

1

Introduction

About This Course

Java SE 8 Fundamentals Quiz Lesson 3

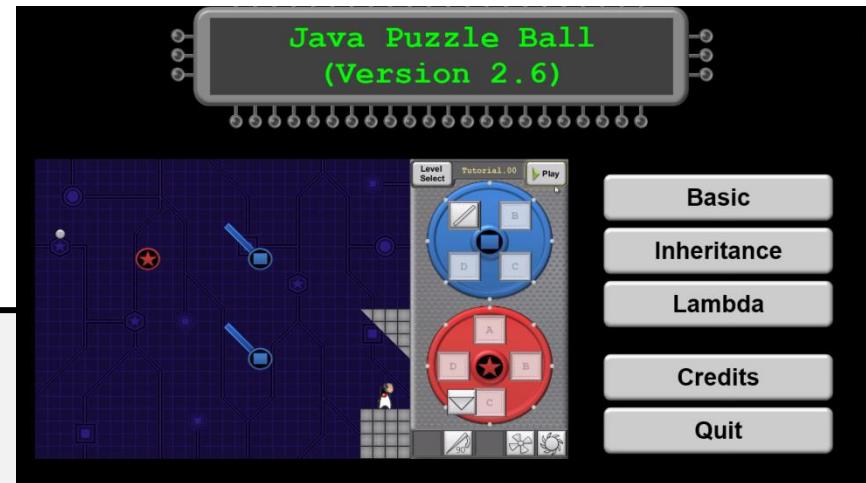
Based on what you see in the directory display, what is the name of the package in which `Student.java` resides?

type your text here

```
Directory of C:\test
02/14/2014 12:19 PM <DIR> .
02/14/2014 12:19 PM .
02/14/2014 12:19 PM 0 Student.java
1 File(s) 0 bytes
2 Dir(s) 142,443,081,728 bytes free
C:\test>
```

SUBMIT

```
12 Shirt myShirt = new Shirt();
13 Shirt yourShirt = new Shirt();
14
15 myShirt = yourShirt;
16
17 myShirt.colorCode = 'R';
18 yourShirt.colorCode = 'G';
19
20 System.out.println("Shirt color: " + myShirt.colorCode);
```



Audience

- Beginners to programming who have basic mathematical, logical, and analytical problem-solving skills and who want to begin learning the Java programming language
- Novice programmers and those programmers who prefer to start learning the Java programming language at an introductory level
- Students who want to begin their study of the Oracle Certified Java Associate exam

Course Objectives

After completing this course, you should be able to:

- Demonstrate knowledge of basic programming language concepts
- Demonstrate knowledge of the Java programming language
- Implement intermediate Java programming and object-oriented (OO) concepts

Schedule

Day One

- Getting Started
 - Lesson 1: Introduction
 - Lesson 2: What Is a Java Program?
- The Basic Shopping Cart
 - Lesson 3: Creating a Java Main Class
 - Lesson 4: Data in a Cart
 - Lesson 5: Managing Multiple Items

Schedule

Day Two

- Filling the Cart
 - Lesson 6: Describing Objects and Classes
 - Lesson 7: Manipulating and Formatting the Data in Your Program
- Improving Cart Efficiency
 - Lesson 8: Creating and Using Methods

Day Three

- Lesson 9: Using Encapsulation
- Expanding the Business
 - Lesson 10: More on Conditionals

Schedule

Day Four

- Lesson 11: Working with Arrays, Loops, and Dates
- Bringing It Home
 - Spring / Spring Boot

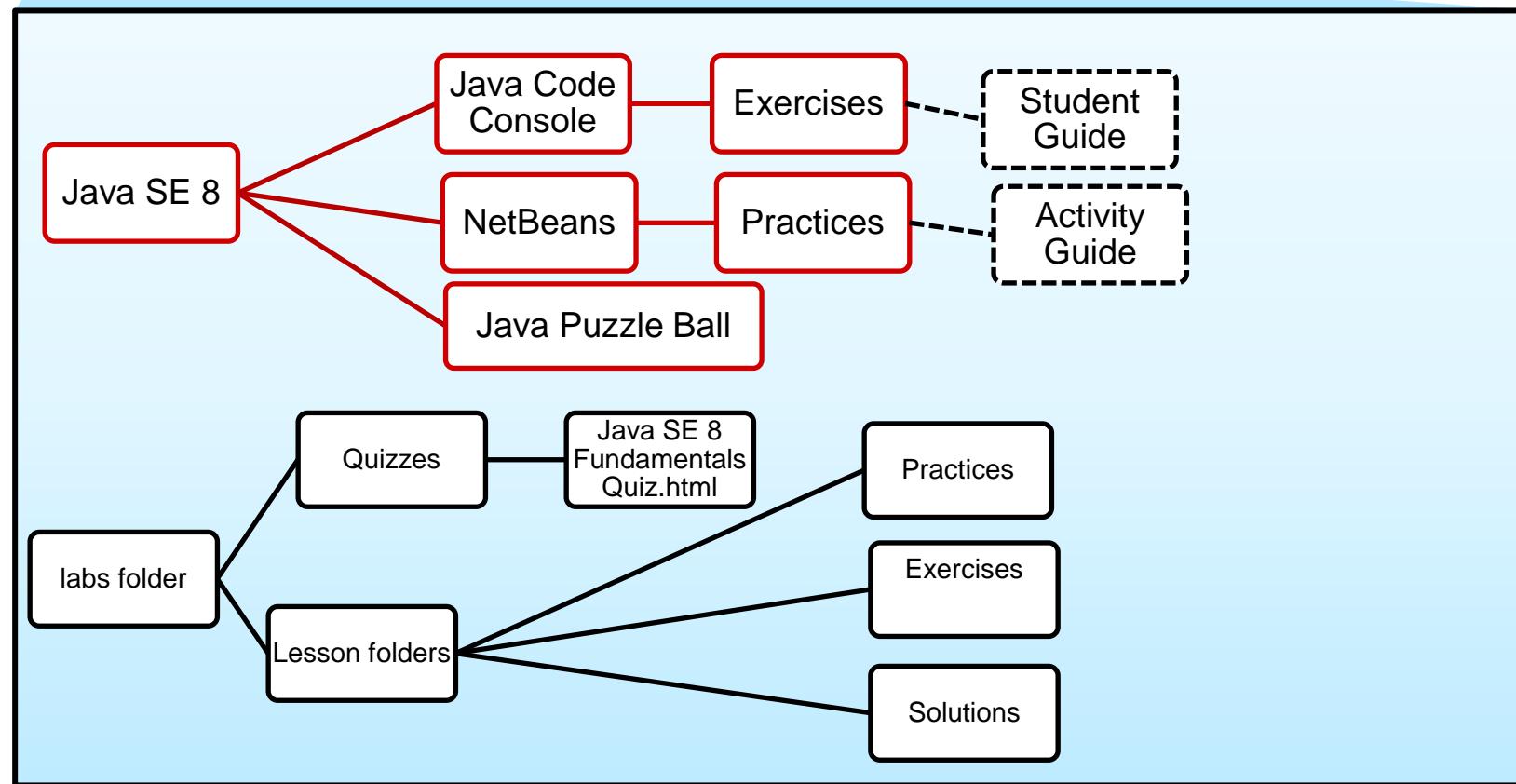
Day Five

- Spring/Spring Boot

Course Environment



Classroom Computer



Test Your Lab Machines

1. Go to your lab machine.
2. Open the Firefox browser.
3. Go to the URL specified by your instructor



Quiz

- a. What is your name?
- b. What do you do for a living, and where do you work?
- c. What is the most interesting place you have visited?
- d. Why are you interested in Java?



Summary

In this lesson, you reviewed the course objectives and the tentative class schedule. You met your fellow students, and you saw an overview of the computer environment that you will use during the course.

Enjoy the next five days of *Java SE 8 Fundamentals*.

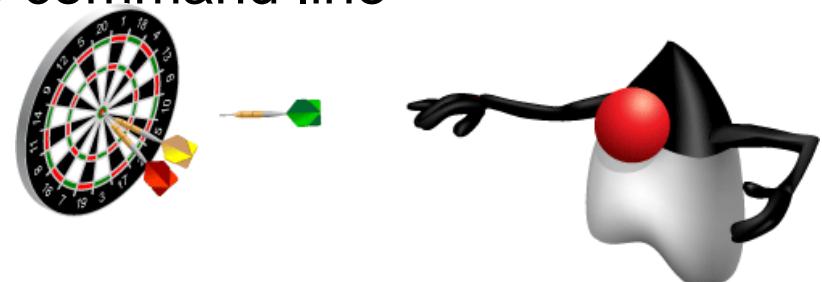


What Is a Java Program?

Objectives

After completing this lesson, you should be able to:

- Contrast the terms “platform-dependent” and “platform-independent”
- Describe the purpose of the JVM
- Explain the difference between a procedural program and an object-oriented program
- Describe the purpose of `javac` and `java` executables
- Verify the Java version on your system
- Run a Java program from the command line



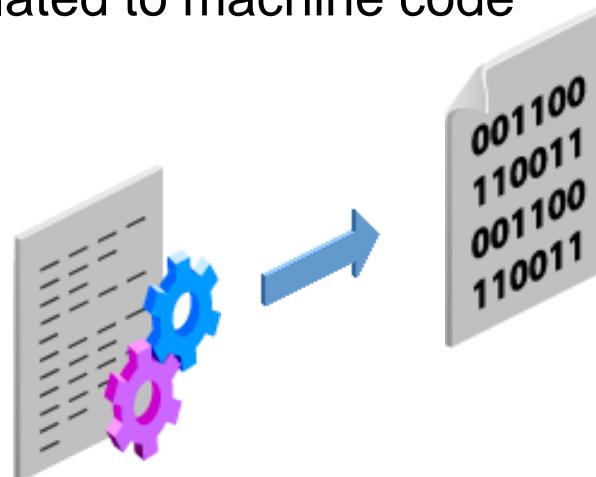
Topics

- Introduction to computer programs
- Introduction to the Java language
- Verifying the Java development environments
- Running and testing a Java program

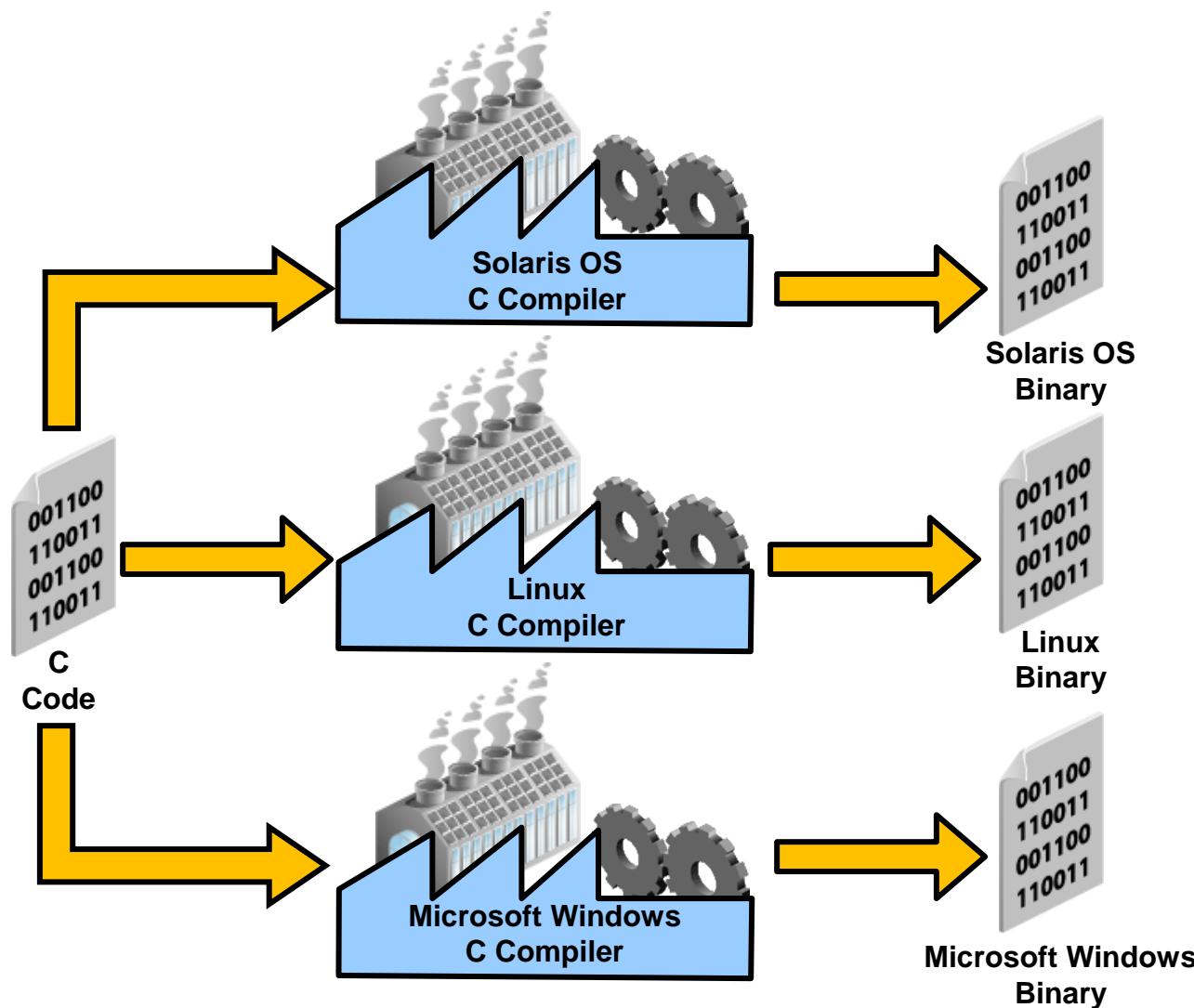
Purpose of a Computer Program

A computer program is a set of instructions that run on a computer or other digital device.

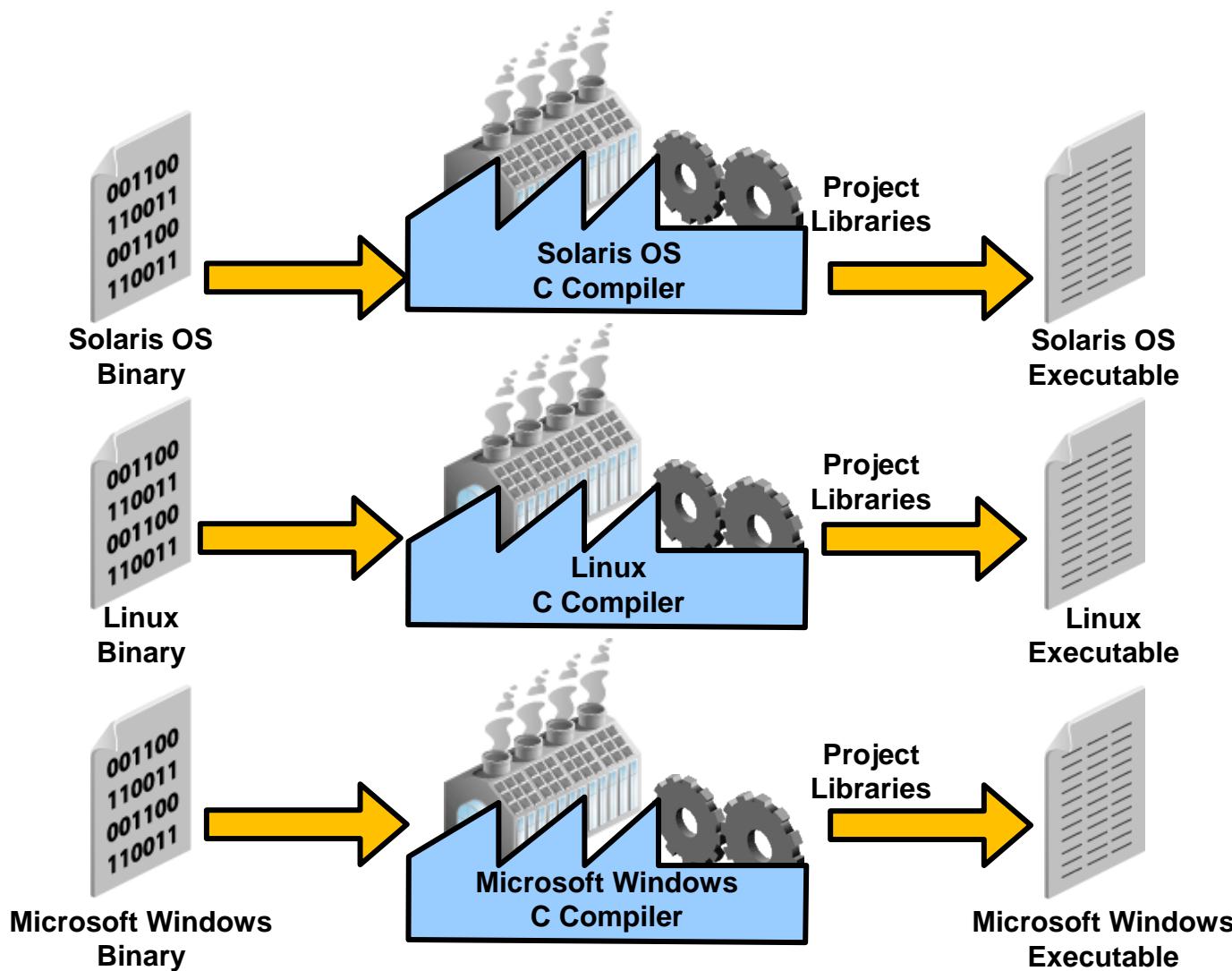
- At the machine level, the program consists of binary instructions (1s and 0s).
 - Machine code
- Most programs are written in *high-level* code (readable).
 - Must be translated to machine code



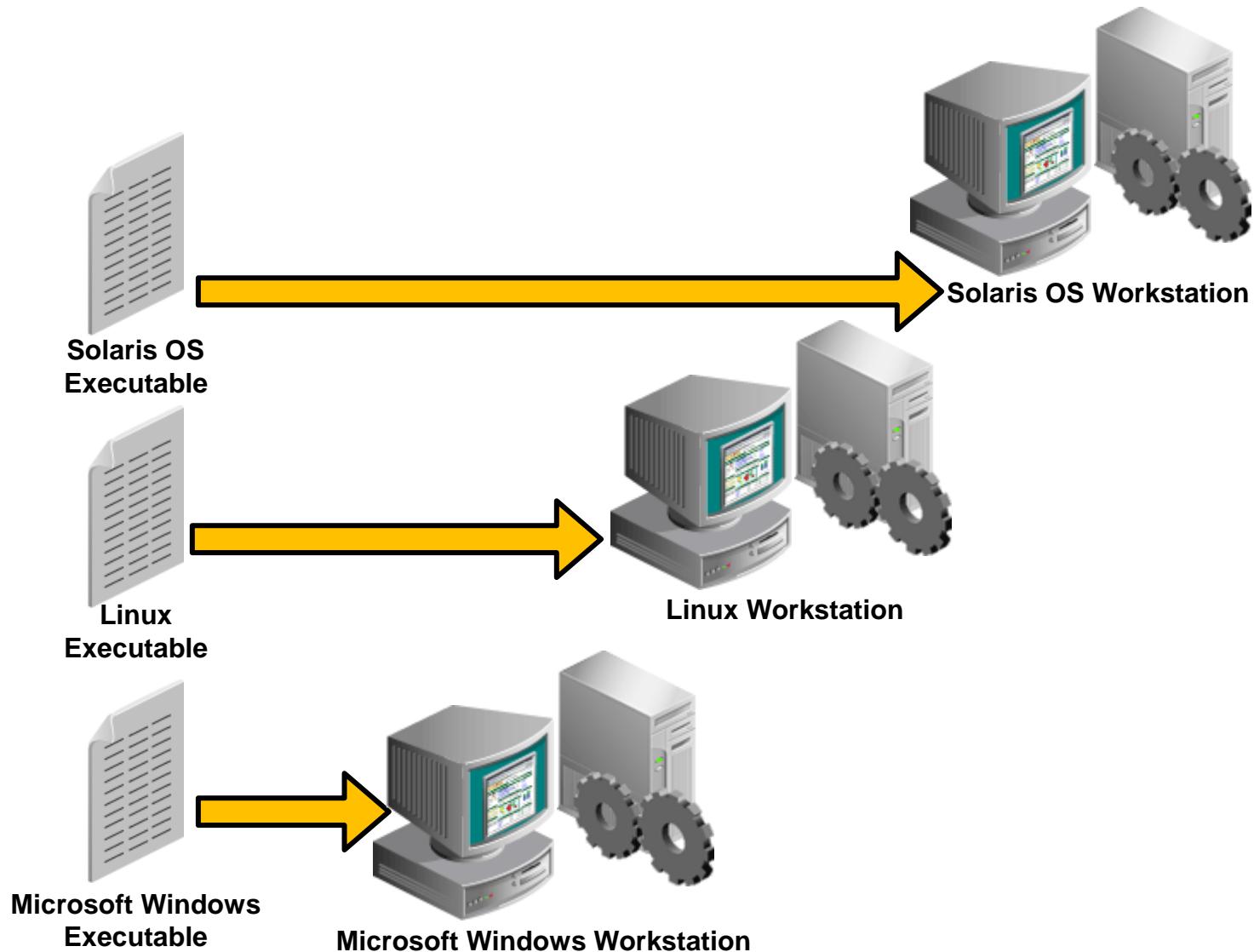
Translating High-Level Code to Machine Code



Linked to Platform-Specific Libraries



Platform-Dependent Programs



Topics

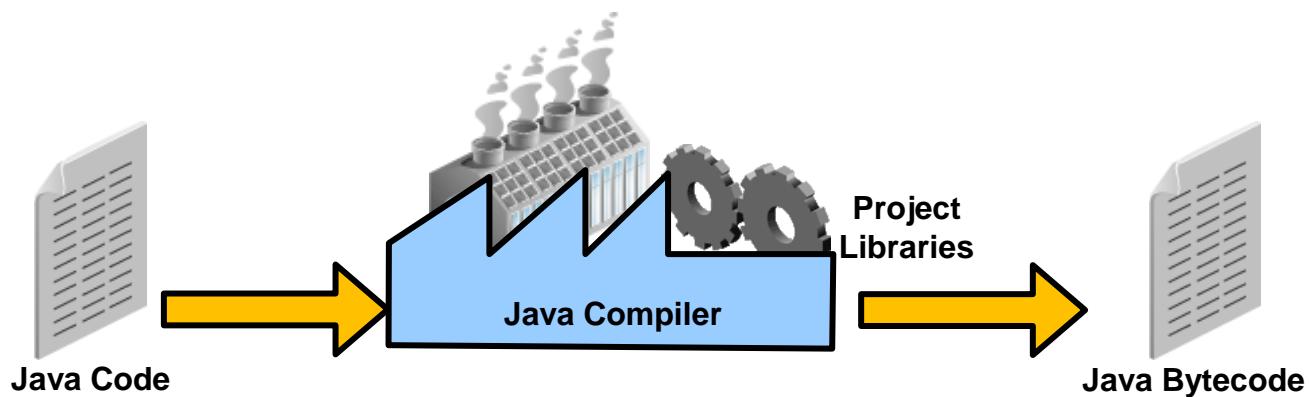
- Introduction to computer programs
- **Introduction to the Java language**
- Verifying the Java development environment
- Running and testing a Java program

Key Features of the Java Language

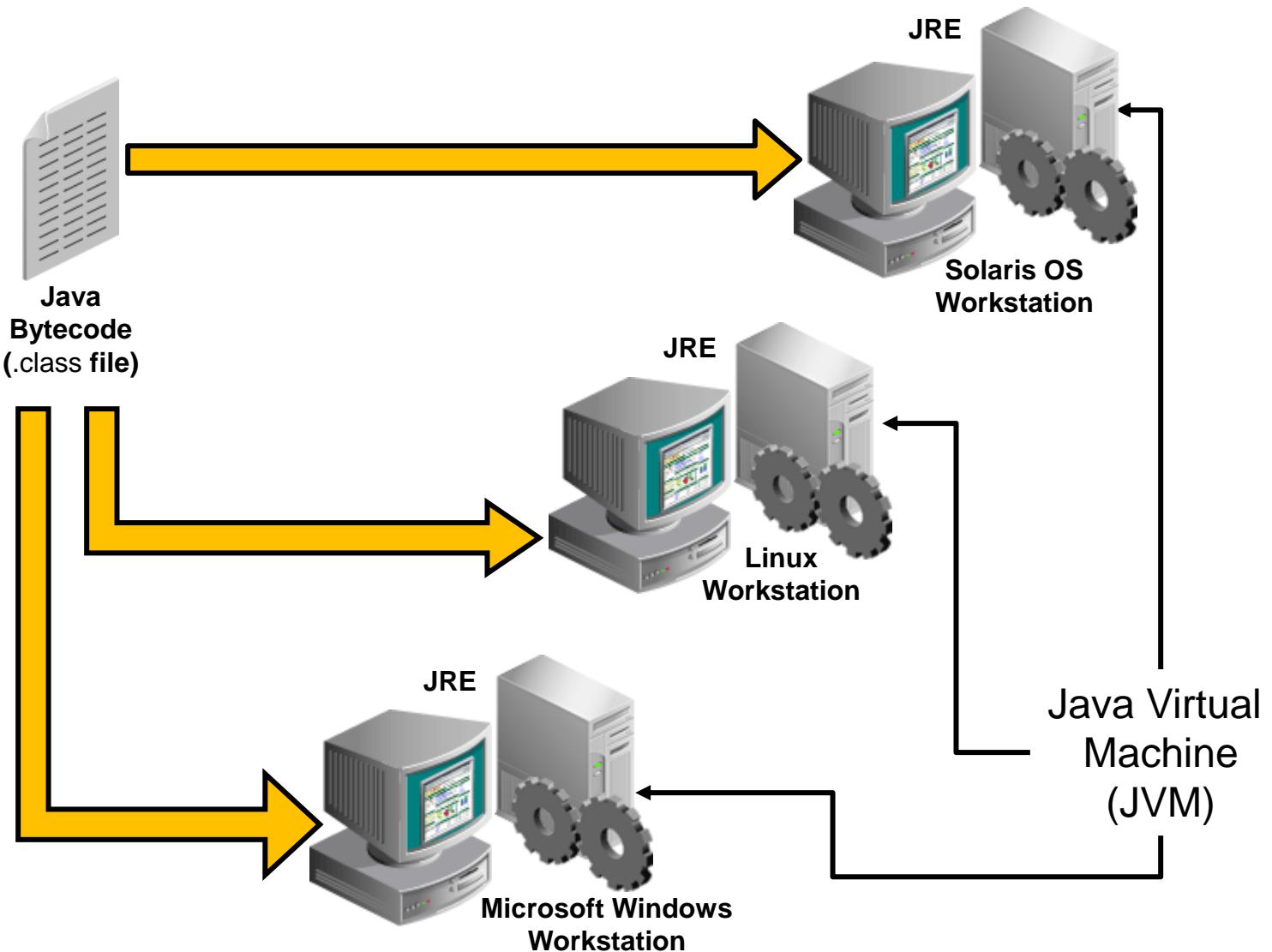
Some of the features that set Java apart from most other languages are that:

