever you do not write a constructor

An Introduction to Using Constructors (cont'd.)

- A constructor method:
  - Must have the same name as the class it constructs
  - Cannot have a return type
  - public access modifier

An Introduction to Using Constructors (cont'd.)

- A constructor method:
  - Let us create one in a program

Chapter 5 - Decisions

The if Statement

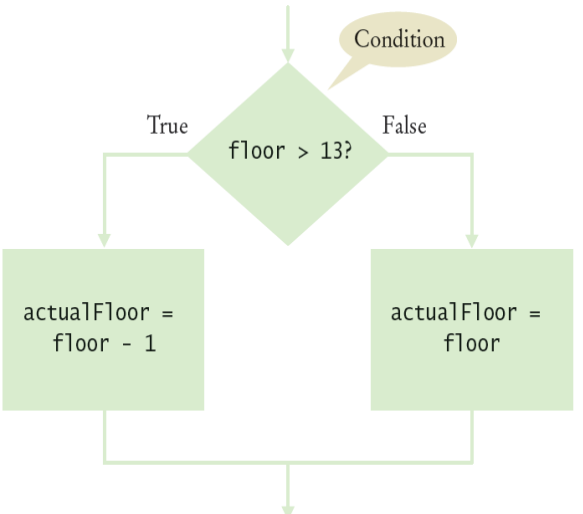
- The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

This elevator panel "skips" the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

```
int actualFloor;
if (floor > 13)
{
  actualFloor = floor - 1;
}
else
{
  actualFloor = floor;
}
```

The if Statement

- Flowchart with two branch



- You can include as many statements in each branch as you like.

Self Check 5.5

The variables `fuelAmount` and `fuelCapacity` hold the actual amount of fuel and the size of the fuel tank of a vehicle. If less than 10 percent is remaining in the tank, a status light should show a red color; otherwise it shows a green color. Simulate this process by printing out either `"red"` or `"green"`.

Answer:

```
if (fuelAmount < 0.10 * fuelCapacity)
{
  System.out.println("red");
}
else
{
  System.out.println("green");
}
```

Comparing Values: Relational Operators

- Relational operators compare values:

Java	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

- The == denotes equality testing:

```
floor = 13; // Assign 13 to floor

if (floor == 13) // Test whether floor equals 13
```

- Relational operators have lower precedence than arithmetic operators:

```
floor - 1 < 13
```

Comparing Strings

- To test whether two strings are equal to each other, use `equals` method:

```
if (string1.equals(string2)) . . .
```

- Don't use == for strings!

```
if (string1 == string2) // Not useful for string variables
```

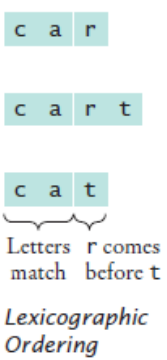
Comparing Strings - compareTo method

- `compareTo` method compares strings in lexicographic order - dictionary order.
- `string1.compareTo(string2) < 0` means:

  `string1` comes before `string2` in the dictionary
- `string1.compareTo(string2) > 0` means:

  `string1` comes after `string2` in the dictionary
- `string1.compareTo(string2) == 0` means:

  `string1` and `string2` are equal
- Lexicographic Ordering



Lexicographic Ordering

Comparing Objects

- The == operator tests whether two object references are identical,

  whether they refer to the same object.

- Look at this code

```
Rectangle box1 = new
Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

  `box1 == box2` is true

  `box1 == box3` is false

- Use the `equals` method to test if two rectangles have the same content

  `box1.equals(box3)`

  They have the same upper-left corner and the same width and height

- Caveat: `equals` must be defined for the class

Relational Operator Examples

Table 2 Relational Operator Examples

Expression	Value	Comment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
`3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.
`3 = 6 / 2`	**Error**	Use == to test for equality.
`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal. See Section 5.2.2.
`"10" > 5`	**Error**	You cannot compare a string to a number.
`"Tomato".substring(0, 3).equals("Tom")`	`true`	Always use the `equals` method to check whether two strings have the same contents.
`"Tomato".substring(0, 3) == ("Tom")`	`false`	Never use == to compare strings; it only checks whether the strings are stored in the same location. See Common Error 5.2 on page 192.

Self Check 5.9

Supply a condition in this `if` statement to test whether the user entered a Y:

```
System.out.println("E
nter Y to quit.");
String input =
in.next();
if (. . .)
{
  System.out.println("Goodbye.");
}
```

Answer: `input.equals("Y")`

Self Check 5.10

Give two ways of testing that a string `str` is the empty string.

Answer: `str.equals("")` or `str.length() == 0`

Boolean Variables and Operators

- You often need to combine Boolean values when making complex decisions
- An operator that combines Boolean conditions is called a Boolean operator.
- The `&&` operator is called **and**

  Yields `true` only when both conditions are true.
- The `||` operator is called **or**

  Yields the result `true` if at least one of the conditions is true.

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

A	!A
true	false
false	true

Figure 9 Boolean Truth Tables

Boolean Variables and Operators

Table 5 Boolean Operator Examples

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for "either-or". If both conditions are true, the result is true.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The `&&` operator has a higher precedence than the `||` operator (see Appendix B).
`0 < x < 100`	**Error**	**Error:** This expression does not test whether `x` is between 0 and 100. The expression `0 < x` is a Boolean value. You cannot compare a Boolean value with the integer 100.
`x && y > 0`	**Error**	**Error:** This expression does not test whether `x` and `y` are positive. The left-hand side of `&&` is an integer, `x`, and the right-hand side, `y > 0`, is a Boolean value. You cannot use `&&` with an integer argument.
`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Application: Input Validation

- You need to make sure that the user-supplied values are valid before you use them.
- Example: elevator panel has buttons labeled 1 through 20 (but not 13)

  The number 13 is invalid
-

```
if (floor == 13)
{
   System.out.println("Error: There is no thirteenth floor.");
}
```

- Numbers out of the range 1 through 20 are invalid

```
if (floor <= 0 || floor > 20)
{
   System.out.println("Error: floor must be between 1 and 20.");
}
```

- To avoid input that is not an integer

```
if (in.hasNextInt())
{
   int floor = in.nextInt();
   // Process the input value.
}
else
{
   System.out.println("Error: Not an integer.");
}
```

Chapter 6 - Loops

The while Loop

- How can you "Repeat steps while the balance is less than $20,000?"
- With a `while` loop statement
- Syntax

```
while (condition)
{
  statements
}
```

- As long condition is true, the statements in the `while` loop execute.

Self Check 6.5

What does the following loop print?

```
int n = 1;
while (n < 100)
{
   n = 2 * n;
   System.out.print(n + " ");
}
```

Answer: 2 4 8 16 32 64 128

Problem Solving: Hand-Tracing

- A simulation of code execution in which you step through instructions and track the values of the variables.
- What value is displayed?

```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```

The for Loop

- To execute a sequence of statements a given number of times:

Could use a `while` loop controlled by a counter

```
int counter = 1; // Initialize the counter  while
(counter <= 10) // Check the counter
{
  System.out.println(counter);  counter++; //
  Update the counter
}
```

Use a special type of loop called `for` loop

```
for (int counter = 1; counter <= 10; counter++)
{
  System.out.println(counter);
}
```

- Use a `for` loop when a variable runs from a starting value to an ending value with a constant increment or decrement.

The for Loop

- To traverse all the characters of a string:

```
for (int i = 0; i < str.length(); i++)
{
  char ch = str.charAt(i);
  // Process ch.
}
```

- The counter variable `i` starts at 0, and the loop is terminated when `i` reaches the length of the string.

Questions?