- It is platform-independent
- It is object-oriented

Java Is Platform-Independent

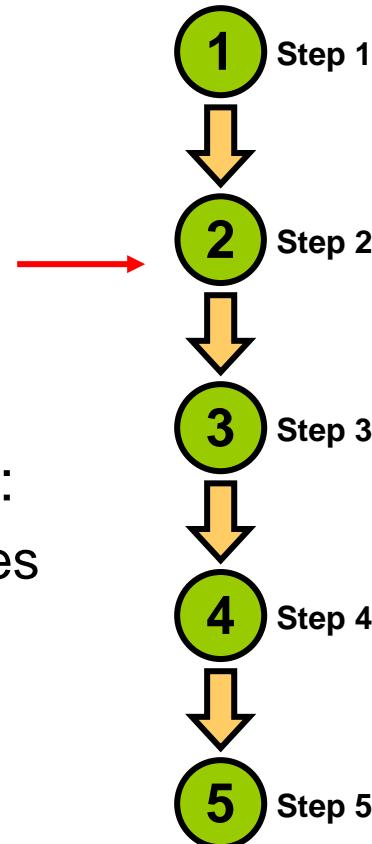


Java Programs Run In a Java Virtual Machine



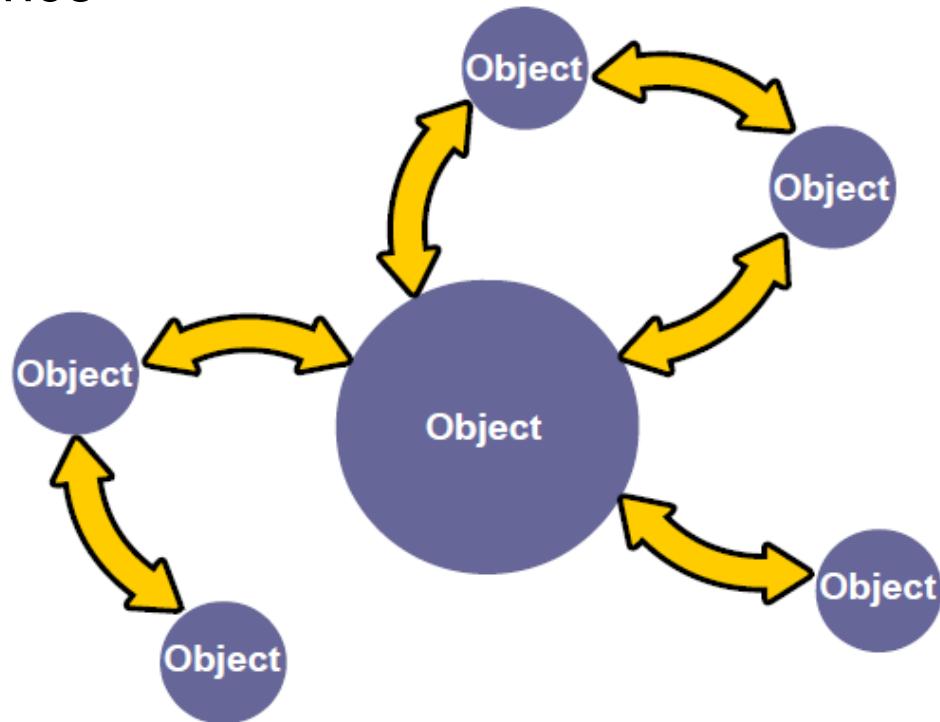
Procedural Programming Languages

- Many early programming languages followed a paradigm called *Procedural Programming*.
- These languages use a sequential pattern of program execution.
- Drawbacks to procedural programming:
 - Difficult to translate real-world use cases to a sequential pattern
 - Difficult to maintain programs
 - Difficult to enhance as needed



Java Is an Object-Oriented Language

- Interaction of objects
- No prescribed sequence
- Benefits:
 - Modularity
 - Information hiding
 - Code reuse
 - Maintainability



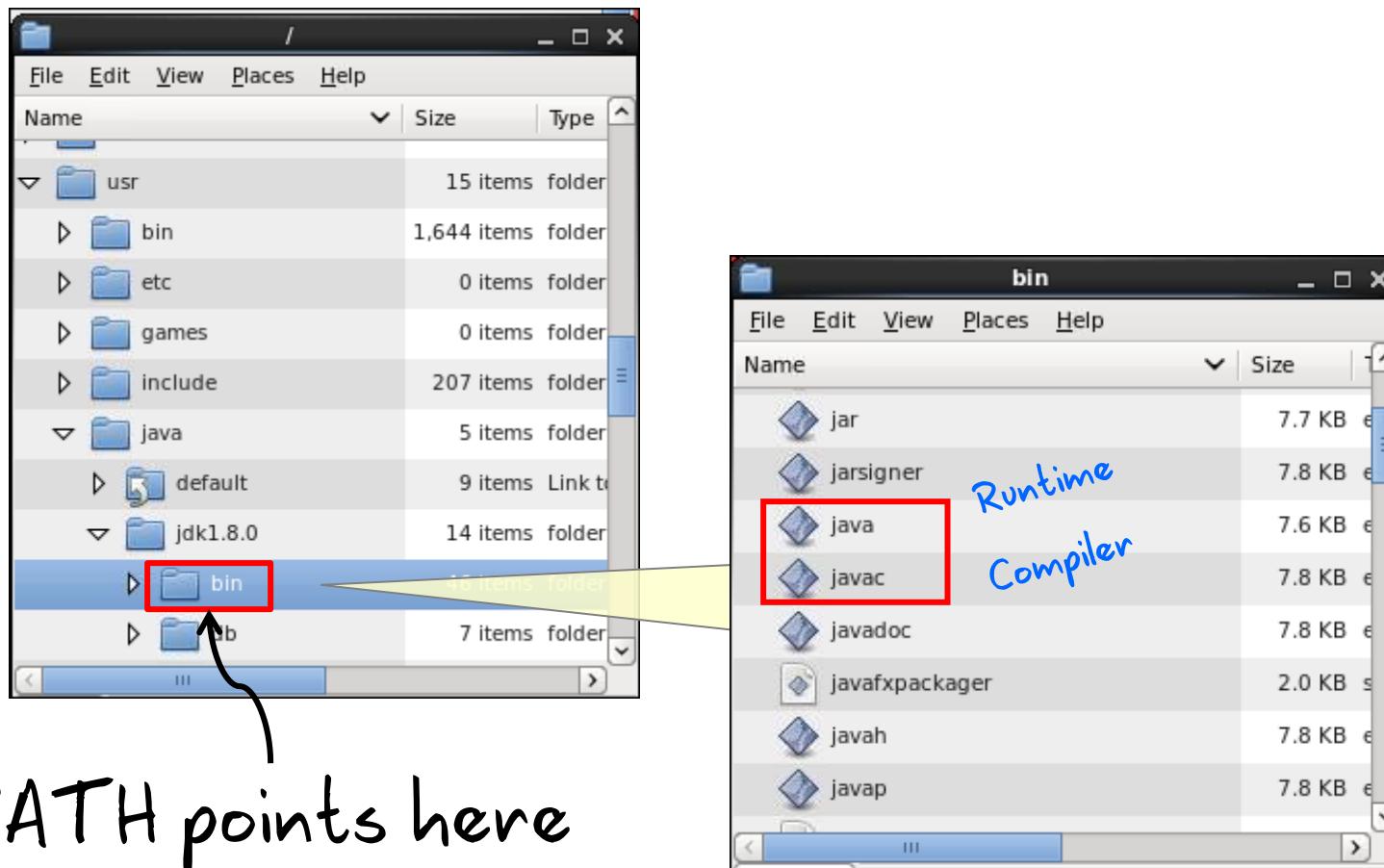
Topics

- Introduction to computer programs
- Introduction to the Java language
- **Verifying the Java development environment**
- Running and testing a Java program

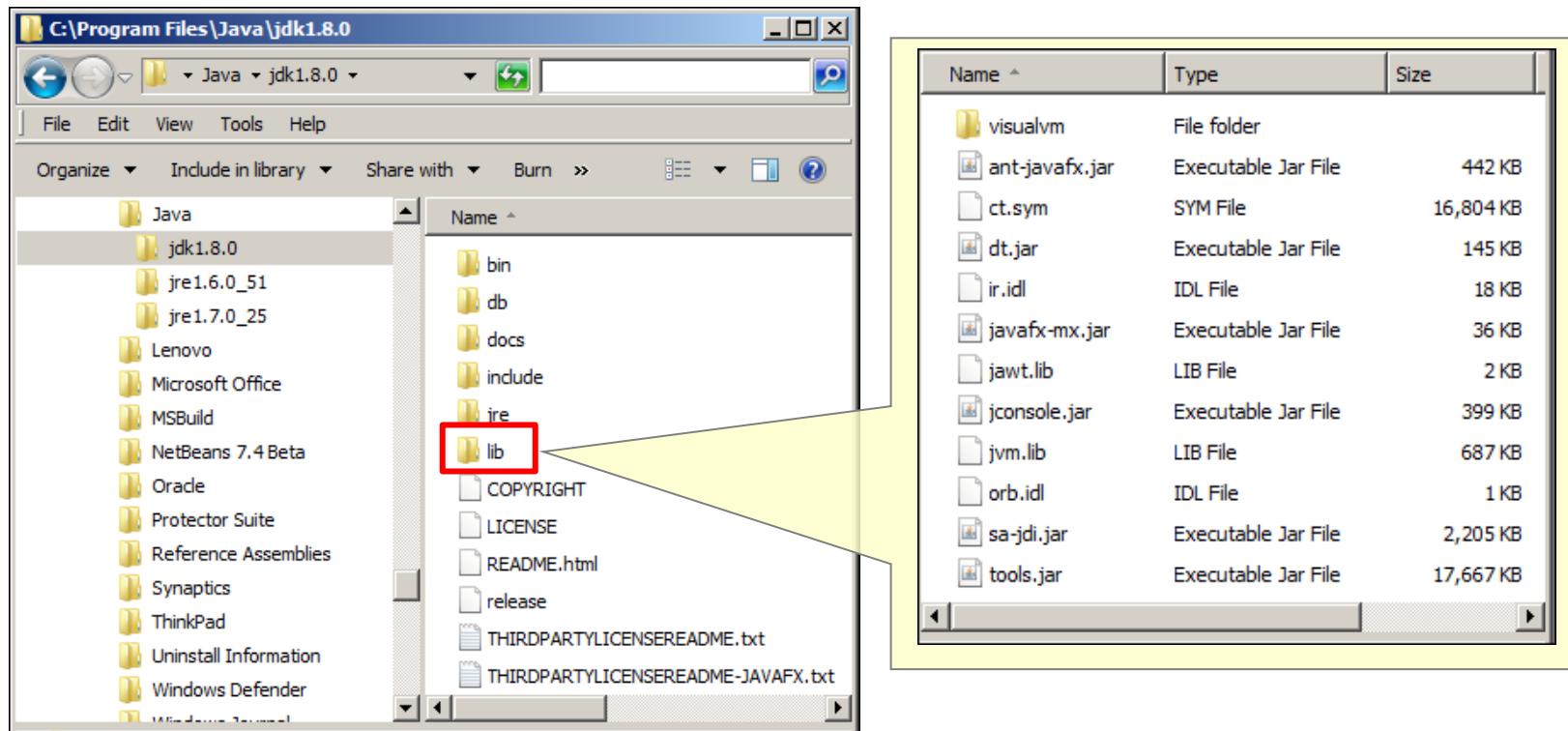
Verifying the Java Development Environment

1. Download and install the Java Development Kit (JDK) from oracle.com/java.
2. Explore the Java Help menu.
3. Compile and run a Java application by using the command line.

Examining the Installed JDK (Linux Example): The Tools



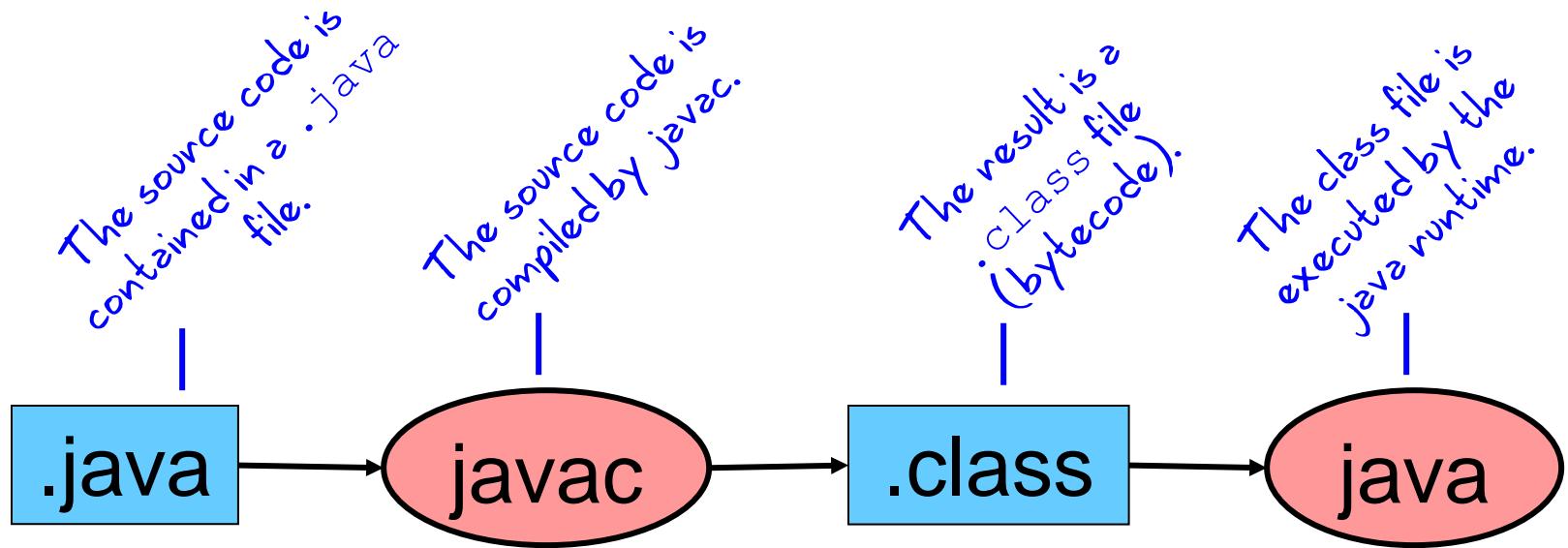
Examining the Installed JDK (Windows Example): The Libraries



Topics

- Introduction to computer programs
- Introduction to the Java language
- Verifying the Java development environment
- Running and testing a Java program

Compiling and Running a Java Program



Compiling a Program

1. Go to the directory where the source code files are stored.
 2. Enter the following command for each .java file you want to compile.
- Syntax:

```
javac SayHello.java
```

- Example:

```
javac SayHello.java
```

Executing (Testing) a Program

1. Go to the directory where the class files are stored.
2. Enter the following for the class file that contains the main method:
 - Syntax:

```
java <classname>
```

- Example: Do not specify .class.

```
java SayHello
```

- Output:

```
Hello World!
```

Output for a Java Program

A Java program can output data in many ways. Here are some examples:

- To a file or database
- To the console
- To a webpage or other user interface

Exercise 2-1

- From a Terminal window, enter `java -version` to see the system's Java version.
- Look for `SayHello.java` in:
`/labs/02-GettingStarted/Exercises/Exercise1`
- Compile it: `javac SayHello.java`
- Run the resulting class file: `java SayHello`
 - Did you see the output?

Quiz

Which of the following is correct? (Choose all that apply.)

- a. javac OrderClass
- b. java OrderClass
- c. javac OrderClass.java
- d. java OrderClass.java

Summary

In this lesson, you should have learned how to:

- Describe the distinction between high-level language and machine code
- Describe what platform-independence means
- Describe how a Java program is compiled and to what format
- Explain what it means to say that Java is an object-oriented language
- Determine the version number of a Java install
- Use the `javac` tool to compile Java source code and the `java` tool to run or test your program

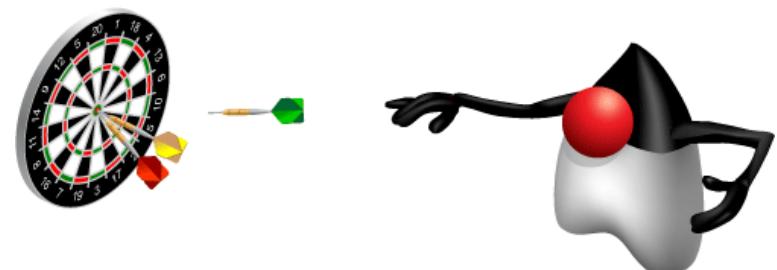


Creating a Java Main Class

Objectives

After completing this lesson, you should be able to:

- Create a Java class
- Write a main method
- Use `System.out.println` to write a String literal to system output

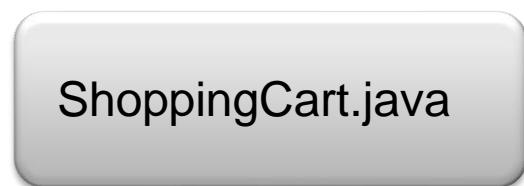


Topics

- Java classes and packages
- The `main` method

Java Classes

A Java class is the building block of a Java application.



Includes code that:

- Allows a customer to add items to the shopping cart
- Provides visual confirmation to the customer

Program Structure

- A class consists of:
 - The class name. Class names begin with a capital letter.
 - The body of the class surrounded with braces { }
 - Data (called fields)
 - Operations (called methods)
- Example:

```
public class Hello {  
    // fields of the class  
    // methods  
}
```

Java is case-sensitive!

Java Packages

- A package provides a namespace for the class.
 - This is a folder in which the class will be saved.
 - The folder name (the package) is used to uniquely identify the class.
 - Package names begin with a lowercase letter.
- Example:

```
package greeting;  
  
public class Hello {  
    // fields and methods here  
}
```

Package name

The class's unique
name is: greeting.Hello

Using the Java Code Console

For the exercises in this course, you use a browser-based Java IDE.

1. Open a browser and enter: <http://localhost:8080/JavaCC>
2. Click the [Lessons](#) link.
3. Click the exercise number for the current lesson.

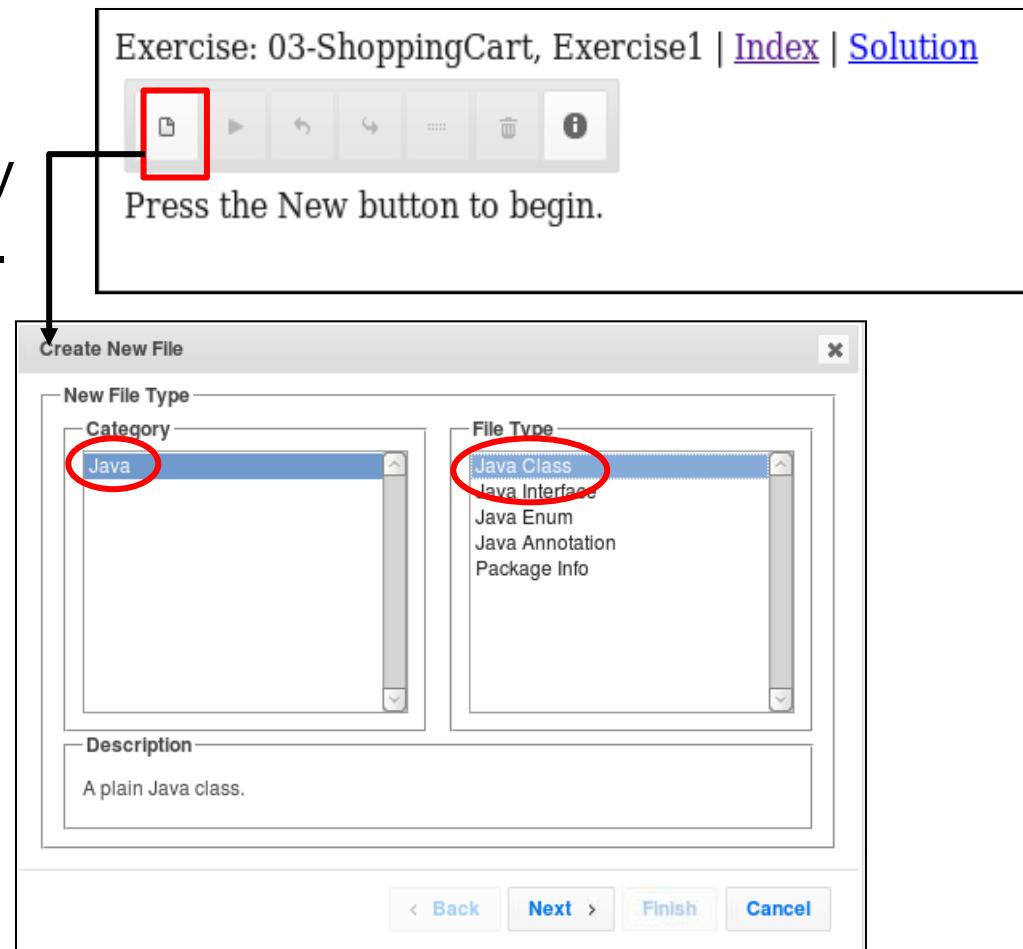


Exercises

- 02-GettingStarted
 - [Exercise1](#)
- 03-ShoppingCart
 - [Exercise1](#)
- 04-Variables
 - [Exercise1](#)
 - [Exercise2](#)
- 05-ConditionsArraysLoops
 - [Exercise1](#)
 - [Exercise2](#)
 - [Exercise3](#)

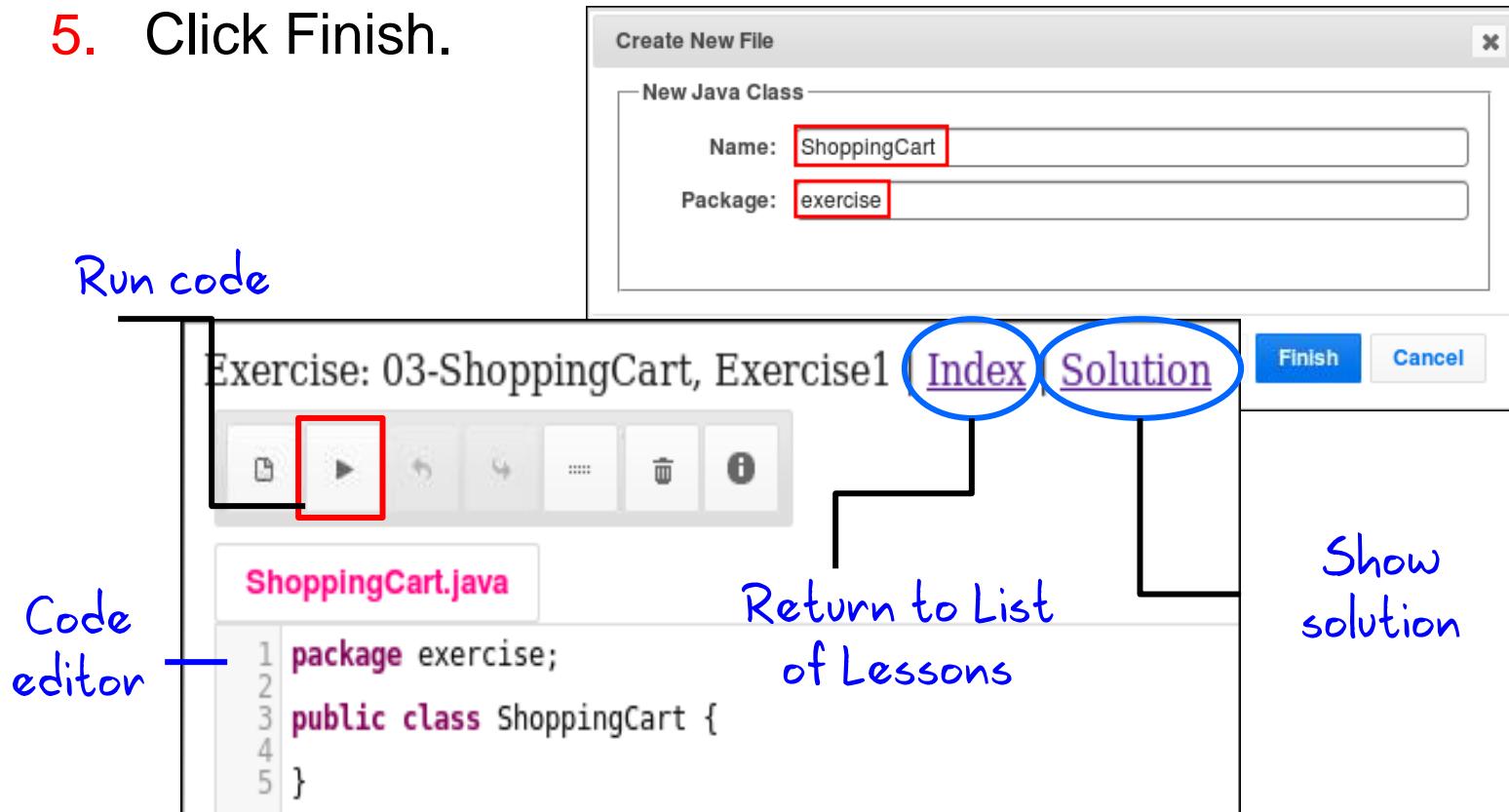
Using the Java Code Console: Creating a New Java Class

1. Click the New button to create a new file.
2. Select the **Java** category and **Java Class** file type.
3. Click Next.



Using the Java Code Console: Creating a New Java Class for an Exercise

4. Enter a class name and a package name.
5. Click Finish.



Exercise 3-1: Creating a Class

In this exercise, you use the Java Code Console to create a new Java Class.

- Click New to create a new class:
 - Class name = ShoppingCart
 - Package name = exercise
- Leave the tabbed view open in the browser because you will modify the code in the next exercise.

Topics

- Java classes
- The main method

The main Method

- It is a special method that the JVM recognizes as the starting point for every Java program.
- The syntax is always the same:

```
public static void main (String args[]) {  
    // code goes here in the code block  
}
```

- It surrounds entire method body with braces `{ }`.

A main Class Example

```
public class Hello {
```

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

```
        // Entry point to the program.
```

```
        // Write code here:
```

```
        System.out.println ("Hello World!");
```

```
}
```

main
method

Class name

Comments

Program
output

Output to the Console

- Syntax:

```
System.out.println (<some string value>);
```

- Example:

```
System.out.println ("This is my message.");
```

String literal

Fixing Syntax Errors

- If you have made a syntax error, the error message appears in the Output panel.
- Common errors:
 - Unrecognized word (check for case-sensitivity error)
 - Missing semicolon
 - Missing close quotation mark
 - Unmatched brace



The screenshot shows a Java code editor window titled "HelloWorld.java". The code is:1 package exercise;
2 public class Hello {
3 System.out.println ("Hello World!");
4 }
5
6 }

```
Below the code editor is an "Output" panel. It contains the text "POST error: error,Internal Server Error". A red circle highlights the closing brace at the end of the code, indicating it is unmatched.
```

Exercise 3-2: Creating a main Method

In this practice, you manually enter a `main` method that prints a message to the console.

- In the code editor, add the `main` method structure to the `ShoppingCart` class.
- In the code block, use a `System.out.println` method to print “Welcome to the Shopping Cart!”
- Click the **Run** button to test program.



A screenshot of an IDE interface. On the left is a code editor window containing the following Java code:

```
1 package exercise;
2
3 public class Hello {
4     public static void main (String args[]){
5         System.out.println("Hello World!");
6     }
7 }
```

Below the code editor is an 'Output' tab, which is highlighted with a red box. The tab has a small 'x' icon next to it. Underneath the tab, the text "Hello World!" is displayed, also enclosed in a red box. A blue arrow points from the text "Example" in the bottom right corner towards the red box around the output text.

Quiz

Which main method syntax is correct?

- a. Public static void main (String[] args) { }
- b. public Static void Main (String[] args) { }
- c. public static void main (String () args) []
- d. public static void main (String[] args) { }

Summary

In this lesson, you should have learned how to:

- Create a class using the Java Code Console
- Create (declare) a Java class
- Define a main method within a class
- Use `System.out.println` to write to the program output
- Run a program in the Java Code Console



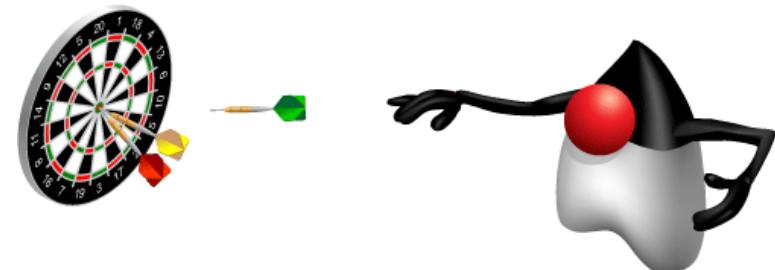
4

Data in a Cart

Objectives

After completing this lesson, you should be able to:

- Describe the purpose of a variable in the Java language
- List and describe four data types
- Declare and initialize `String` variables
- Concatenate `String` variables with the '+' operator
- Make variable assignments
- Declare and initialize `int` and `double` variables
- Modify variable values by using numeric operators
- Override default operator precedence using ()

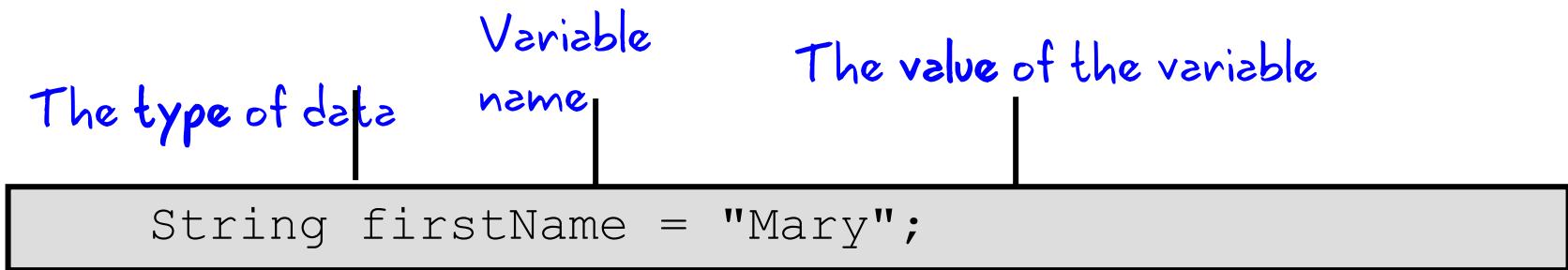


Topics

- Introducing variables
- Working with String variables
- Working with numbers
- Manipulating numeric data

Variables

- A variable refers to something that can change.
 - Variables can be initiated with a value.
 - The value can be changed.
 - A variable holds a specific type of data.



Variable Types

- Some of the types of values a variable can hold:
 - `String` (example: "Hello")
 - `int` (examples: -10, 0, 2, 10000)
 - `double` (examples: 2.00, 99.99, -2042.09)
 - `boolean` (true or false)
- If uninitialized, variables have a default value:
 - `String`: `null`
 - `int`: `0`
 - `double`: `0.0`
 - `boolean`: `false`

Naming a Variable

Guidelines:

- Begin each variable with a lowercase letter. Subsequent words should be capitalized:
 - myVariable
- Names are case-sensitive.
- Names cannot include white space.
- Choose names that are mnemonic and that indicate to the casual observer the intent of the variable.
 - outOfStock (a boolean)
 - itemDescription (a String)

Uses of Variables

- Holding data used within a method:

```
String name = "Sam" ;  
double price = 12.35;  
boolean outOfStock = true;
```

- Assigning the value of one variable to another:

```
String name = name1;
```

- Representing values within a mathematical expression:

```
total = quantity * price ;
```

- Printing the values to the screen:

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

Topics

- Introducing variables
- Working with String variables
- Working with numbers
- Manipulating numeric data

Variable Declaration and Initialization

- Syntax :

```
type identifier [= value];
```

- Examples:
 - String customer;
 - String name, city;
 - String address = "123 Oak St";
 - String country = "USA", state = "CO";
- Variable declared*
- Two variables declared*
- Variable declared and initialized*
- Two variables declared and initialized*

String Concatenation

- String variables can be combined using the '+' operator.
 - stringVariable1 + stringVariable2
 - stringVariable1 + "String literal"
 - stringVariable1 + "String literal" + stringVariable2

- Example:

```
String greet1 = "Hello";  
String greet2 = "World";  
String message = greet1 + " " + greet2 + "!";  
String message = greet1 + " " + greet2 + " " + 2014 +"!";
```

String Concatenation Output

You can concatenate **String** variables within a method call:

```
System.out.println(message);  
System.out.println(greet1 + " " + greet2 + "!");
```

Output:

```
Hello World!  
Hello World!
```

Exercise 4-1: Using String Variables

In this exercise, you declare, initialize, and concatenate String variables and literals.

Exercise: 04-Variables, Exercise1 | [Index](#) | [Solution](#)

ShoppingCart.java

```
1 package ex04_1_exercise;
2
3 public class ShoppingCart {
4
5     public static void main(String[] args) {
6         // Declare and initialize String variables. Do not initialize message yet.
7
8
9
10
11
12
13
14     // Assign the message variable
15
16
17     // Print and run the code
18
19
20 }
21
22 }
```

Exercise 4-1:

1. Declare and initialize 2 String variables: custName and itemDesc
2. Declare a String variable called message. Do not initialize it.
3. Assign the message variable with a concatenation of the custName and itemDesc.
Include a String literal that results in a complete sentence.
(example: "Mary Smith wants to purchase a Shirt")
4. Print the message to the System output.
5. Run the code.



Quiz

Which of the following variable declarations and/or initializations are correct?

- a. int count = 5; quantity = 2;
- b. string name, label;
- c. boolean complete = "false";
- d. boolean complete = true;

Topics

- Introducing variables
- Working with String variables
- Working with numbers
- Manipulating numeric data

int and double Values

- int variables hold whole number values between:
 - -2,147,483,648
 - 2,147,483,647
 - Examples: 2, 1343387, 1_343_387
- double variables hold larger values containing decimal portions.
 - Use when greater accuracy is needed.
 - Examples: 987640059602230.7645 , -1111, 2.1E12

Initializing and Assigning Numeric Values

- int variables:

- int quantity = 10;
 - int quantity = 5.5;



Compilation fails!

- double variables:

- double price = 25.99;
 - double price = 75;



Run time will
interpret as 75.0.

Topics

- Introducing variables
- Working with String variables
- Working with numbers
- Manipulating numeric data

Standard Mathematical Operators

Purpose	Operator	Example	Comments
Addition	+	sum = num1 + num2; If num1 is 10 and num2 is 2, sum is 12.	
Subtraction	-	diff = num1 - num2; If num1 is 10 and num2 is 2, diff is 8.	
Multiplication	*	prod = num1 * num2; If num1 is 10 and num2 is 2, prod is 20.	
Division	/	quot = num1 / num2; If num1 is 31 and num2 is 6, quot is 5.	Division by 0 returns an error. The remainder portion is discarded.

Increment and Decrement Operators (++ and --)

The long way:

```
age = age + 1;
```

or

```
count = count - 1;
```

The short way:

```
age++;
```

or

```
count--;
```

Operator Precedence

Here is an example of the need for rules of precedence.

Is the answer to the following problem 34 or 9?

```
int c = 25 - 5 * 4 / 2 - 10 + 4;
```

Operator Precedence

Rules of precedence:

1. Operators within a pair of parentheses
2. Increment and decrement operators (++ or --)
3. Multiplication and division operators, evaluated from left to right
4. Addition and subtraction operators, evaluated from left to right

Using Parentheses

Examples:

```
int c = (((25 - 5) * 4) / (2 - 10)) + 4;  
int c = ((20 * 4) / (2 - 10)) + 4;  
int c = (80 / (2 - 10)) + 4;  
int c = (80 / -8) + 4;  
int c = -10 + 4;  
int c = -6;
```

Exercise 4-2: Using and Manipulating Numbers

In this exercise, you declare, initialize, and concatenate String variables and literals.



Quiz

Which of the following statements are correct Java code?

- a. int count = 11.4;
- b. double amount = 11.05;
- c. int cost = 133_452_667;
- d. double total = 1.05 * amount;

Quiz

Given:

```
String name = "Bob";  
String msg;  
int num = 3;
```

Which of the following statements correctly assigns the value
“Bob wrote 3 Java programs.” to the msg variable?

- a. msg = name + " wrote " + num " Java programs.;"
- b. msg = name + " wrote " + 3 + " Java programs.;"
- c. msg = "Bob wrote "+ (2+1) + " Java programs.;"
- d. msg = name + " wrote " + 2+1 + " Java
programs.;"

Summary

In this lesson, you should have learned how to:

- Describe the purpose of a variable in the Java language
- List and describe four data types
- Declare and initialize `String` variables
- Concatenate `String` variables with the '+' operator
- Make variable assignments
- Declare and initialize `int` and `double` variables
- Modify numeric values by using operators
- Override default operator precedence using ()

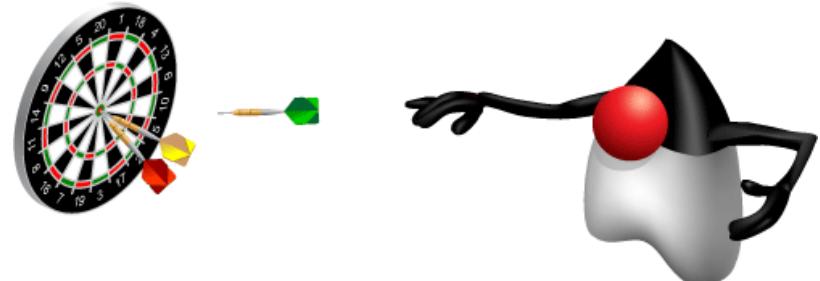


Managing Multiple Items

Objectives

After completing this lesson, you should be able to:

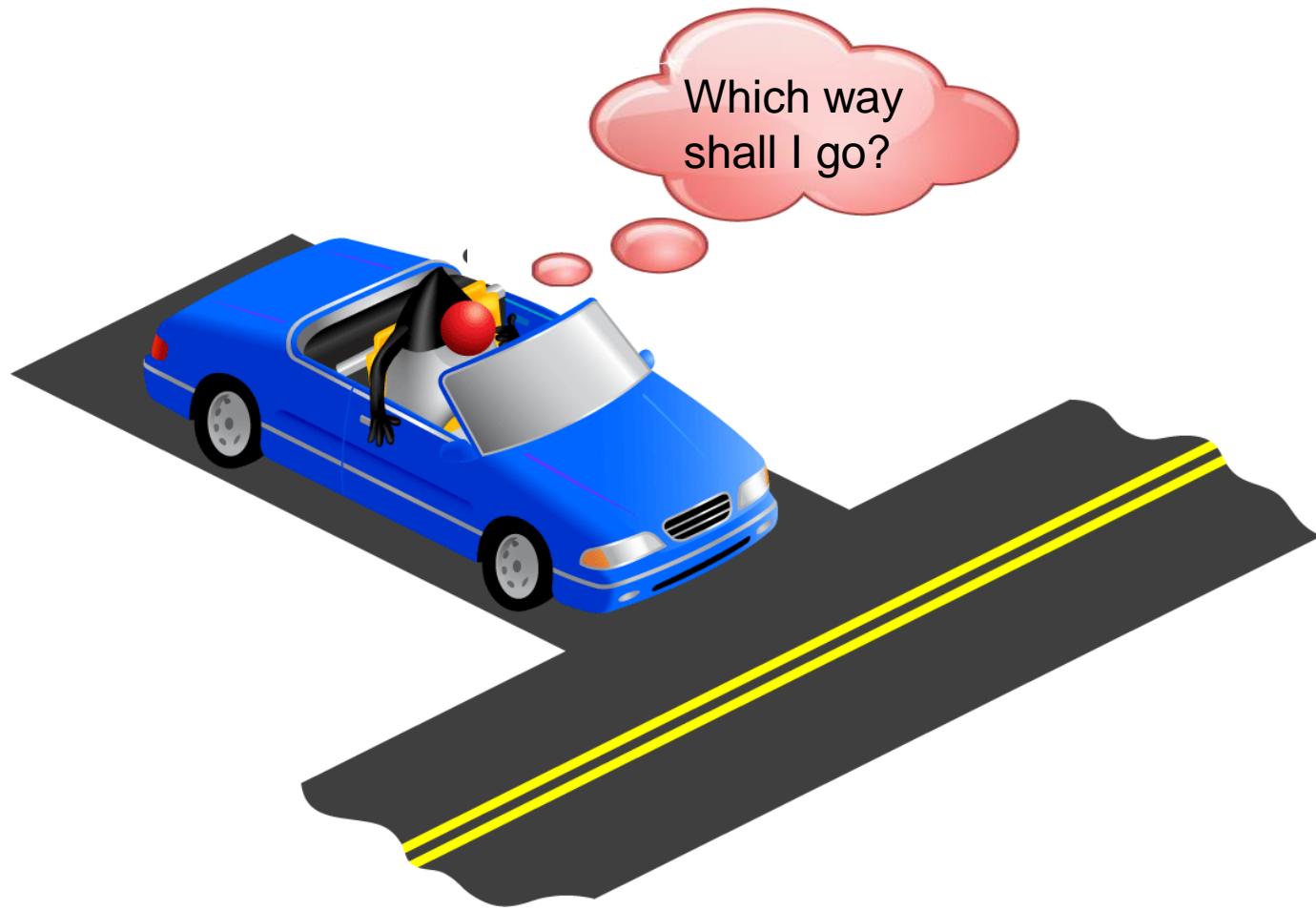
- Explain what a boolean expression is
- Create a simple `if/else` statement
- Describe the purpose of an array
- Declare and initialize a `String` or `int` array
- Access the elements of an array
- Explain the purpose of a `for` loop
- Iterate through a `String` array using a `for` loop



Topics

- Working with conditions
- Working with an array of items
- Processing an array of items

Making Decisions



The if/else Statement

```
if ( <some condition is true> ) {  
    // do something  
}  
  
else {  
    // do something different  
}
```

boolean expression

if block

else block

The diagram illustrates the structure of an if/else statement. It starts with the keyword 'if' followed by a condition enclosed in parentheses, which is highlighted with a blue box and labeled 'boolean expression'. A bracket labeled 'if block' spans from the opening brace of the if-statement to the closing brace. Following the 'else' keyword, another bracket labeled 'else block' spans from its opening brace to its closing brace. The code itself is written in a monospaced font.

Boolean Expressions

Review:

- boolean data type has only two possible values:
 - true
 - false

A boolean expression is a combination of variables, values, and operators that evaluate to true or false.

- length $>$ 10;
- size \leq maxSize;
- total \equiv (cost * price);

Relational operators

Relational Operators

Condition	Operator	Example
Is equal to	<code>==</code>	<code>int i=1; (i == 1)</code>
Is not equal to	<code>!=</code>	<code>int i=2; (i != 1)</code>
Is less than	<code><</code>	<code>int i=0; (i < 1)</code>
Is less than or equal to	<code><=</code>	<code>int i=1; (i <= 1)</code>
Is greater than	<code>></code>	<code>int i=2; (i > 1)</code>
Is greater than or equal to	<code>>=</code>	<code>int i=1; (i >= 1)</code>

Examples

Sometimes there is a quicker way to meet your objective.
boolean expressions can be used in many ways.

```
24     int attendees = 4;  
25     boolean largeVenue;  
26  
27     // if statement example  
28     if (attendees >= 5){  
29         largeVenue = true;  
30     }  
31     else {  
32         largeVenue = false;  
33     }  
34  
35     // same outcome with less code  
36     largeVenue = (attendees >= 5);
```

Assign a boolean by
using an if
statement.

Assign the boolean
directly from the
boolean expression.

Exercise 5-1: Using if Statements

In this exercise, you use an `if` and an `if/else` statement:

- Declare a boolean, `outOfStock`.
- `if quantity > 1`
 - Change the `message` variable to indicate plural
- `if/else:`
 - `if item is out of stock:`
 - Inform the user that the item is unavailable
 - `else`
 - Print the `message`
 - Print the total cost



Quiz

What is the purpose of the `else` block in an `if/else` statement?

- a. To contain the remainder of the code for a method
- b. To contain code that is executed when the expression in an `if` statement is false
- c. To test if an expression is false

Topics

- Working with conditions
- Working with an array of items
- Processing an array of items

What If There Are Multiple Items in the Shopping Cart?

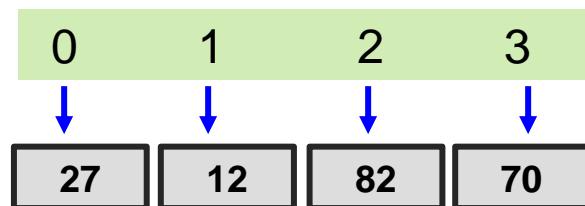
```
01      // Without an array
02      String itemDesc1 = "Shirt"; 
03      String itemDesc2 = "Trousers";
04      String itemDesc3 = "Scarf";
05      // Using an array 
06
07      String[] items = {"Shirt", "Trousers", "Scarf"};
```

Not realistic if
100s of items!

Much better!

Introduction to Arrays

- An array is an indexed container that holds a set of values of a single type.
- Each item in an array is called an *element*.
- Each element is accessed by its numerical index.
- The index of the first element is 0 (zero).
 - A four-element array has indices: 0, 1, 2, 3.



Array Examples

Array of int types

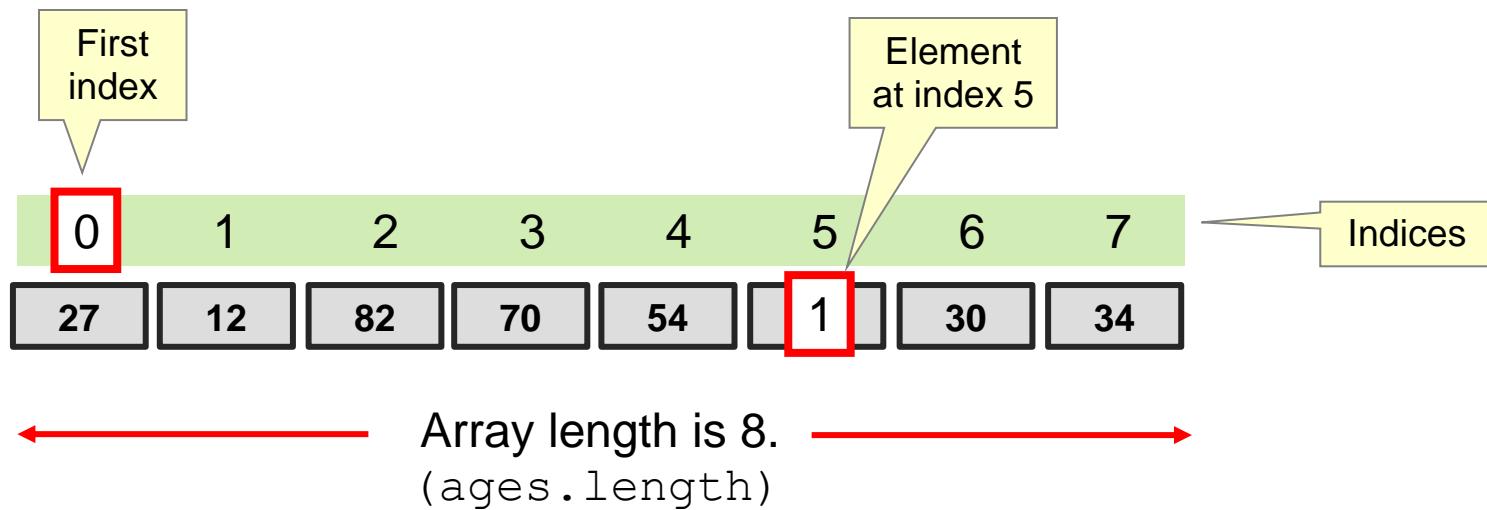


Array of String types

Hugh Mongus
Aaron Datires
Stan Ding
Albert Kerkie
Carrie DeKeys
Walter Mellon
Hugh Morris
Moe DeLawn

Array Indices and Length

The `ages` array has eight elements.



Declaring and Initializing an Array

- Syntax:

```
type[] arrayIdentifier = {comma-separated list of values};
```

- Declare arrays of types String and int:

```
String[] names = {"Mary", "Bob", "Carlos"};
```

```
int[] ages = {25, 27, 48};
```

All in one
line

Declaring and Initializing an Array

- Examples:

```
1  int[] ages = new int[3];
2  ages[0] = 19;          Multistep
3  ages[1] = 42;          approach
4  ages[2] = 92;
5
6  String[] names = new String[3];    Multistep
7  names[0] = "Mary";                approach
8  names[1] = "Bob";
9  names[2] = "Carlos";
```

Accessing Array Elements

- Get values from the ages array:

```
int[] ages = {25, 27, 48};

int myAge = ages[0];
int yourAge = ages[1];
System.out.println("My age is " + ages[0]);
```

- Set values from the names array:

```
String[] names = {"Mary", "Bob", "Carlos"};

names[0] = "Gary";
names[1] = "Rob";
```

Exercise 5-2: Using an Array

In this exercise, you declare and initialize a `String` array to hold names. Then you experiment with accessing the array:

- Declare a `String` array, `names`, and initialize it with four `String` values.
- Print the number of items the customer wants to buy.
- Print one of the array elements.



Quiz

Why does the following code not compile? Select all that apply.

```
int[] lengths = {2, 4, 3.5, 0, 40.04};
```

- a. lengths cannot be used as an array identifier.
- b. All of the element values should have the same format (all using double values, or all using int values).
- c. The array was declared to hold int values. double values are not allowed.

Quiz

Given the following array declaration, which of the following statements are true?

```
int[] classSize = {5, 8, 0, 14, 194};
```

- a. `classSize[0]` is the reference to the first element in the array.
- b. `classSize[5]` is the reference to the last element in the array.
- c. There are 5 integers in the `classSize` array.
- d. `classSize.length = 5`

Topics

- Working with conditions
- Working with an array of items
- Processing an array of items

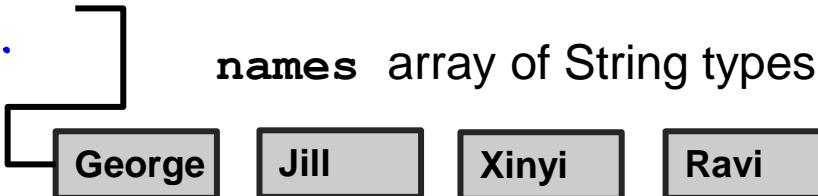
Loops

Loops are used in programs to repeat blocks of statements

- Until an expression is false
 - or
- For a specific number of times:
 - I want to print each element of an array.
 - I want to print each element of an `ArrayList`. (The `ArrayList` class is covered in the lesson titled “Working with Arrays, Loops, and Dates.”)

Processing a String Array

Loop accesses each element in turn.



```
for (String name : names ) {  
    System.out.println("Name is " + name);  
}
```

Each iteration returns the next element of the array.

Output:

```
Name is George  
Name is Jill  
Name is Xinyi  
Name is Ravi
```

Using break with Loops

break example:

```
01 int passmark = 12;
02 boolean passed = false;
03 int[] scores = {4, 6, 2, 8, 12, 35, 9};
04 for (int unitScore : scores) {
05     if (unitScore >= 12) {
06         passed = true;
07         break;
08     }
09 }
10 System.out.println("At least one passed? " +passed);
```

No need to go through the loop again, so use break.

Output:

```
At least one passed? true
```

Exercise 5-3: Using a Loop to Process an Array

In this exercise, you loop through an array called `itemPrices` to print a message indicating each item price.



Quiz

Given the following code,

```
int[] sizes = {4, 18, 5, 20};  
for (int size : sizes) {  
    if (size > 16) {break;}  
    System.out.println("Size: "+size + ", ");  
}
```

which option below shows the correct output?

- a. Size: 4,
- b. Size: 4
- c. Size: 4,
 Size: 5,
- d. There is no output.

Summary

In this lesson, you should have learned how to:

- Use a boolean expression
- Create a simple `if/else` block
- Describe the purpose of an array
- Declare and initialize a `String` or `int` array
- Access the elements of an array
- Explain the purpose of a `for` loop
- Iterate through a `String` Array using a `for` loop



Play Time!

Basic.05 is most important.

Play **Basic Puzzles 1 through 5** before the lesson titled
“Describing Objects and Classes.”

Your Goal: Design a solution that deflects the ball to Duke.

Consider the following:

What happens when you put a triangle wall or simple wall icon on
the blue wheel?



About Java Puzzle Ball

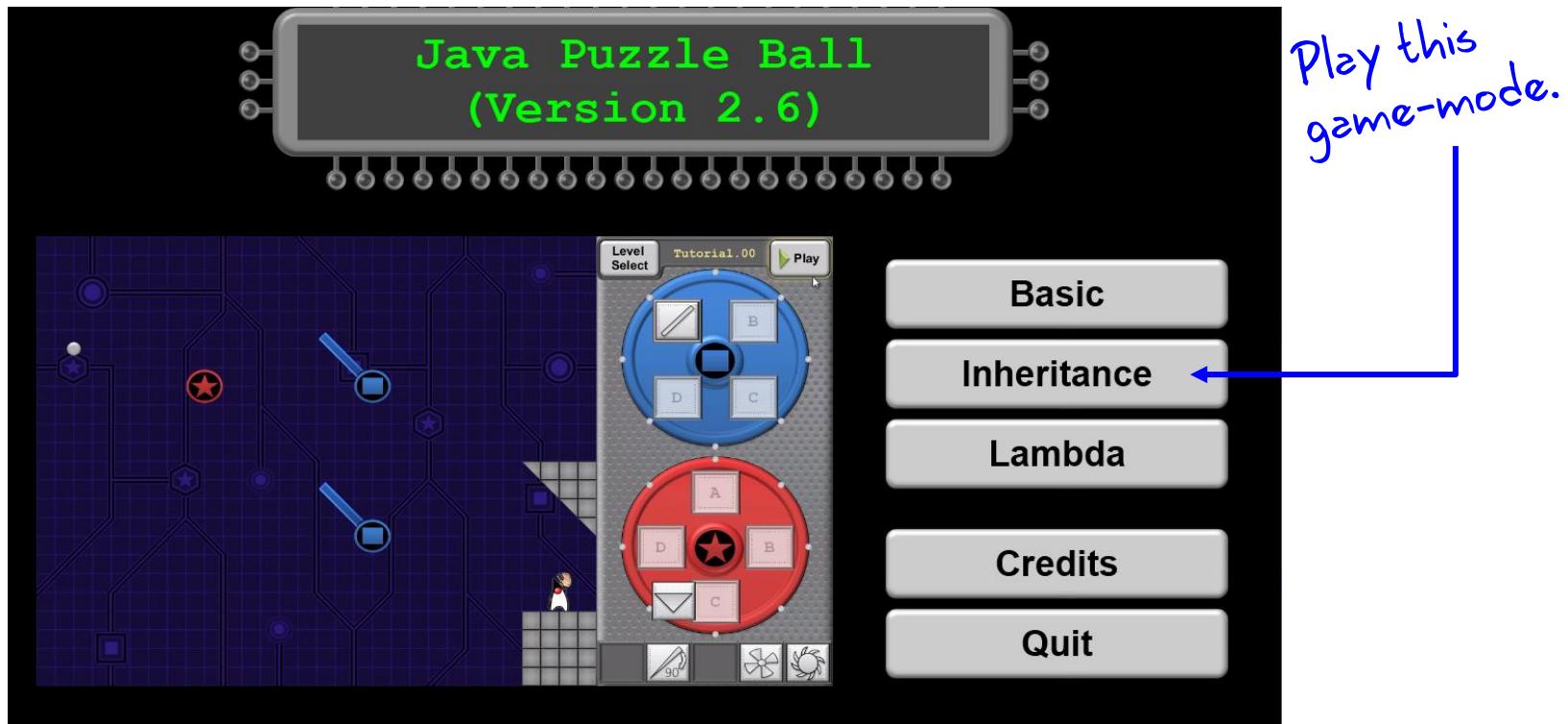
- It is used throughout the course.
- Play a set of puzzles.
- Become familiar with the game mechanics.
- Consider a question as you play.
- The lesson titled “Describing Objects and Classes” debriefs on what you have observed.
- Apply your observations to understand Java concepts.



Tips



- You must have Java 8 installed to run the game.
- The game may perform better on your personal machine.





Describing Objects and Classes

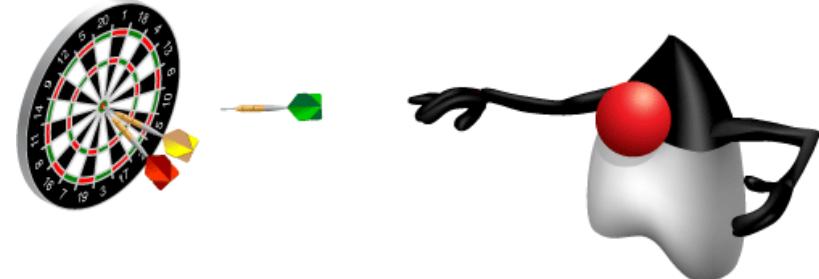
Interactive Quizzes



Objectives

After completing this lesson, you should be able to:

- List the characteristics of an object
- Define an object as an instance of a class
- Instantiate an object and access its fields and methods
- Describe how objects are stored in memory
- Instantiate an array of objects
- Describe how an array of objects is stored in memory
- Declare and instantiate an object as a field
- Use the NetBeans IDE to create and test Java classes



Topics

- Describing objects and classes
- Defining fields and methods
- Declaring, instantiating, and using objects
- Working with object references
- Doing more with arrays
- Introducing NetBeans IDE
- Introducing the soccer league use case

Java Puzzle Ball

Have you played **Basic Puzzle 5**?

Consider the following:

What happens when you put a triangle wall or simple wall icon on the blue wheel?

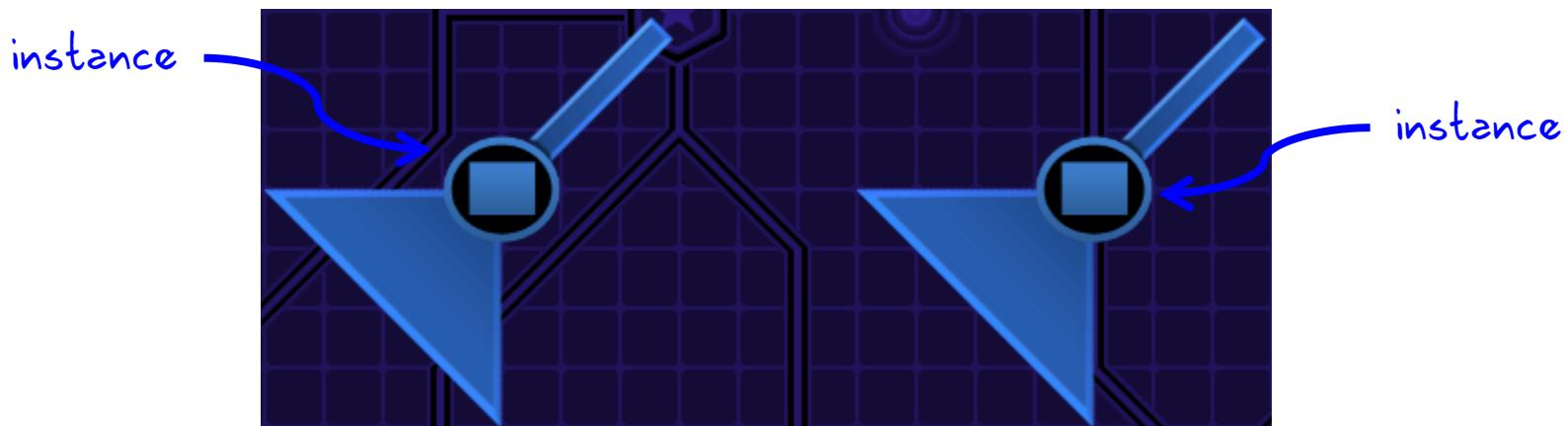


Java Puzzle Ball Debrief



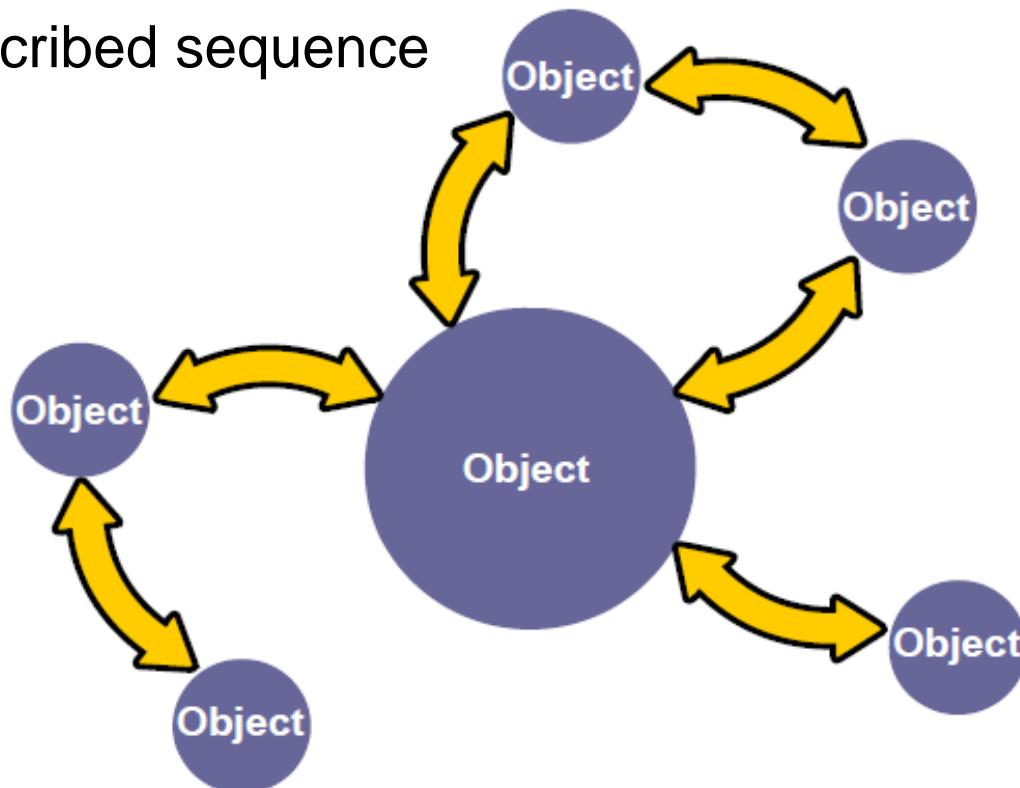
What happens when you put a triangle wall or simple wall icon on a blue wheel?

- A wall appears on every **instance** of a blue bumper **object**.
- Walls give bumpers **behaviors** that deflect and interact with the ball.
- All blue bumper instances share these same behaviors.

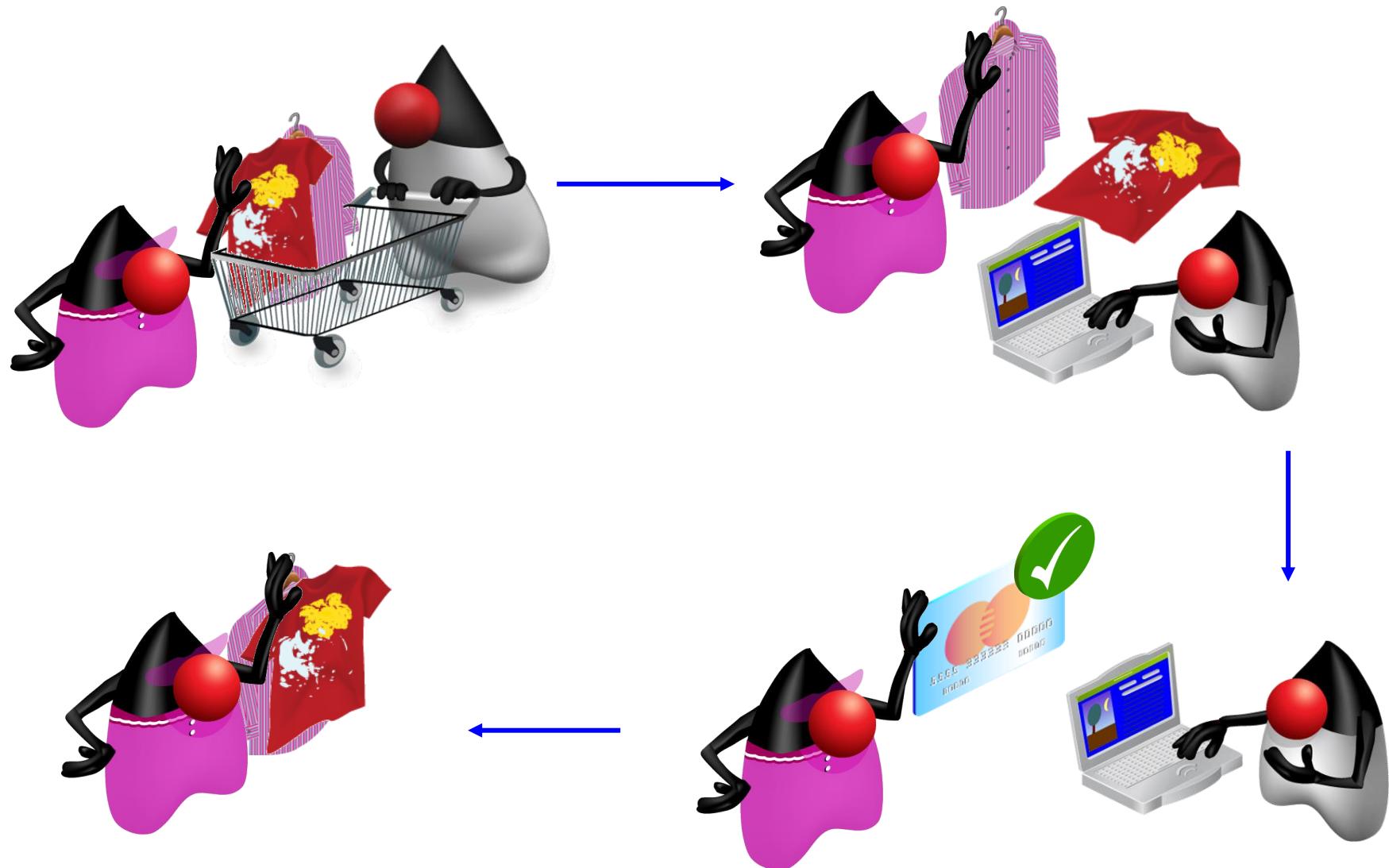


Object-Oriented Programming

- Interaction of objects
- No prescribed sequence



Duke's Choice Order Process



Characteristics of Objects

Objects are physical or conceptual.

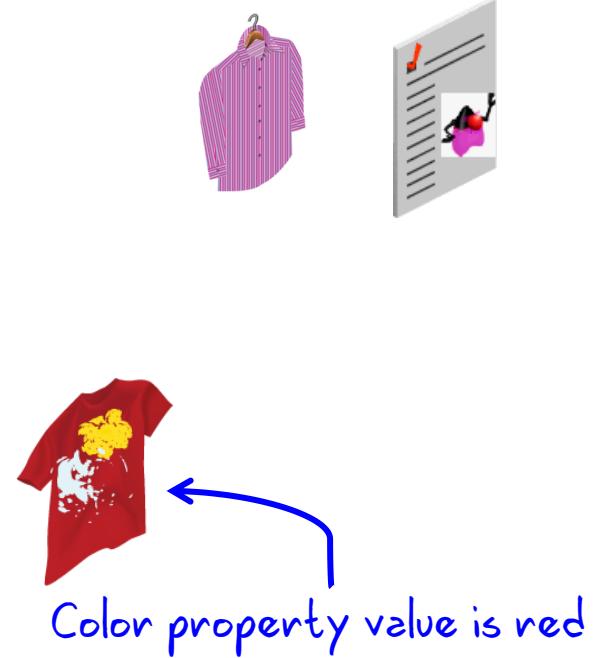
- Objects have **properties**:

- Size
 - Shape
 - Name
 - Color



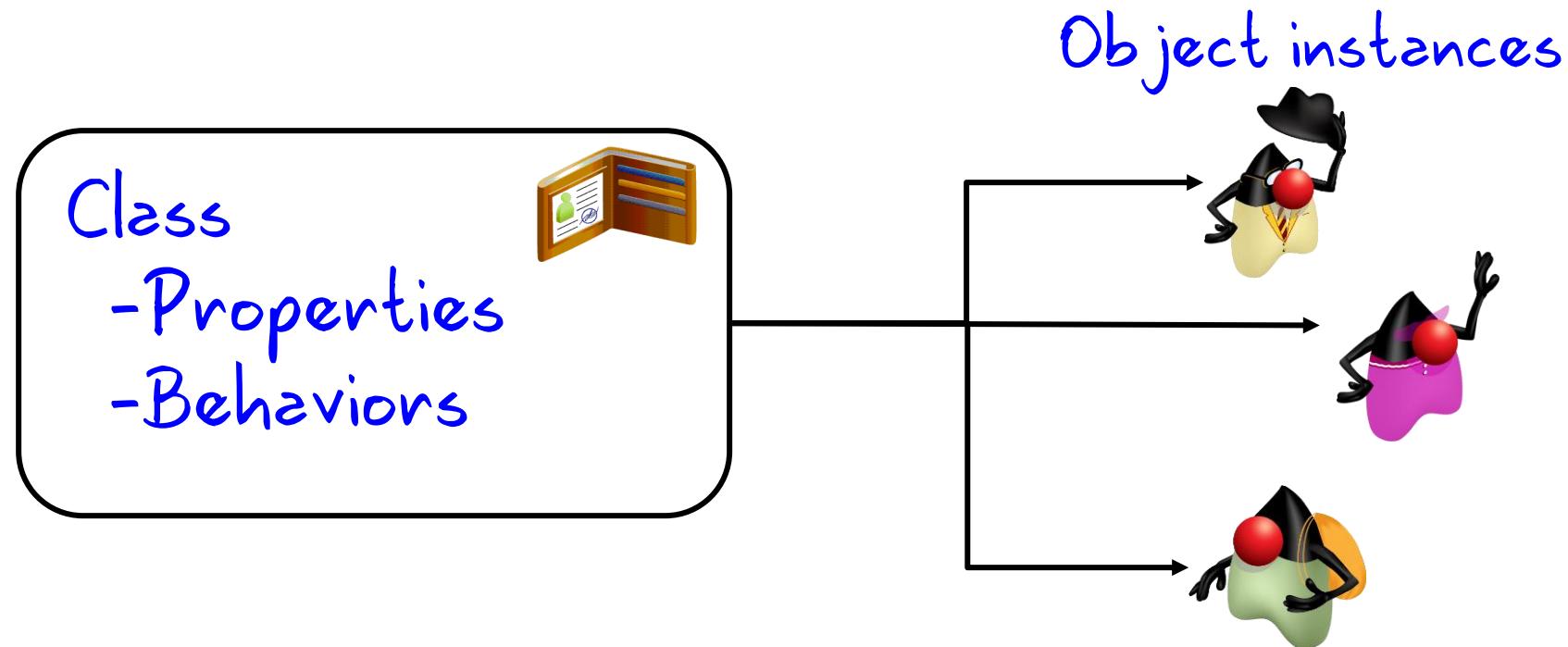
- Objects have **behaviors**:

- Shop
 - Put item in cart
 - Pay



Classes and Instances

- A class:
 - Is a blueprint or recipe for an object
 - Describes an object's properties and behaviors
 - Is used to create object instances



Quiz

Which of the following statements is true?

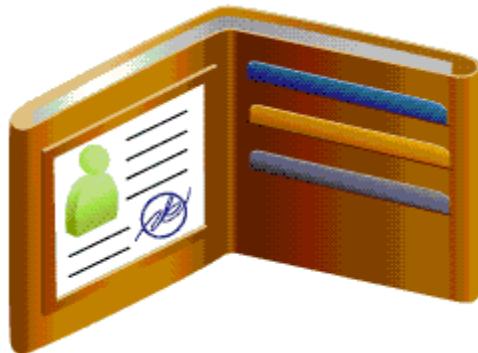
- a. An object is a blueprint for a class.
- b. An object and a class are exactly the same.
- c. An object is an instance of a class.
- d. A class is an instance of an object.



Topics

- Describing objects and classes
- Defining fields and methods
- Declaring, instantiating, and using objects
- Working with object references
- Doing more with arrays
- Introducing NetBeans IDE
- Introducing the soccer league use case

The Customer Properties and Behaviors



Properties:

- Name
- Address
- Age
- Order number
- Customer number

Behaviors:

- Shop
- Set Address
- Add item to cart
- Ask for a discount
- Display customer details

The Components of a Class

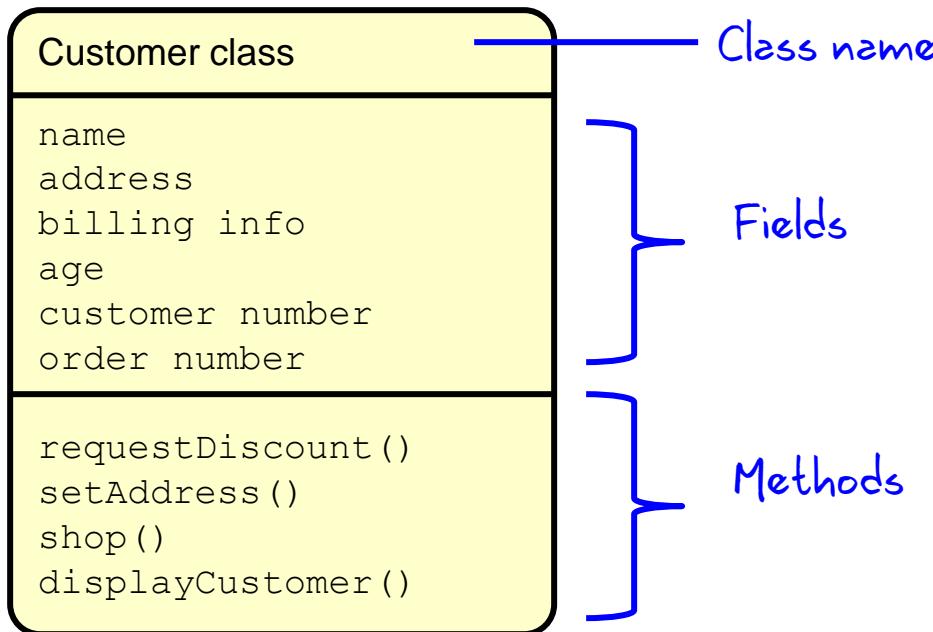
Class declaration

```
1 public class Customer {  
2     public String name = "Junior Duke";  
3     public int    custID = 1205;  
4     public String address;  
5     public int    orderNum;  
6     public int    age;  
7  
8     public void displayCustomer(){  
9         System.out.println("Customer: "+name);  
10    }  
11 }
```

Fields
(Properties)
(Attributes)

Methods
(Behaviors)

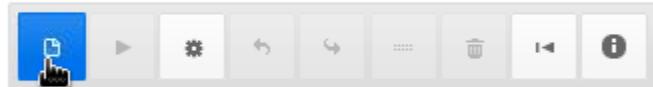
Modeling Properties and Behaviors



Exercise 6-1: Creating the Item Class

In this exercise, you create the Item class and declare public fields for ID (int), descr, quantity (int), and price (double).

Exercise: 06-ObjectsClasses, Exercise1 | [Index](#) | [Solution](#)



Press the New button to begin.

Exercise 6-1

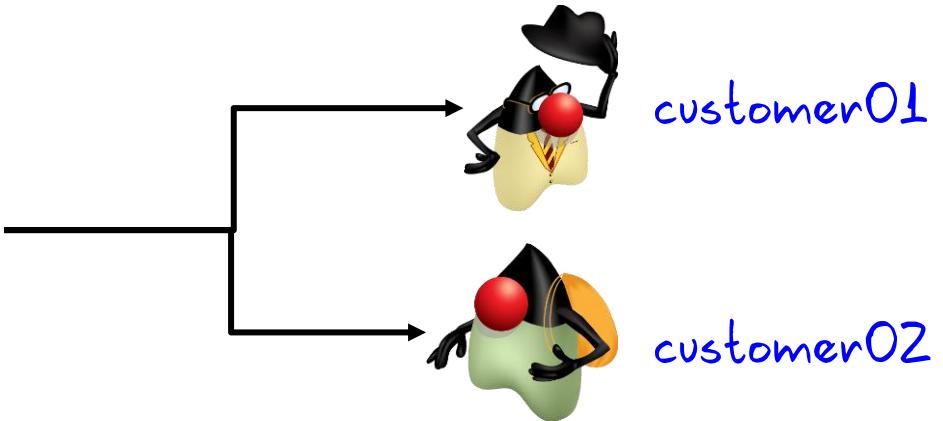
1. Create the Item class as a plain "Java class".
2. Declare public fields for ID (int), descr (String), quantity (int), price (double).



Topics

- Describing objects and classes
- Defining fields and methods
- Declaring, instantiating, and using objects
- Working with object references
- Doing more with arrays
- Introducing NetBeans IDE
- Introducing the soccer league use case

Customer Instances



```
public static void main(String[] args) {  
  
    Customer customer01 = new Customer();  
    Customer customer02 = new Customer();  
  
    customer01.age = 40;  
    customer02.name = "Duke";  
  
    customer01.displayCustomer();  
    customer02.displayCustomer();  
}  
}
```

- } Create new instances (instantiate).
- } Fields are accessed.
- } Methods are called.

Object Instances and Instantiation Syntax

The syntax is:

<class name> variable = new <class name>()

variable becomes a reference
to that object.

The new keyword creates
(instantiates) a new instance.

```
public static void main(String[] args) {  
  
    Customer customer01 = new Customer();      //Declare and instantiate  
  
    Customer customer02;                      //Declare the reference  
    customer02 = new Customer();                //Then instantiate  
  
    new Customer();                          //Instantiation without a reference  
                                            //We can't use this object later  
                                            //without knowing how to reference it.  
}  
}
```

The Dot (.) Operator

Follow the reference variable with a dot operator (.) to access the fields and methods of an object.

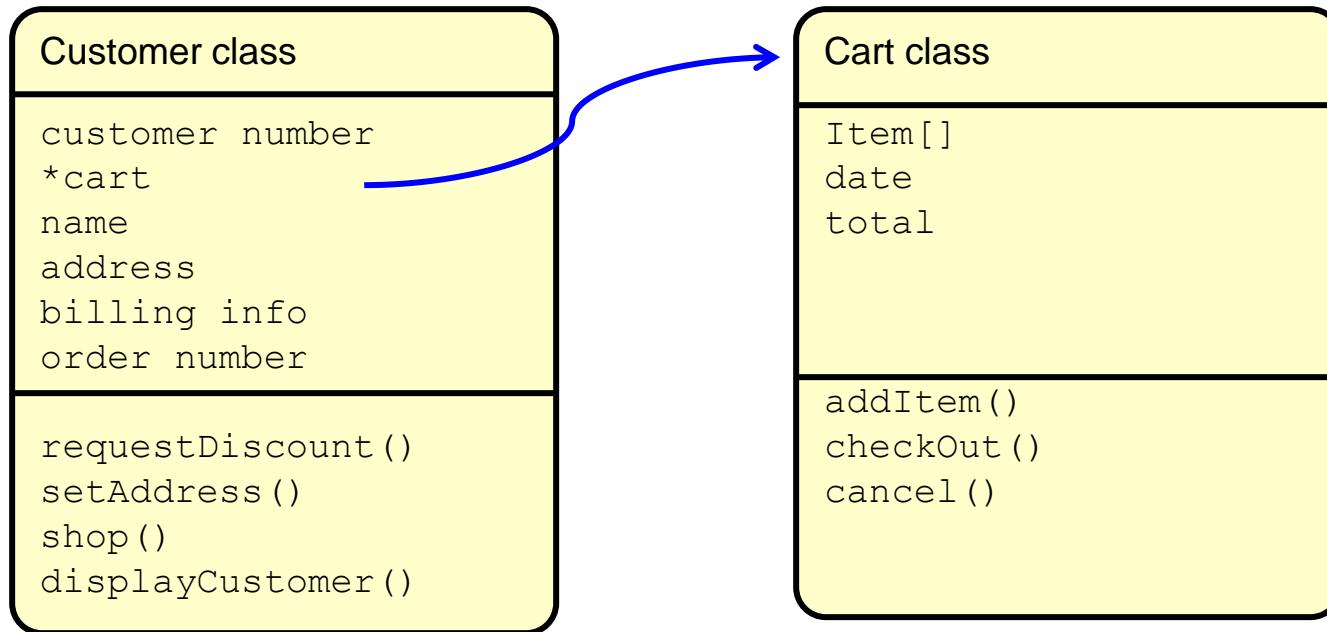
Customer class

name
address
billing info
age
customer number
order number

requestDiscount()
setAddress()
shop()
displayCustomer()

```
public static void main(String[] args) {  
  
    Customer customer01 = new Customer();  
  
    //Accessing fields  
    System.out.println(customer01.name);  
    customer01.age = 40;  
  
    //Calling methods  
    customer01.requestDiscount();  
    customer01.displayCustomer();  
}  
}
```

Objects with Another Object as a Property



```
public static void main(String[] args){  
  
    Customer customer01 = new Customer();  
    customer01.cart.cancel();           //How to access methods of an  
                                     //object within another object  
}
```

Quiz

Which of the following lines of code instantiates a Boat object and assigns it to a sailBoat object reference?

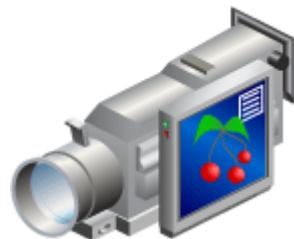
- a. Boat sailBoat = new Boat();
- b. Boat sailBoat;
- c. Boat = new Boat()
- d. Boat sailBoat = Boat();



Topics

- Describing objects and classes
- Defining fields and methods
- Declaring, instantiating, and using objects
- **Working with object references**
- Doing more with arrays
- Introducing NetBeans IDE
- Introducing the soccer league use case

Accessing Objects by Using a Reference



The camera is like the object that is accessed using a reference.

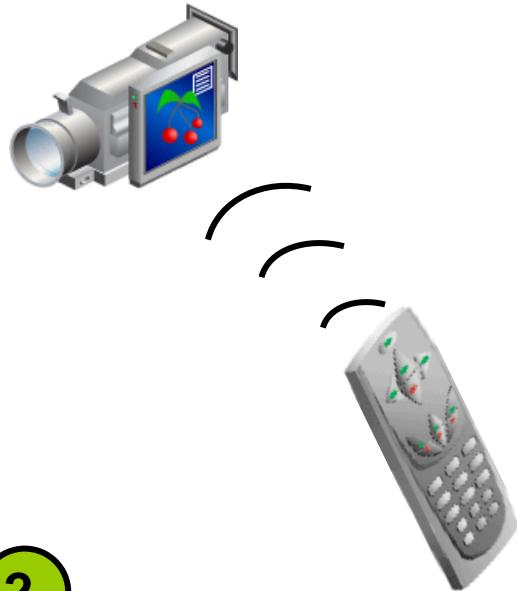


The remote is like the reference used to access the camera.

Working with Object References

1

Pick up the remote to gain access to the camera.



2

Press the remote's controls to have camera do something.

1

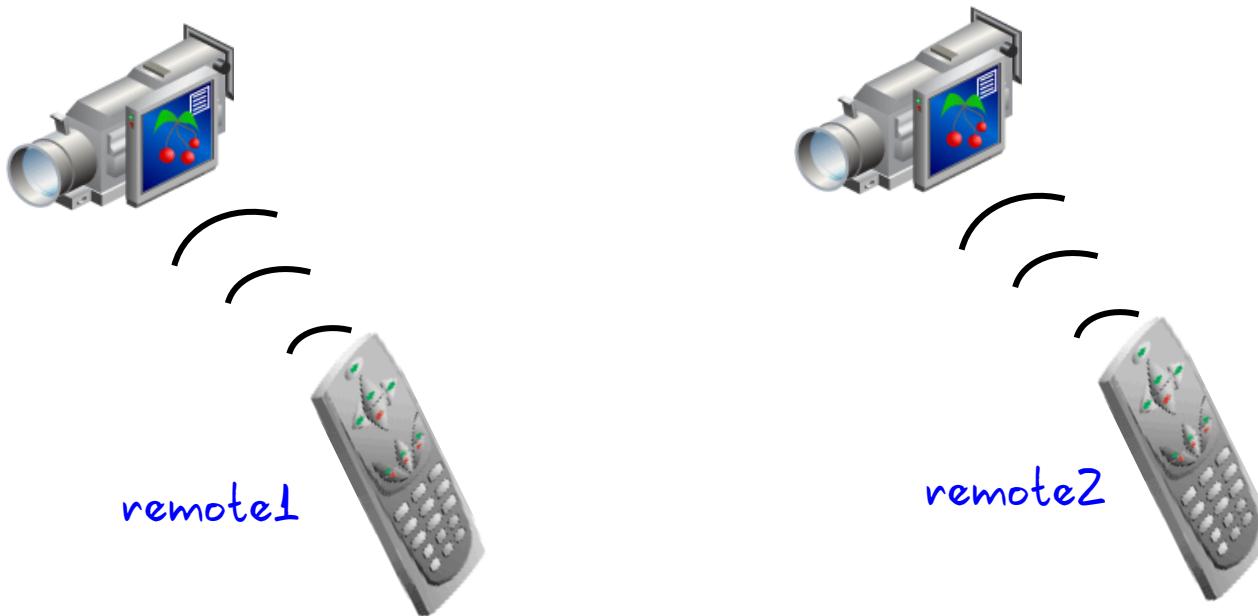
Create a Camera object and get a reference to it.

```
11 Camera remotel;  
12  
13 remotel = new Camera();  
14  
15 remotel.play();
```

2

Call a method to have the Camera object do something.

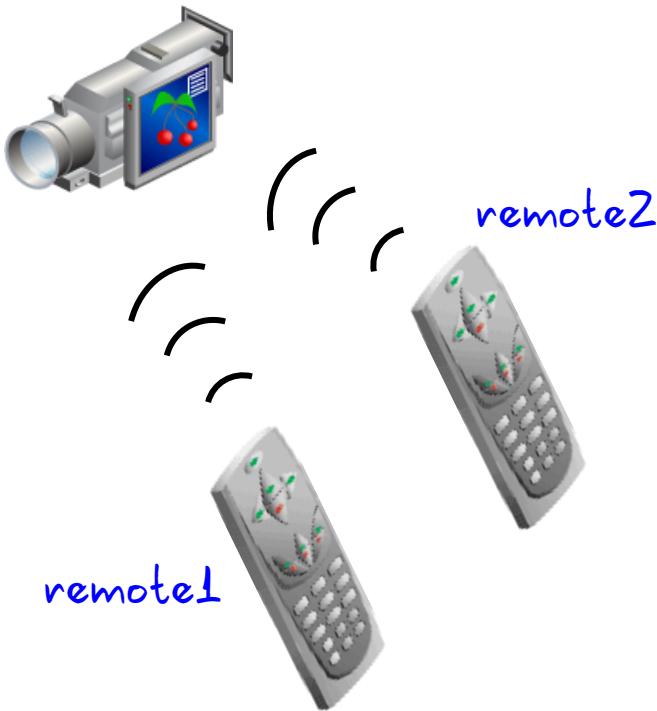
Working with Object References



```
12 Camera remote1 = new Camera();  
13  
14 Camera remote2 = new Camera();  
15  
16 remote1.play();  
17  
18 remote2.play();
```

There are two Camera objects.

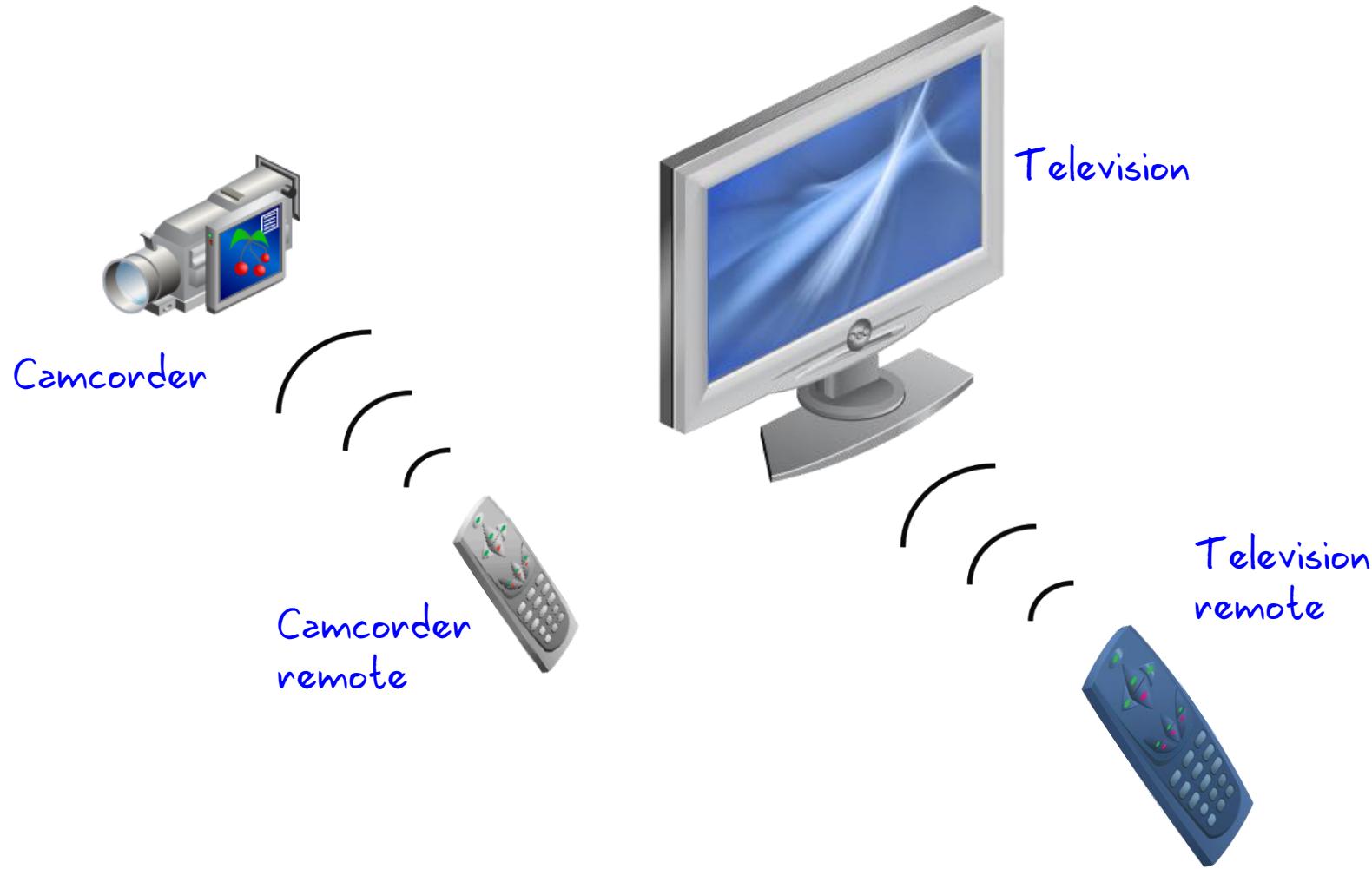
Working with Object References



There is only one Camera object.

```
12 Camera remote1 = new Camera();  
13  
14 Camera remote2 = remote1;  
15  
16 remote1.play();  
17  
18 remote2.stop();
```

References to Different Objects



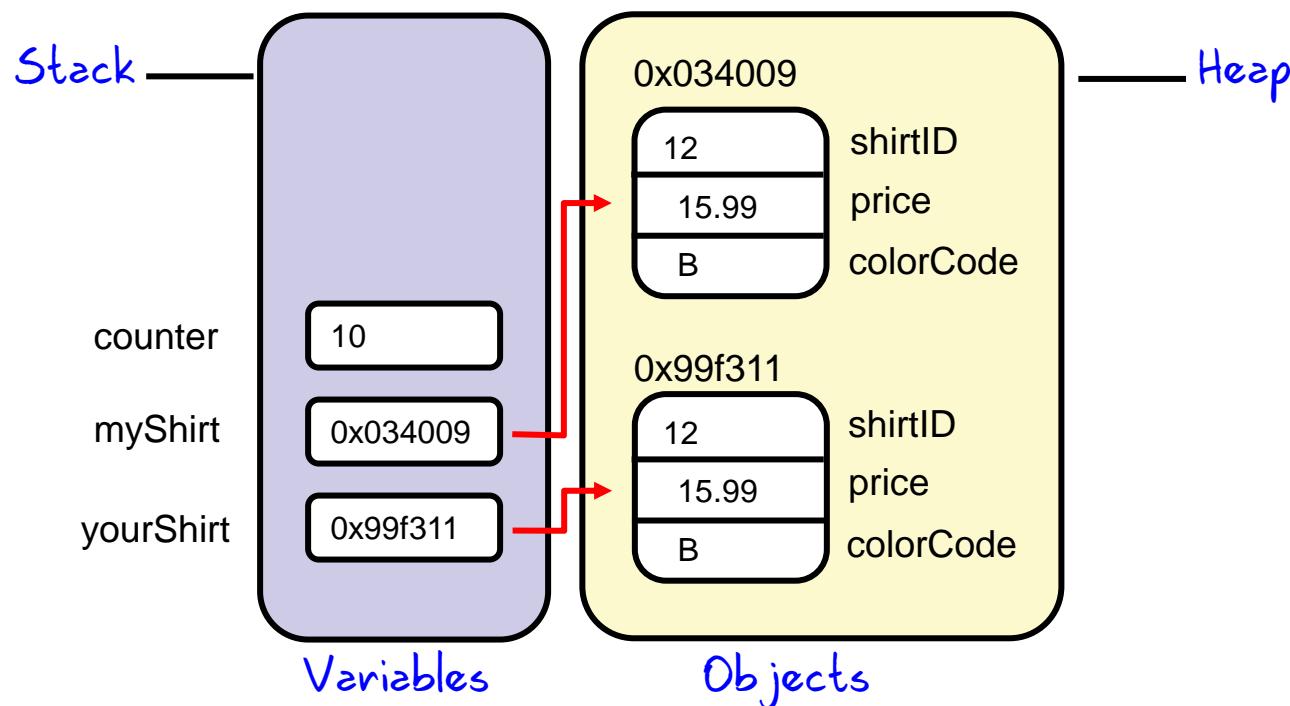
References to Different Objects



```
6 Camera remotel = new Camera();
7 remotel.menu();
8
9 TV remote2 = new TV();
10 remote2.menu();
11
12 Shirt myShirt = new Shirt();
13 myShirt.display();
14
15 Trousers myTrousers = new Trousers();
16 myTrousers.display();
```

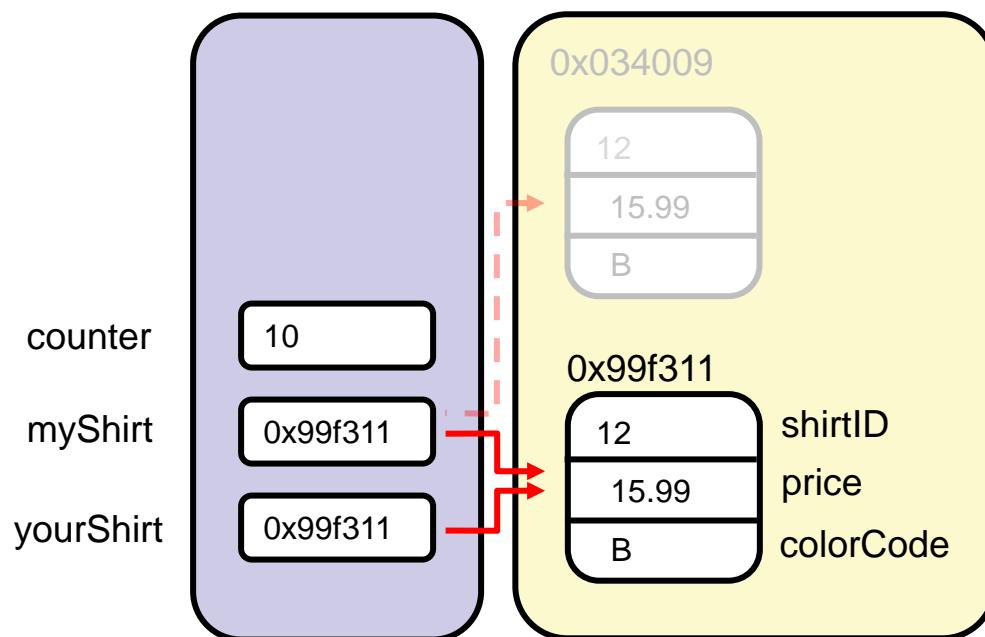
References and Objects in Memory

```
12 int counter = 10;  
13 Shirt myShirt = new Shirt();  
14 Shirt yourShirt = new Shirt();
```



Assigning a Reference to Another Reference

```
myShirt = yourShirt;
```



Two References, One Object

Code fragment:

```
12 Shirt myShirt = new Shirt();
13 Shirt yourShirt = new Shirt();
14
15 myShirt = yourShirt;           //The old myShirt object is
16                           //no longer referenced
17 myShirt.colorCode = 'R';
18 yourShirt.colorCode = 'G';
19
20 System.out.println("Shirt color: " + myShirt.colorCode);
```

Output from code fragment:

```
Shirt color: G
```

Exercise 6-2: Modify the ShoppingCart to Use Item Fields

- In this exercise, you:
 - Declare and instantiate two variables of type `Item` in the `ShoppingCart` class
 - Experiment with accessing properties and calling methods on the object



Topics

- Describing objects and classes
- Defining fields and methods
- Declaring, instantiating, and using objects
- Working with object references
- **Doing more with arrays**
- Introducing NetBeans IDE
- Introducing the soccer league use case

Arrays Are Objects

Arrays are handled by an implicit Array *object*.

- The Array variable is an *object reference*, not a primitive data type.
- It must be instantiated, just like other objects.
 - Example:

```
int[] ages = new int[4];
```

This array
can hold four
elements.

- Previously, you have been using a shortcut to instantiate your arrays.
 - Example:

```
int[] ages = {8, 7, 4, 5};
```

Declaring, Instantiating, and Initializing Arrays

- Examples:

```
1  String[] names = {"Mary", "Bob", "Carlos"};  
2  
3  int[] ages = new int[3];  
4  ages[0] = 19;  
5  ages[1] = 42;  
6  ages[2] = 92;
```

All in one
line

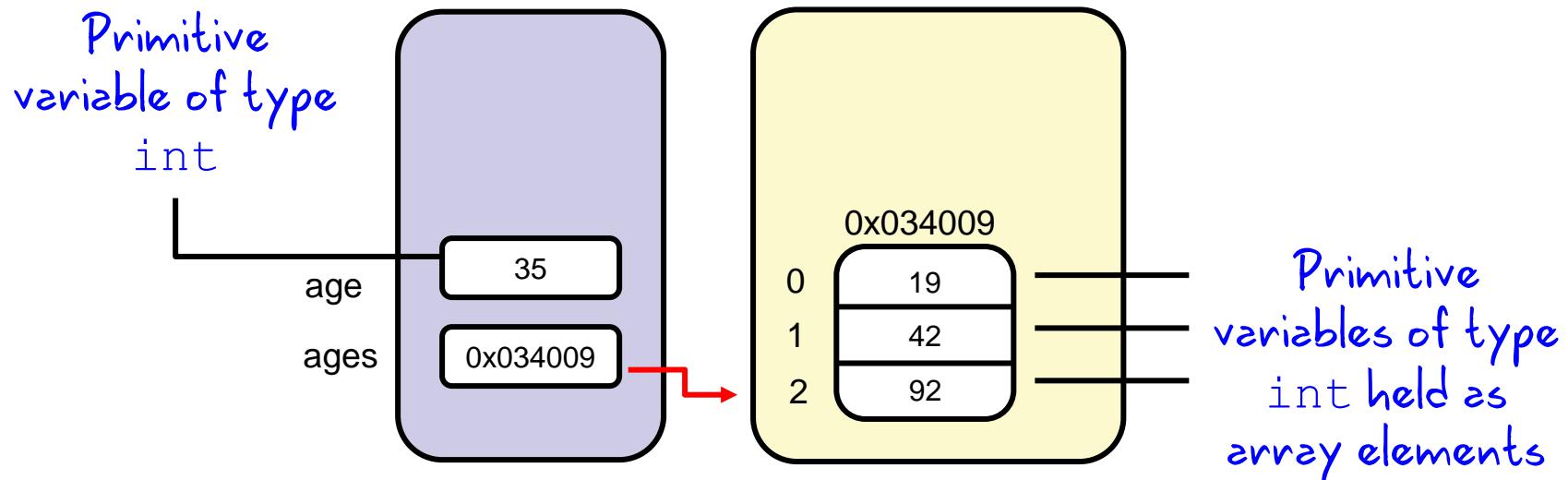
Multistep
approach

- Not permitted (compiler will show an error):

```
int [] ages;  
ages = {19, 42, 92};
```

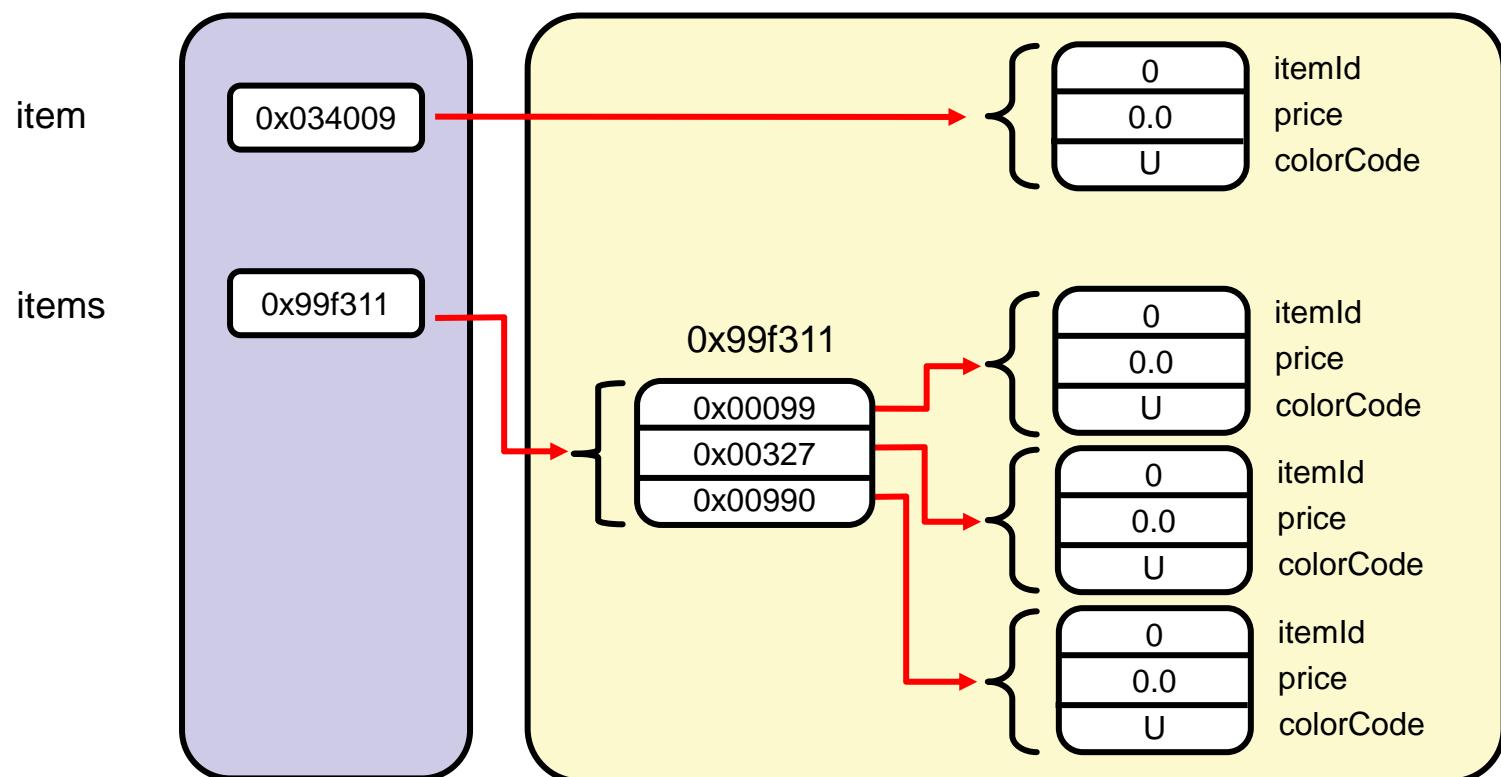
Storing Arrays in Memory

```
int age = 35;  
int[] ages = {19, 42, 92};
```



Storing Arrays of Object References in Memory

```
Item item = new Item();  
Item[] items = { new Item(), new Item(), new Item() };
```



Quiz

The following code is the correct syntax for _____ an array:

```
array_identifier = new type[length];
```

- a. Declaring
- b. Setting array values for
- c. Instantiating
- d. Declaring, instantiating, and setting array values for



Quiz

Given the following array declaration, which of the following statements are true?

```
int [ ] ages = new int [13];
```

- a. ages [0] is the reference to the first element in the array.
- b. ages [13] is the reference to the last element in the array.
- c. There are 13 integers in the ages array.
- d. ages [5] has a value of 0.



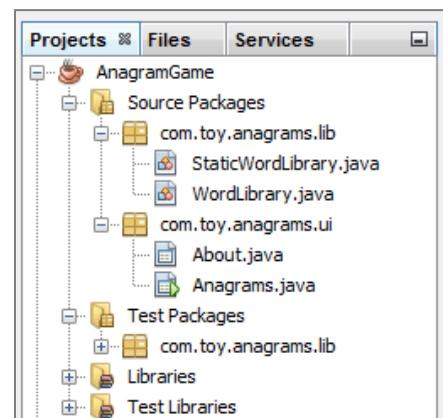
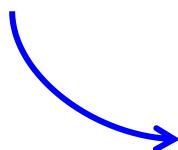
Topics

- Describing objects and classes
- Defining fields and methods
- Declaring, instantiating, and using objects
- Working with object references
- Doing more with arrays
- Introducing NetBeans IDE
- Introducing the soccer league use case

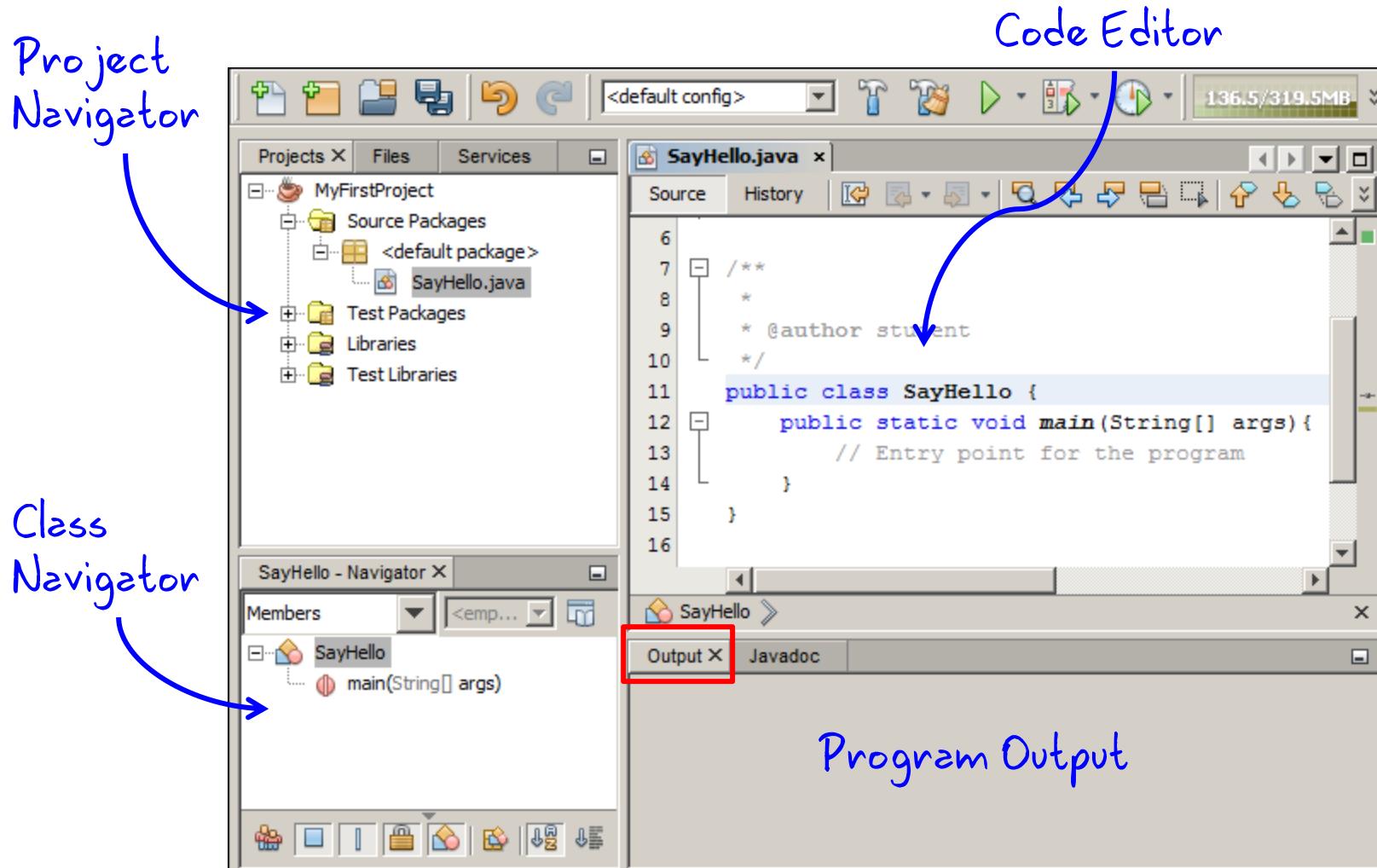
Java IDEs

A Java Integrated Development Environment (IDE) is a type of software that makes it easier to develop Java applications.

- An IDE provides:
 - Syntax checking
 - Various automation features
 - Runtime environment for testing
- It enables you to organize all your Java resources and environment settings into a *Project*.

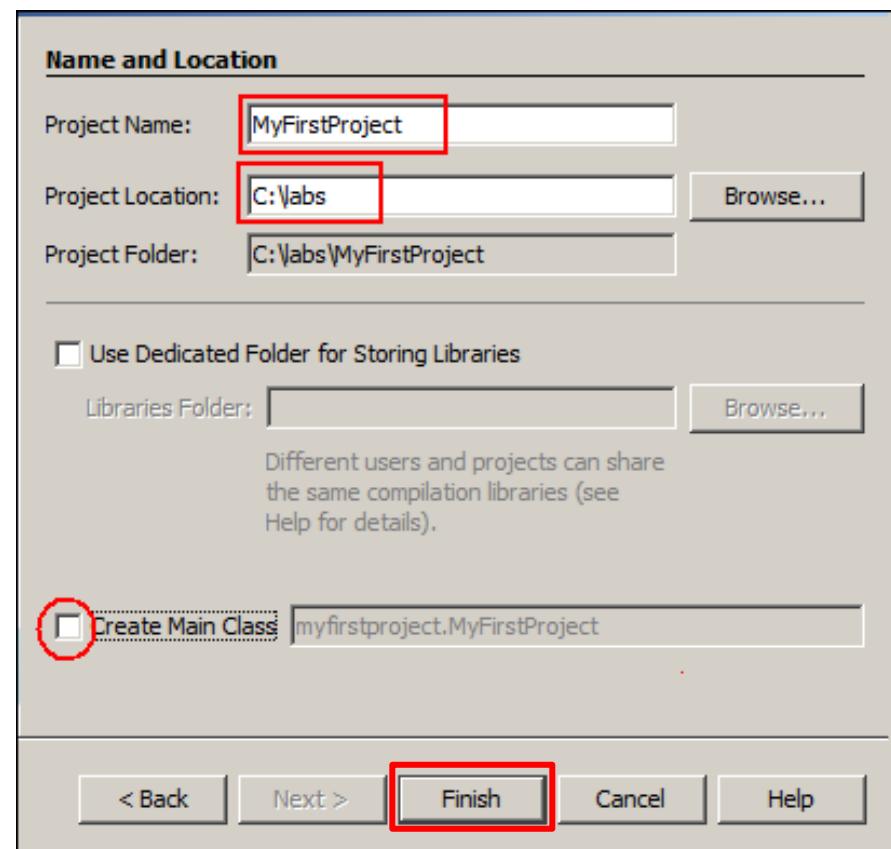


The NetBeans IDE



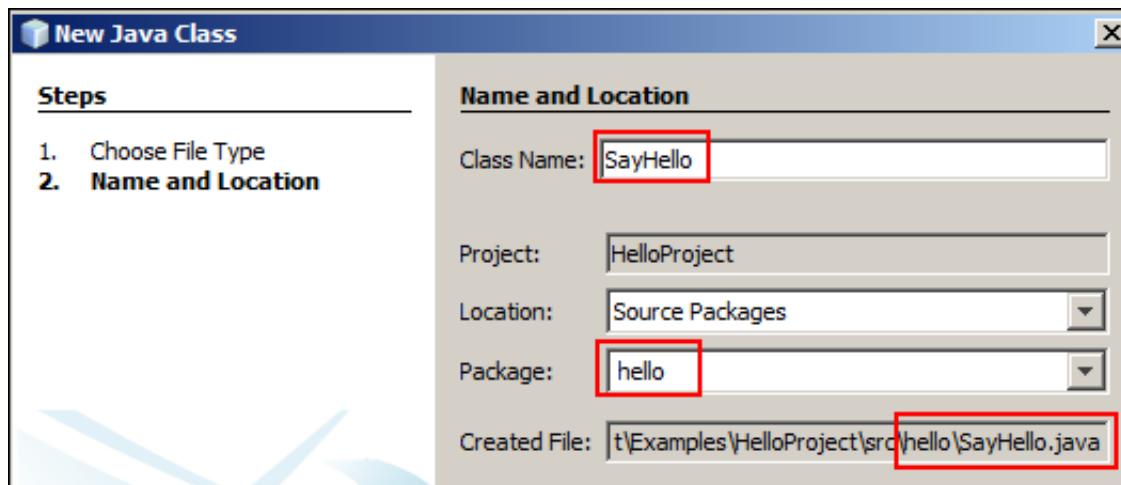
Creating a Java Project

1. Select **File > New Project**.
2. Select Java Application.
3. Name and set the location for the project.
4. Select “Create Main Class” if you want it done for you automatically.
5. Click **Finish**.



Creating a Java Class

1. Select **File > New File**.
2. Select your project and choose **Java Class**.
3. Name the class.
4. Assign a package.
5. Click **Finish**.



Avoiding Syntax Problems

The code editor will tell you when you have done something wrong.

The screenshot shows a Java code editor window titled "SayHello.java". The code is a simple "Hello World" program:6
7 /**/
8 *
9 * @author student
10 */
11 ;' expected sayHello {
12 public void main(String[] args) {
13 System.out.println("Hello World!");
14 }
15 }
16 }The editor highlights several errors:

- A red circle highlights the semicolon at line 11, which is followed by the message "' expected'".
- A red circle highlights the word "out" in the line "System.out.println("Hello World!");".
- A red circle highlights the closing brace at line 15.

A tooltip "Alt-Enter shows hints" is visible near the error at line 11. The status bar at the bottom shows "SayHello > main >".

Compile Error: Variable Not Initialized

```
1 public static void main(String[] args) {  
2  
3     Customer customer01;                      //Declare the reference  
4                                         //No instantiation  
5     customer01.name = "Robert";  
9  
10 }
```

NetBeans indicates that
the variable may not
have been initialized.

```
public static void main(String[] args) {  
  
    Customer customer01;                      //Declare the reference  
                                                //but no instantiation  
  
    customer01.name = "Robert";  
  
    variable customer01 might not have been initialized  
    ----  
    (Alt-Enter shows hints)
```

Runtime Error: NullPointerException

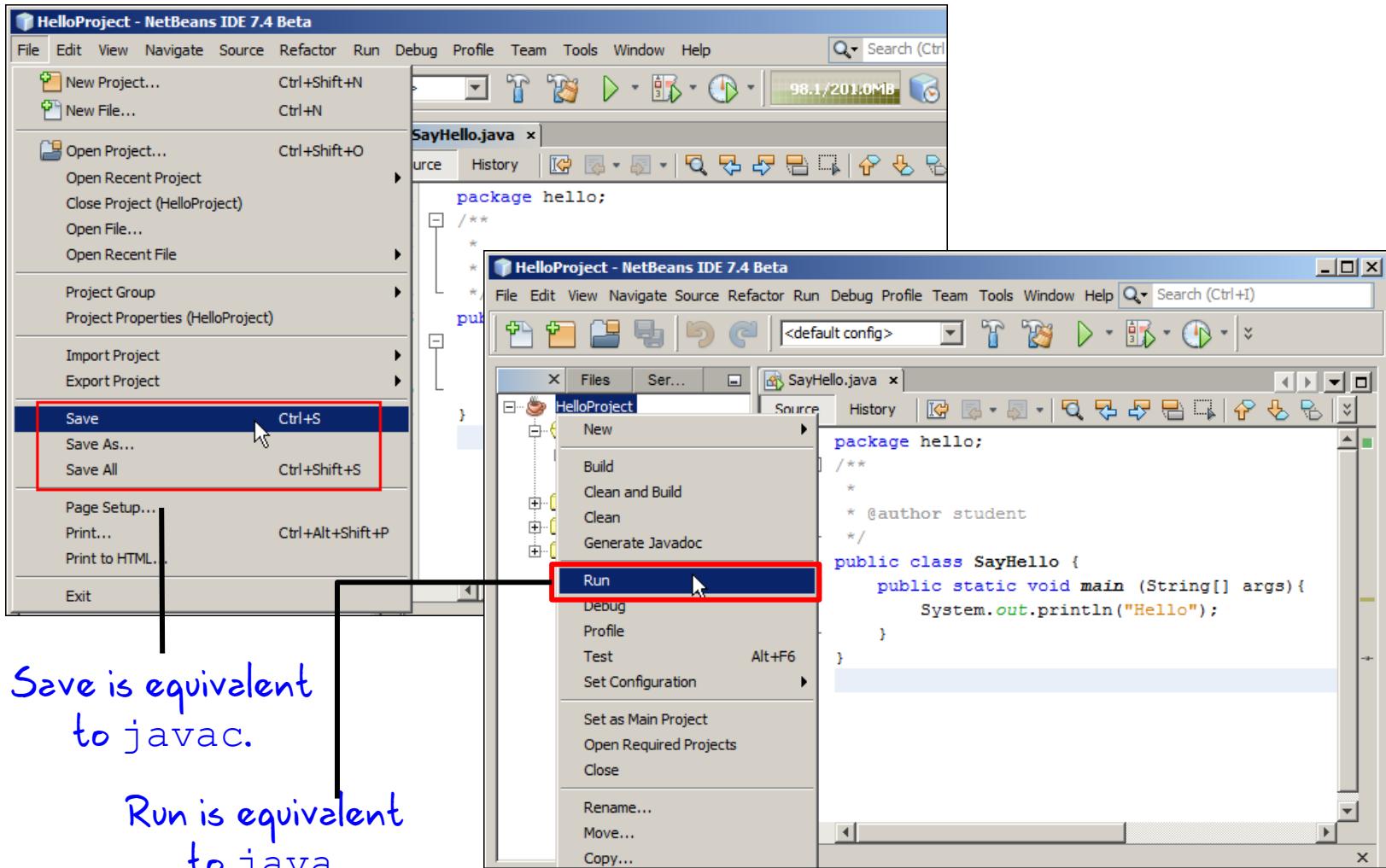
```
1 public static void main(String[] args) {  
2  
3     Customer customer01;                      //Declare the reference  
4     customer01 = new Customer();                //Instantiate and assign  
5     customer01.name = "Robert";  
6  
7     Customer[] customers = new Customer[5];  
8     customers[0].name = "Robert";  
9  
10 }
```

This reference has
not been assigned.

NetBeans output window
indicates a
NullPointerException.

```
run:  
Exception in thread "main" java.lang.NullPointerException  
at Test.main(Test.java:49)  
Java Result: 1  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Compiling and Running a Program by Using NetBeans



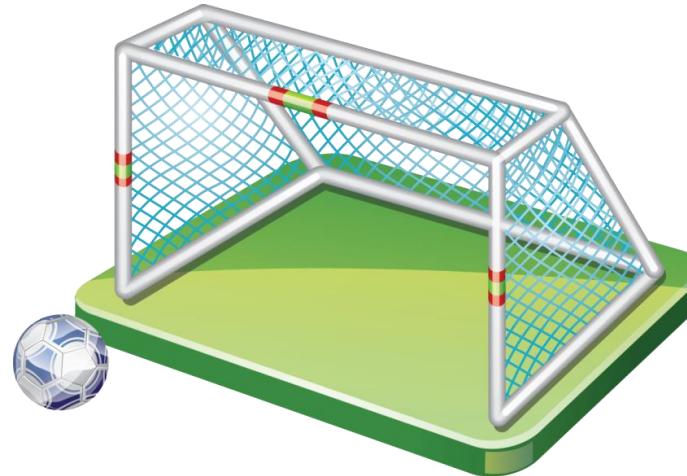
Topics

- Describing objects and classes
- Defining fields and methods
- Declaring, instantiating, and using objects
- Working with object references
- Doing more with arrays
- Introducing NetBeans IDE
- Introducing the soccer league use case

Soccer Application

Practices 6 through 14 build a soccer league application with the following features:

- Any number of soccer teams, each with up to 11 players
- Set up an all-play-all league.
- Use a random play game generator to create test games.
- Determine the rank order of teams at the end of the season.



Creating the Soccer Application

A separate project for each practice

The screenshot shows an IDE interface. On the left is the 'Projects X' view, which lists a project named '11-MoreArraysLoops_Practice1'. Inside this project, there is a 'Source Packages' folder containing a 'soccer' package. The 'soccer' package contains five files: Game.java, Goal.java, League.java, Player.java, and PlayerDatabase.java. A blue bracket on the left points to the 'soccer' package. On the right is the 'Start Page' view, showing the source code for 'League.java'. The code defines a public class 'League' with a main method that creates an instance of 'League'. Lines 20 and 21 contain JavaDoc comments.

```
public class League {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        League theLeague = new League();  
    }  
}
```

Sample output showing events in a game

The screenshot shows the terminal output of the application. It displays a sequence of events from a soccer game between 'The Greys' and 'The Pinks'. The output includes kickoffs, possession changes, goals, and the final score. Below this, it shows a summary of team points.

```
The Greys vs. The Pinks (2014-03-08)  
Kickoff by Agatha Christie of The Greys. (0.0 mins.)  
Arthur Conan Doyle of The Pinks currently has possession. (6.0 mins.)  
GOAL! Scored by W. B. Yeats of The Greys. (7.0 mins.)  
Kickoff by Alan Patton of The Pinks. (8.0 mins.)  
Alexander Solzhenitsyn of The Pinks currently has possession. (11.0 mins.)  
GOAL! Scored by Arthur Conan Doyle of The Pinks. (14.0 mins.)  
Kickoff by Agatha Christie of The Greys. (18.0 mins.)  
Alan Patton of The Pinks currently has possession. (23.0 mins.)  
Agatha Christie of The Greys currently has possession. (24.0 mins.)  
GOAL! Scored by Agatha Christie of The Greys. (40.0 mins.)  
Kickoff by Arthur Conan Doyle of The Pinks. (44.0 mins.)  
Arthur Conan Doyle of The Pinks currently has possession. (49.0 mins.)  
GOAL! Scored by Arthur Conan Doyle of The Pinks. (55.0 mins.)  
Kickoff by Agatha Christie of The Greys. (59.0 mins.)  
Alan Patton of The Pinks currently has possession. (73.0 mins.)  
GOAL! Scored by W. B. Yeats of The Greys. (89.0 mins.)  
The Pinks win! (3 - 2)  
  
Team Points  
The Reds:17:20  
The Blues:17:17  
The Pinks:12:17  
The Greens:8:12  
The Greys:6:13  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Sample output showing rank order of teams

Soccer Web Application

Soccer League Games

[Replay games](#)

		Away Teams								
Home Teams	The Magpies	The Magpies	(0 - 1)	(4 - 2)	(1 - 0)	(3 - 0)	(1 - 0)	15	18	
		The Crows	(2 - 1)	X	(1 - 0)	(0 - 1)	(0 - 0)	(0 - 0)	10	18
		The Reds	(0 - 1)	(0 - 1)	X	(1 - 1)	(1 - 0)	(1 - 0)	13	14
		The Blues	(4 - 1)	(0 - 2)	(0 - 1)	X	(3 - 4)	(1 - 0)	12	14
		The Rovers	(3 - 0)	(5 - 2)	(2 - 4)				18	15
		The Harriers	(1 - 3)	(1 - 1)	(3 - 3)				8	7
				The Rovers vs. The Reds (2 - 4)						
Team	Player	Time								
The Reds	Jane Austin	7								
The Rovers	J. M. Synge	21								
The Reds	Jane Austin	41								
The Reds	Mark Twain	46								
The Reds	Brian Moore	76								
The Rovers	Charlotte Bronte	83								

Teams listed in rank order

Click the score of a game to show game details.

Points and goals scored used for ordering

[Return to main page](#)

Summary

In this lesson, you should have learned how to:

- Describe the characteristics of a class
- Define an object as an instance of a class
- Instantiate an object and access its fields and methods
- Describe how objects are stored in memory
- Instantiate an array of objects
- Describe how an array of objects is stored in memory
- Declare an object as a field
- Use the NetBeans IDE



Challenge Questions: Java Puzzle Ball

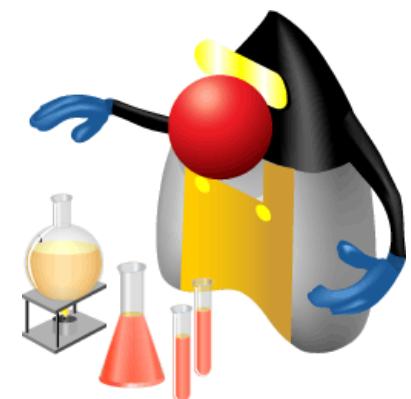
- How many objects can you identify in the game?
- Given that a class is a blueprint for an object, which game components best reflect the class-instance relationship?
- How many object properties can you find?
- Can you guess what some of the methods might be?



Practice 6-1 Overview: Creating Classes for the Soccer League

This practice covers creating the five classes required for the soccer application:

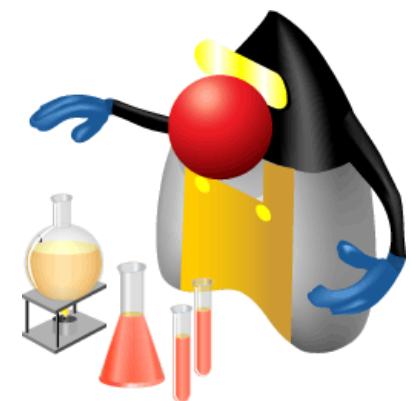
- Goal
- Game
- Player
- Team
- League



Practice 6-2 Overview: Creating a Soccer Game

This practice covers the following topics:

- Creating a new game
- Adding some goals

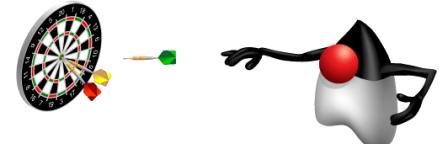


Manipulating and Formatting the Data in Your Program

Objectives

After completing this lesson, you should be able to:

- Describe the `String` class and use some of the methods of the `String` class
- Use the JDK documentation to search for and learn how to use a class
- Describe the `StringBuilder` class
- Explain what a constant is and how to use it
- Explain the difference between promoting and casting of variables



Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

String Class

```
String hisName = "Fred Smith"; — Standard syntax
```

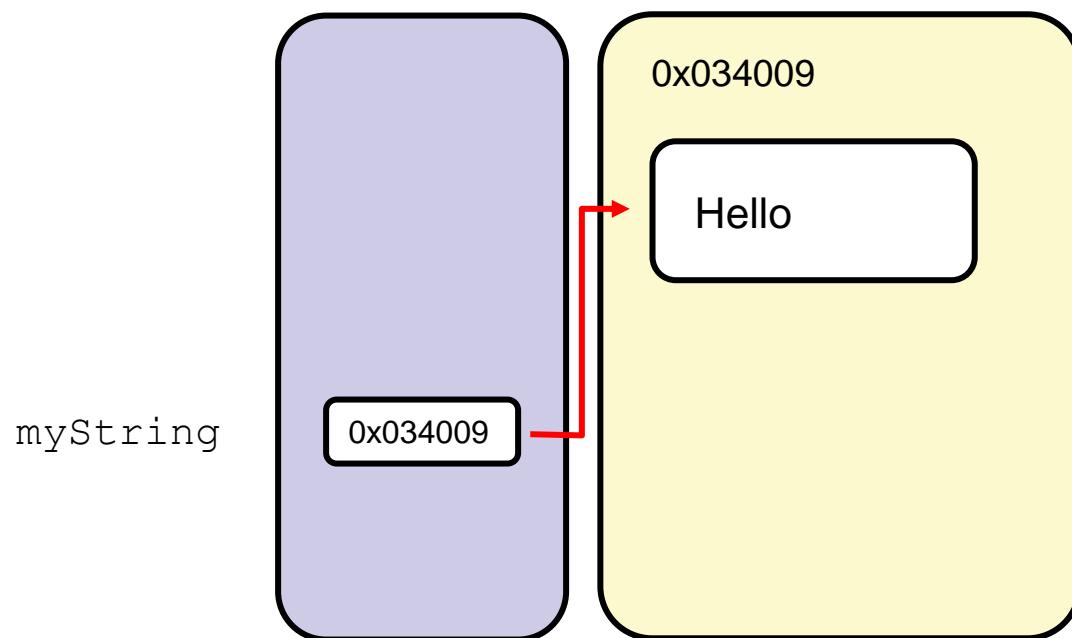
The new keyword can be used,
but it is not best practice:

```
String herName = new String("Anne Smith");
```

- A String object is immutable; its value cannot be changed.
- A String object can be used with the string concatenation operator symbol (+) for concatenation.

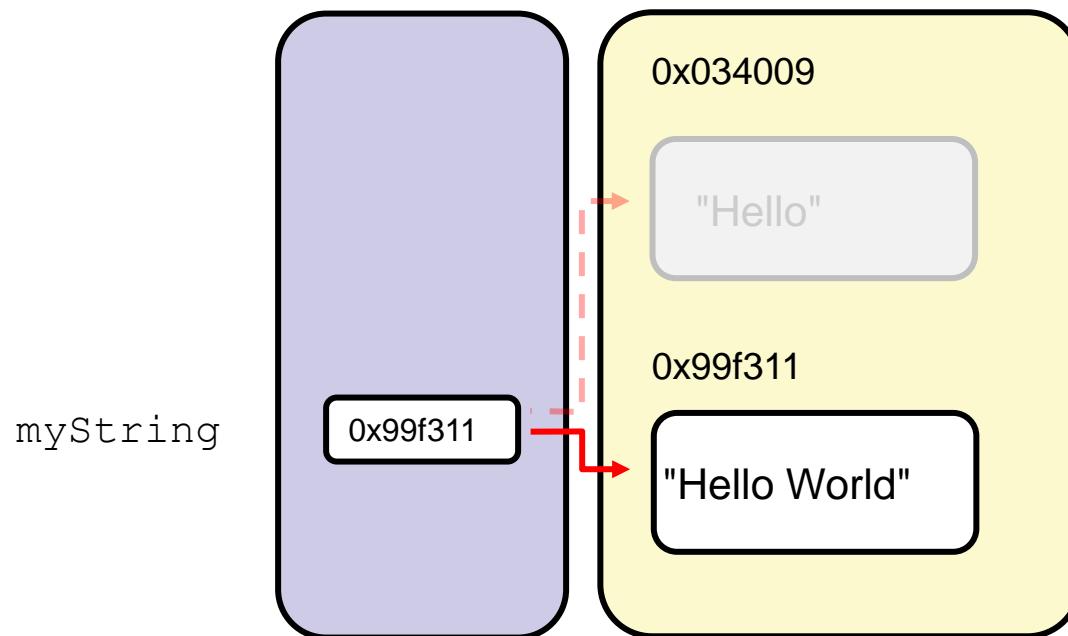
Concatenating Strings

```
String myString = "Hello";
```



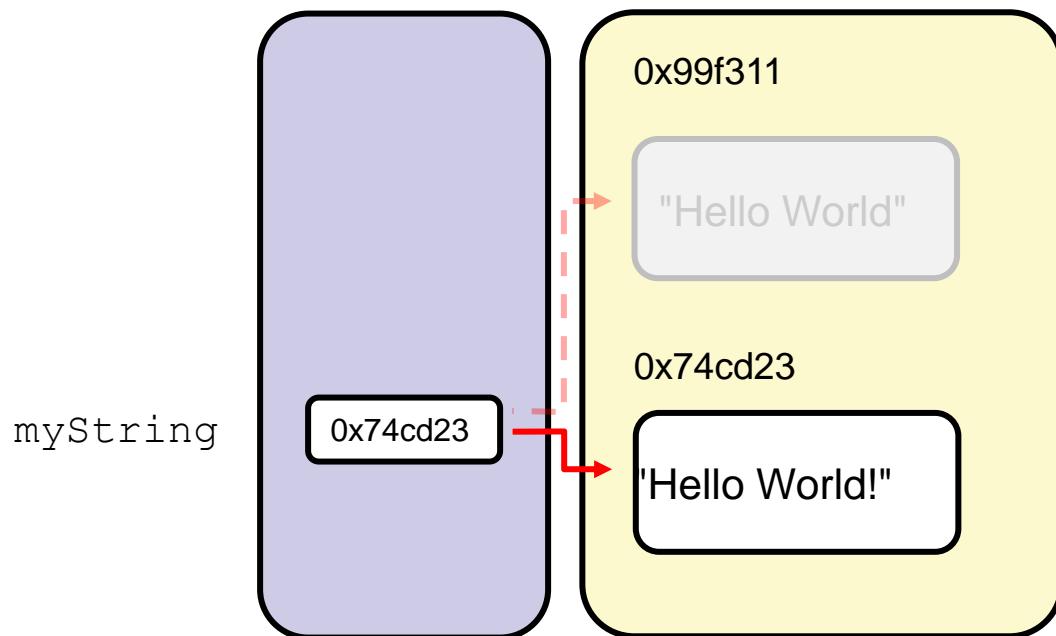
Concatenating Strings

```
String myString = "Hello";  
myString = myString.concat(" World");
```



Concatenating Strings

```
String myString = "Hello";
myString = myString.concat(" World");
myString = myString + "!"
```



String Method Calls with Primitive Return Values

A method call can return a single value of any type.

- An example of a method of primitive type int:

```
String hello = "Hello World";  
int stringLength = hello.length();
```

String Method Calls with Object Return Values

Method calls returning objects:

```
String greet = " HOW ".trim();
```

```
String lc = greet + "DY".toLowerCase();
```

Or

```
String lc = (greet + "DY").toLowerCase();
```

Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Java API Documentation

Consists of a set of webpages;

- Lists all the classes in the API
 - Descriptions of what the class does
 - List of constructors, methods, and fields for the class
- Highly hyperlinked to show the interconnections between classes and to facilitate lookup
- Available on the Oracle website at:
<http://download.oracle.com/javase/8/docs/api/index.html>

Java Platform SE 8 Documentation

Select All Classes or a particular package.

The classes for the selected package(s) are listed here.

Details about the class selected

The screenshot shows the Java Platform SE 8 Documentation interface. On the left, there is a sidebar with navigation links: 'All Classes', 'All Profiles', 'Packages' (which is expanded to show 'java.awt' and 'java.awt.color'), and 'Enums' (which is expanded to show 'Character.UnicodeScript', 'ProcessBuilder.Redirect.Type', and 'Thread.State'). The main content area displays details for the 'String' class. At the top, it says 'Please note that the specifications and other information contained herein are not final and are subject to change. The information is being made available to you solely for purpose of evaluation.' Below this is a navigation bar with tabs: 'Overview', 'Package', 'Class' (which is highlighted in orange), 'Use', 'Tree', 'Deprecated', 'Index', and 'Help'. Under the 'Class' tab, there are links for 'Prev Class', 'Next Class', 'Frames', and 'No Frames'. Below these are links for 'summary: Nested | Field | Constr | Method' and 'Detail: Field | Constr | Method'. The main content area starts with the class name 'String' and its inheritance: 'public final class String extends Object implements Serializable, Comparable<String>, CharSequence'. It also lists implemented interfaces: 'Serializable, CharSequence, Comparable<String>'. A note states: 'The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.' The right side of the interface has a header: 'Java™ Platform Standard Ed. 8 DRAFT ea-b113'.

Java Platform SE 8: Method Summary

```
public int charAt(String str)
```

The return type
of the method

The name of
the method

The type of the parameter that must
be passed into the method

Method Summary	
Methods	
Modifier and Type	Method and Description
char	<code>charAt(int index)</code> Returns the char value at the specified index.
int	<code>codePointAt(int index)</code> Returns the character (Unicode code point) at the specified index.
int	<code>codePointBefore(int index)</code> Returns the character (Unicode code point) before the specified index.
int	<code>codePointCount(int beginIndex, int endIndex)</code> Returns the number of Unicode code points in the specified text range of this String.
int	<code>compareTo(String anotherString)</code> Compares two strings lexicographically.
int	<code>compareToIgnoreCase(String str)</code> Compares two strings lexicographically, ignoring case differences.
String	<code>concat(String str)</code> Concatenates the specified string to the end of this string.

Java Platform SE 8: Method Detail

Click here to get the detailed description of the method.

int **indexOf(String str)**

Returns the index within this string of the first occurrence of the specified substring.

int **indexOf(String str, int fromIndex)**

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

Detailed description for the indexOf() method

indexOf

public int indexOf(String str)

Returns the index within this string of the first occurrence of the specified substring.

The returned index is the smallest value *k* for which:

this.startsWith(str, k)

If no such value of *k* exists, then -1 is returned.

Parameters:

str - the substring to search for.

Returns:

the index of the first occurrence of the specified substring, or -1 if there is no such occurrence.

Further details about parameters and return value are shown in the method list.

indexOf Method Example

```
1 String phoneNum = "404-543-2345";  
2 int idx1 = phoneNum.indexOf('-');  
3 System.out.println("index of first dash: "+ idx1);  
4  
5  
6 int idx2 = phoneNum.indexOf('-', idx1+1);  
7 System.out.println("second dash idx: "+idx2);
```

The 1-arg version

The 2-arg version

Topics

- Using the `String` class
- Using the Java API docs
- **Using the `StringBuilder` class**
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

StringBuilder Class

`StringBuilder` provides a mutable alternative to `String`.
`StringBuilder`:

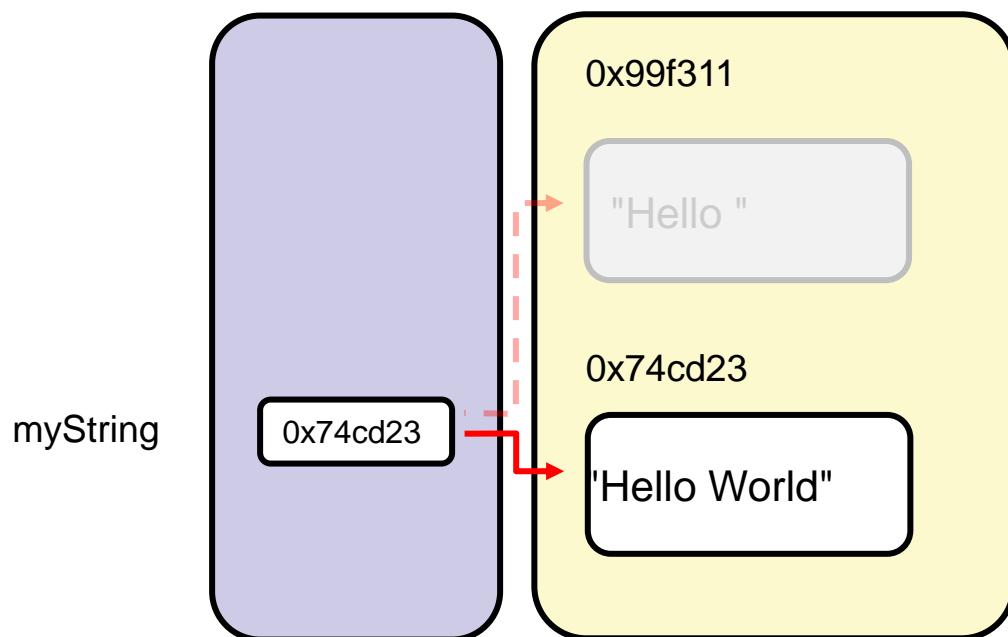
- Is instantiated using the `new` keyword
- Has many methods for manipulating its value
- Provides better performance because it is mutable
- Can be created with an initial capacity

`String` is still needed because:

- It may be safer to use an immutable object
- A method in the API may require a string
- It has many more methods not available on
`StringBuilder`

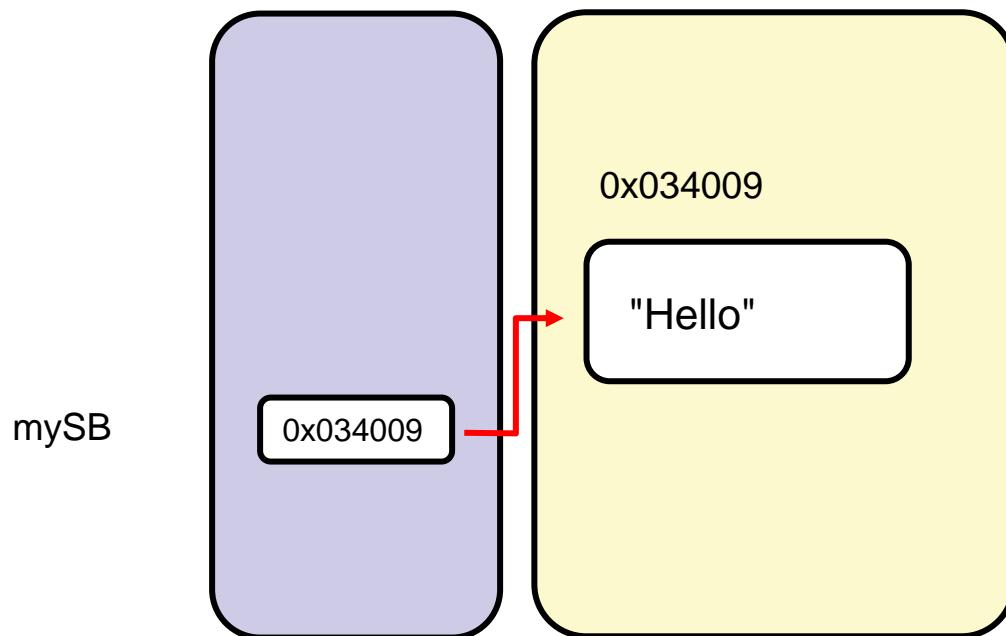
StringBuilder Advantages over String for Concatenation (or Appending)

```
String myString = "Hello";
myString = myString + " World";
```



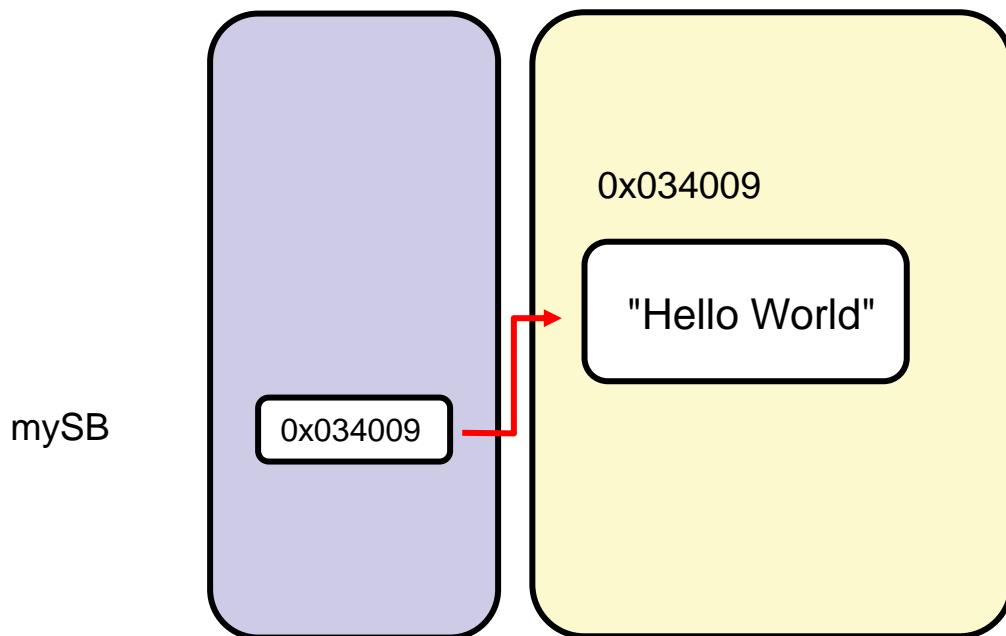
StringBuilder: Declare and Instantiate

```
StringBuilder mySB = new StringBuilder("Hello");
```



StringBuilder Append

```
StringBuilder mySB = new StringBuilder("Hello");  
mySB.append(" World");
```



Quiz

Which of the following statements are true? (Choose all that apply.)

- a. The dot (.) operator creates a new object instance.
- b. The String class provides you with the ability to store a sequence of characters.
- c. The Java API specification contains documentation for all of the classes in a Java technology product.
- d. String objects cannot be modified.

Exercise 7-1: Use indexOf and substring Methods

In this exercise, you use `indexOf` and `substring` methods to get just the customer's first name and display it.



Exercise 7-2: Instantiate the `StringBuilder` object

In this exercise, you instantiate a `StringBuilder` object, initializing it to `firstName` using the `StringBuilder` constructor.



Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Primitive Data Types

- Integral types (byte, short, int, and long)
- Floating point types (float and double)
- Textual type (char)
- Logical type (boolean)

Some New Integral Primitive Types

Type	Length	Range
byte	8 bits	-2^7 to $2^7 - 1$ (−128 to 127, or 256 possible values)
short	16 bits	-2^{15} to $2^{15} - 1$ (−32,768 to 32,767, or 65,535 possible values)
int	32 bits	-2^{31} to $2^{31} - 1$ (−2,147,483,648 to 2,147,483,647, or 4,294,967,296 possible values)
long	64 bits	-2^{63} to $2^{63} - 1$ (−9,223,372,036854,775,808 to 9,223,372,036854,775,807, or 18,446,744,073,709,551,616 possible values)

Floating Point Primitive Types

Type	Float Length
<code>float</code>	32 bits
<code>double</code> (default type for floating point literals)	64 bits

Example:

```
public float pi = 3.141592F;
```

Textual Primitive Type

- The only primitive textual data type is `char`.
- It is used for a single character (16 bits).
- Example:

```
— public char colorCode = 'U';
```

Single quotes must be used with `char` literal values.

Java Language Trivia: Unicode

- Unicode is a standard character encoding system.
 - It uses a 16-bit character set.
 - It can store all the necessary characters from most languages.
 - Programs can be written so they display the correct language for most countries.

Character	UTF-16	UTF-8	UCS-2
A	0041	41	0041
c	0063	63	0063
Ö	00F6	C3 B6	00F6
亞	4E9C	E4 BA 9C	4E9C
♪	D834 DD1E	F0 9D 84 9E	N/A

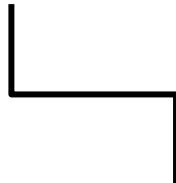
Constants

- Variable (can change):

- double salesTax = 6.25;

- Constant (cannot change):

- **final** int NUMBER_OF_MONTHS = 12;



The **final** keyword causes
a variable to be read only.

Quiz

The variable declaration `public int myInteger=10;` adheres to the variable declaration and initialization syntax.

- a. True
- b. False

Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Modulus Operator

Purpose	Operator	Example	Comments
Remainder	$\%$ <i>modulus</i>	num1 = 31; num2 = 6; mod = num1 % num2; mod is 1	Remainder finds the remainder of the first number divided by the second number. $\begin{array}{r} 5 \text{ R } 1 \\ \hline 6 \overline{)31} \\ 30 \\ \hline 1 \end{array}$ Remainder always gives an answer with the same sign as the first operand.

Combining Operators to Make Assignments

Purpose	Operator	Examples <code>int a = 6, b = 2;</code>	Result
Add to and assign	<code>+=</code>	<code>a += b</code>	<code>a = 8</code>
Subtract from and assign	<code>-=</code>	<code>a -= b</code>	<code>a = 4</code>
Multiply by and assign	<code>*=</code>	<code>a *= b</code>	<code>a = 12</code>
Divide by and assign	<code>/=</code>	<code>a /= b</code>	<code>a = 3</code>
Get remainder and assign	<code>%=</code>	<code>a %= b</code>	<code>a = 0</code>

More on Increment and Decrement Operators

Operator	Purpose	Example
++	Preincrement <i>(++variable)</i>	<code>int id = 6; int newId = ++id; id is 7, newId is 7</code>
	Postincrement <i>(variable++)</i>	<code>int id = 6; int newId = id++; id is 7, newId is 6</code>
--	Predecrement <i>--variable</i>	<i>(same principle applies)</i>
	Postdecrement <i>(variable--)</i>	

Increment and Decrement Operators (++ and --)

Examples:

```
1 int count=15;
2 int a, b, c, d;
3 a = count++;
4 b = count;
5 c = ++count;
6 d = count;
7 System.out.println(a + ", " + b + ", " + c + ", " + d);
```

Output:

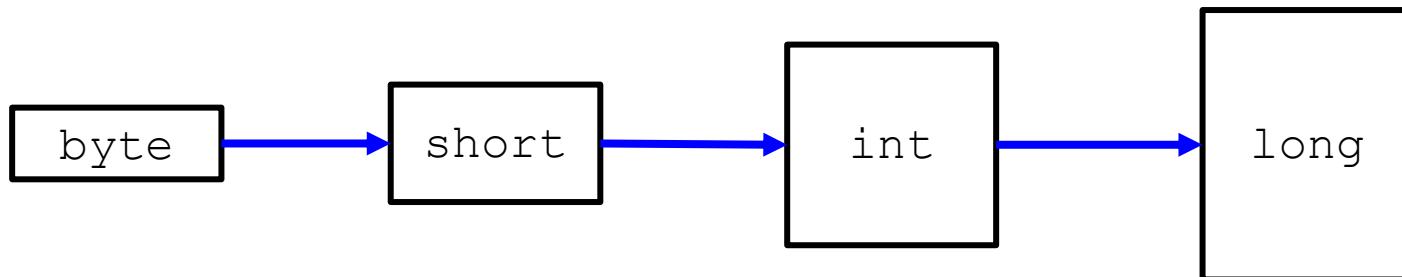
15, 16, 17, 17

Topics

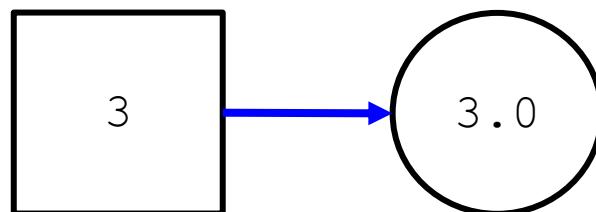
- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Promotion

- Automatic promotions:
 - If you assign a smaller type to a larger type



- If you assign an integral type to a floating point type



- Examples of automatic promotions:
 - `long intToLong = 6;`
 - `double int.ToDouble = 3;`

Caution with Promotion

Equation:

$$55555 * 66666 = 3703629630$$

Example of potential issue:

```
1 int num1 = 55555;  
2 int num2 = 66666;  
3 long num3;  
4 num3 = num1 * num2;           //num3 is -591337666
```

Example of potential solution:

```
1 int num1 = 55555;  
2 long num2 = 66666; ————— Changed from int to long  
3 long num3;  
4 num3 = num1 * num2;           //num3 is 3703629630
```

Caution with Promotion

Equation:

$$7 / 2 = 3.5$$

Example of potential issue:

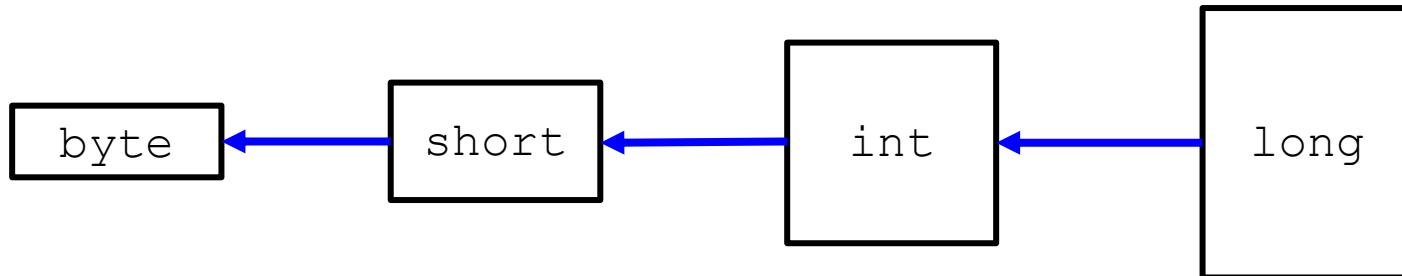
```
1 int num1 = 7;  
2 int num2 = 2;  
3 double num3;  
4 num3 = num1 / num2;           //num3 is 3.0
```

Example of potential solution:

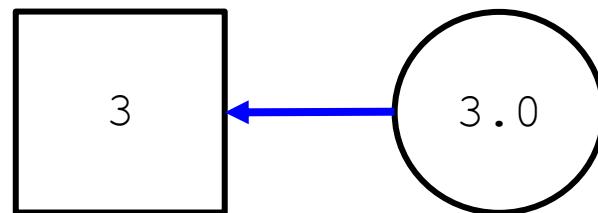
```
1 int num1 = 7;  
2 double num2 = 2; ————— Changed from int to double  
3 double num3;  
4 num3 = num1 / num2;           //num3 is 3.5
```

Type Casting

- When to cast:
 - If you assign a larger type to a smaller type



- If you assign a floating point type to an integral type



- Examples of casting:
 - `int longToInt = (int)20L;`
 - `short doubleToShort = (short)3.0;`

Caution with Type Casting

Example of potential issue:

```
1 int myInt;  
2 long myLong = 123987654321L;  
3 myInt = (int) (myLong); // Number is "chopped"  
4 // myInt is -566397263
```

Safer example of casting:

```
1 int myInt;  
2 long myLong = 99L;  
3 myInt = (int) (myLong); // No data loss, only zeroes.  
4 // myInt is 99
```

Caution with Type Casting

- Be aware of the possibility of lost precision.

Example of potential issue:

```
1 int myInt;  
2 double myPercent = 51.9;  
3 myInt = (int) (myPercent); // Number is "chopped"  
4 // myInt is 51
```

Using Promotion and Casting

Example of potential issue:

```
1 int num1 = 53; // 32 bits of memory to hold the value
2 int num2 = 47; // 32 bits of memory to hold the value
3 byte num3;      // 8 bits of memory reserved
4 num3 = (num1 + num2); // causes compiler error
```

Solution using a larger type for num3:

```
1 int num1 = 53;
2 int num2 = 47;
3 int num3; ————— Changed from byte to int
4 num3 = (num1 + num2);
```

Solution using casting:

```
1 int num1 = 53; // 32 bits of memory to hold the value
2 int num2 = 47; // 32 bits of memory to hold the value
3 byte num3;      // 8 bits of memory reserved
4 num3 = (byte) (num1 + num2); // no data loss
```

Compiler Assumptions for Integral and Floating Point Data Types

- Most operations result in an `int` or `long`:
 - `byte`, `char`, and `short` values are automatically promoted to `int` prior to an operation.
 - If an expression contains a `long`, the entire expression is promoted to `long`.
- If an expression contains a floating point, the entire expression is promoted to a floating point.
- All literal floating point values are viewed as `double`.

Automatic Promotion

Example of potential problem:

```
short a, b, c;  
a = 1 ; } b = 2 ; } a and b are automatically promoted to integers.  
c = a + b ; //compiler error
```

Example of potential solutions:

- Declare `c` as an `int` type in the original declaration:

```
int c;
```

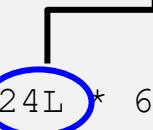
- Type cast the `(a+b)` result in the assignment line:

```
c = (short) (a+b);
```

Using a long

```
1 public class Person {  
2  
3     public int ageYears = 32;  
4  
4     public void calculateAge() {  
5  
6         int ageDays = ageYears * 365;  
7         long ageSeconds = ageYears * 365 * 24L * 60 * 60;  
8  
9         System.out.println("You are " + ageDays + " days old.");  
10        System.out.println("You are " + ageSeconds + " seconds old.");  
11  
12    } // end of calculateAge method  
13 } // end of class
```

Using the L to indicate a long will result in the compiler recognizing the total result as a long.



Using Floating Points

Example of potential problem:

Expressions are automatically promoted to floating points.

```
int num1 = 1 + 2 + 3 + 4.0;           //compiler error
int num2 = (1 + 2 + 3 + 4) * 1.0; }   //compiler error
```

Example of potential solutions:

- Declare num1 and num2 as double types:

```
double num1 = 1 + 2 + 3 + 4.0;          //10.0
double num2 = (1 + 2 + 3 + 4) * 1.0; }  //10.0
```

- Type cast num1 and num2 as int types in the assignment line:

```
int num1 = (int)(1 + 2 + 3 + 4.0);    //10
int num2 = (int)((1 + 2 + 3 + 4) * 1.0); //10
```

Floating Point Data Types and Assignment

- Example of potential problem:

```
float float1 = 27.9; //compiler error
```

- Example of potential solutions:

- The F notifies the compiler that 27.9 is a float value:

```
float float1 = 27.9F;
```

- 27.9 is cast to a float type:

```
float float1 = (float) 27.9;
```

Quiz

Which statements are true?

- a. There are eight primitive types built in to the Java programming language.
- b. byte, short, char, and long are the four integral primitive data types in the Java programming language.
- c. A boolean type variable holds true, false, and nil.
- d. short Long = 10; is a valid statement that adheres to the variable declaration and initialization syntax.

Exercise 7-3: Declare a Long, Float, and Char

In this exercise, you experiment with the data types introduced in this lesson. You:

- Declare and initialize variables
- Cast one numeric type to another



Summary

In this lesson, you should have learned how to:

- Describe the `String` class and use some of the methods of the `String` class
- Use the JDK documentation to search for and learn how to use a class
- Use the `StringBuilder` class to manipulate string data
- Create a constant by using the `final` keyword in the variable declaration
- Describe how the Java compiler can use promotion or casting to interpret expressions and avoid a compiler error



Play Time!

Play **Basic Puzzle 8** before the lesson titled “Creating and Using Methods.”

Consider the following:

What happens when you rotate the blue wheel?

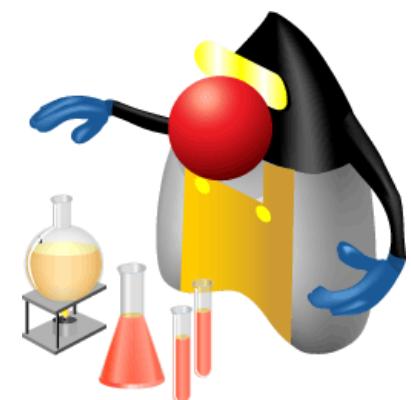
How else can you affect the rotation of bumpers?



Practice 7-1 Overview: Manipulating Text

This practice covers the following topics:

- Searching for a particular player and printing out the last name
- Reversing the player name so that the family name is printed first



Creating and Using Methods



Objectives

After completing this lesson, you should be able to:

- Instantiate a class and call a method on the object
- Describe the purpose of a constructor method
- Create a method that takes arguments and returns a value
- Access a `static` method from a different class
- Use a `static` method of the `Integer` class to convert a string into an `int`
- Overload a method

Topics

- Using methods and constructors
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Basic Form of a Method

The void keyword indicates that the method does not return a value.

Empty parentheses indicate that no arguments are passed to the method.

```
1 public void display () {  
2     System.out.println("Shirt description: " + description);  
3     System.out.println("Color Code: " + colorCode);  
4     System.out.println("Shirt price: " + price);  
5 } // end of display method
```

Calling a Method from a Different Class

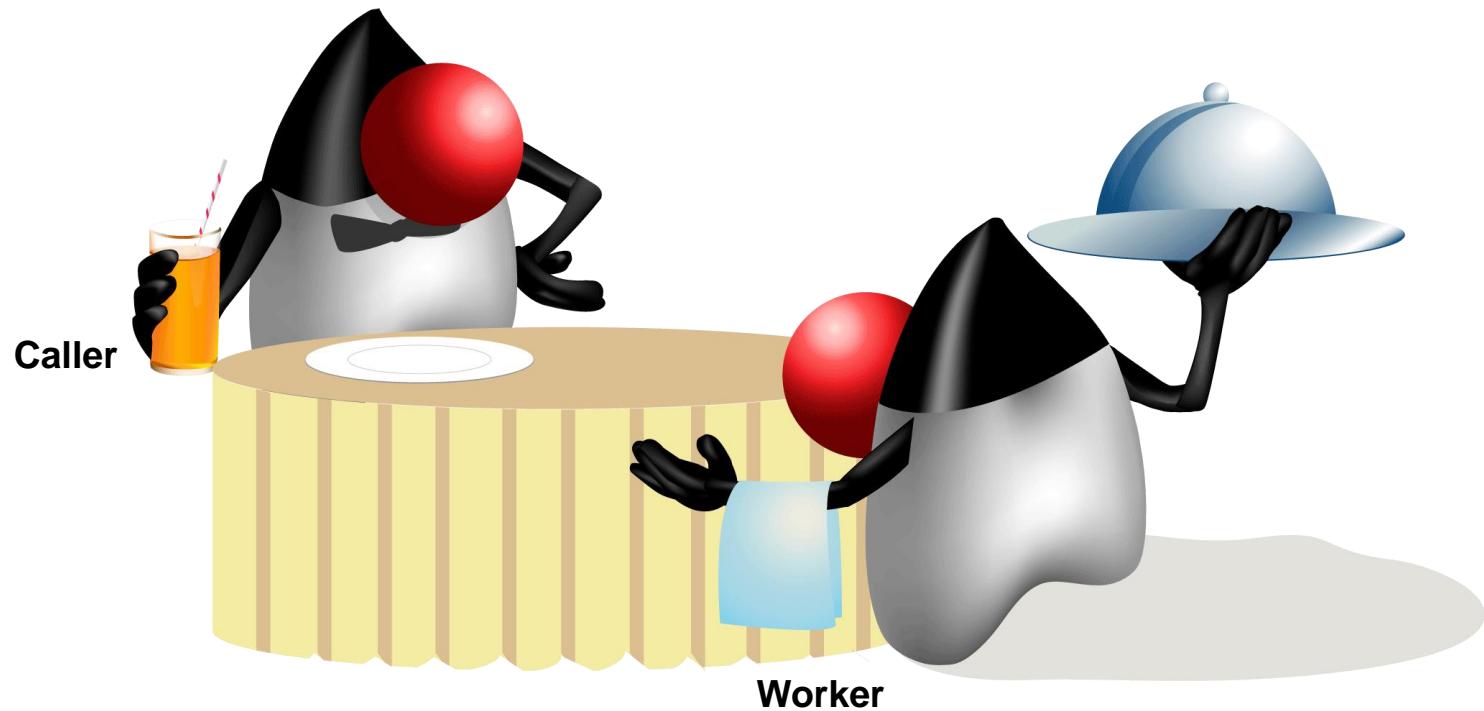
```
1 public class ShoppingCart {  
2     public static void main (String[] args) {  
3         Shirt myShirt = new Shirt();  
4         myShirt.display();  
5     }  
6 }
```

Diagram illustrating the components of the code:
myShirt: Reference variable
display(): Method
. (dot operator): Dot operator

Output:

```
Item description:-description required-  
Color Code: U  
Item price: 0.0
```

Caller and Worker Methods



A Constructor Method

A constructor method is a special method that is invoked when you create an object instance.

- It is called by using the `new` keyword.
- Its purpose is to instantiate an object of the class and store the reference in the reference variable.

```
Shirt myShirt = new Shirt();
```

Constructor method
is called.

- It has a unique method signature.

<modifier> ClassName ()

Writing and Calling a Constructor

```
1 public static void main(String[] args) {  
2     Shirt myShirt = new Shirt();  
3 }
```

```
1 public class Shirt {  
2     //Fields  
3     public String description;  
4     public char colorCode;  
5     public double price;  
6  
7     //Constructor  
8     public Shirt(){  
9         description = "--description required--";  
10        colorCode = 'U'  
11        price = 0.00;  
12    }  
13  
14    //Methods  
15    public void display(){  
16        System.out.println("Shirt description:" + description);  
17        System.out.println("Color Code: " + colorCode);  
18        System.out.println("Shirt price: " + price);  
19    } ...
```

Calling a Method in the Same Class

```
1  public class Shirt {  
2      public String description;  
3      public char colorCode;  
4      public double price;  
5  
6      public Shirt() {  
7          description = "--description required--";  
8          colorCode = 'U'  
9          price = 0.00;  
10  
11         display();           //Called normally  
12         this.display();      //Called using the 'this' keyword  
13     }  
14  
15     public void display(){  
16         System.out.println("Shirt description:" + description);  
17         System.out.println("Color Code: " + colorCode);  
18         System.out.println("Shirt price: " + price);  
19     }  
20 ...
```

Topics

- Using constructors and methods
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Method Arguments and Parameters

- An **argument** is a value that is passed during a method call:

```
Calculator calc = new Calculator();  
double denominator = 2.0  
calc.calculate(3, denominator); //should print 1.5
```

Arguments

- A **parameter** is a variable defined in the method declaration:

```
public void calculate(int x, double y) {  
    System.out.println(x/y);
```

3 2.0
Parameters

Method Parameter Examples

- Methods may have any number or type of parameters:

```
public void calculate0() {  
    System.out.println("No parameters");  
}
```

```
public void calculate1(int x) {  
    System.out.println(x/2.0);  
}
```

```
public void calculate2(int x, double y) {  
    System.out.println(x/y);  
}
```

```
public void calculate3(int x, double y, int z){  
    System.out.println(x/y +z);  
}
```

Method Return Types

- Variables can have values of many different types:

int
short
double
long
boolean
char
float
byte
String
int[]
shirt

- Method calls can also return values of many different types:

int
short
double
long
boolean
char
float
byte
String
int[]
shirt

- How to make a method return a value:
 - Declare the method to be a non-void return type.
 - Use the keyword `return` within a method, followed by a value.

Method Return Types Examples

- Methods must `return` data that matches their return type:

```
public void printString() {  
    System.out.println("Hello");  
}
```

Void methods cannot return values in Java.

```
public String returnString() {  
    return("Hello");  
}
```

```
public int sum(int x, int y){  
    return(x + y);  
}
```

```
public boolean isGreater(int x, int y){  
    return(x > y);  
}
```

Method Return Animation

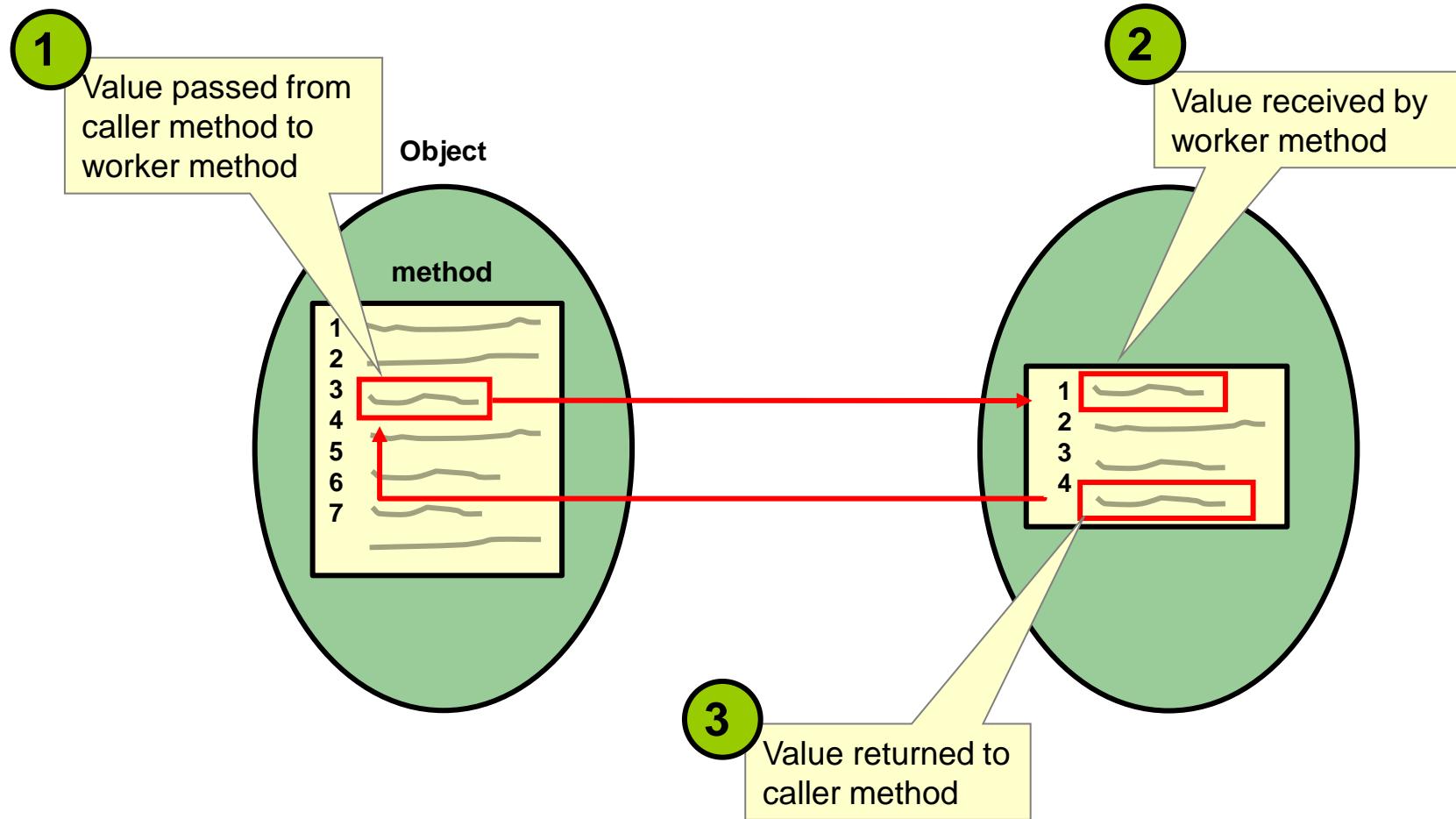
- The following code examples produce equivalent results:

```
public static void main(String[] args) {  
    int num1 = 1, num2 = 2;  
    int result = num1 + num2;  
    System.out.println(result);  
}
```

```
public static void main(String[] args) {  
    int num1 = 1, num2 = 2;  
    int result = sum(num1, num2);  
    System.out.println(result);  
}
```

```
public static int sum(int x, int y) {  
    return(x + y);  
}
```

Passing Arguments and Returning Values



More Examples

```
1 public void setCustomerServices() {  
2     String message ="Would you like to hear about "  
3             +"special deals in your area?";  
4     if (cust.isNewCustomer()) {  
5  
6         cust.sendEmail(message);  
7     }  
8 }
```

```
1 public class Customer{  
2     public boolean isNew;  
3  
4     public boolean isNewCustomer(){  
5         return isNew;           ————— Return a boolean  
6     }  
7     public void sendEmail(String message){  
8         // send email  
9     }  
10 }
```

String argument required

Code Without Methods

```
1 public static void main(String[] args) {
2     Shirt shirt01 = new Shirt();
3     Shirt shirt02 = new Shirt();
4     Shirt shirt03 = new Shirt();
5     Shirt shirt04 = new Shirt();
6
7     shirt01.description = "Sailor";
8     shirt01.colorCode = 'B';
9     shirt01.price = 30;
10
11    shirt02.description = "Sweatshirt";
12    shirt02.colorCode = 'G';
13    shirt02.price = 25;
14
15    shirt03.description = "Skull Tee";
16    shirt03.colorCode = 'B';
17    shirt03.price = 15;
18
19    shirt04.description = "Tropical";
20    shirt04.colorCode = 'R';
21    shirt04.price = 20;
22 }
```

Better Code with Methods

```
1 public static void main(String[] args) {  
2     Shirt shirt01 = new Shirt();  
3     Shirt shirt02 = new Shirt();  
4     Shirt shirt03 = new Shirt();  
5     Shirt shirt04 = new Shirt();  
6  
7     shirt01.setFields("Sailor", 'B', 30);  
8     shirt02.setFields("Sweatshirt", 'G', 25);  
9     shirt03.setFields("Skull Tee", 'B', 15);  
10    shirt04.setFields("Tropical", 'R', 20);  
11 }
```

```
1 public class Shirt {  
2     public String description;  
3     public char colorCode;  
4     public double price;  
5  
6     public void setFields(String desc, char color, double price) {  
7         this.description = desc;  
8         this.colorCode = color;  
9         this.price = price;  
10    }  
11 ...
```

Even Better Code with Methods

```
1 public static void main(String[] args) {  
2     Shirt shirt01 = new Shirt("Sailor", "Blue", 30);  
3     Shirt shirt02 = new Shirt("SweatShirt", "Green", 25);  
4     Shirt shirt03 = new Shirt("Skull Tee", "Blue", 15);  
5     Shirt shirt04 = new Shirt("Tropical", "Red", 20);  
6 }
```

```
1 public class Shirt {  
2     public String description;  
3     public char colorCode;  
4     public double price;  
5  
6     //Constructor  
7     public Shirt(String desc, String color, double price){  
8         setFields(desc, price);  
9         setColor(color);  
10    }  
11    public void setColor (String theColor){  
12        if (theColor.length() > 0)  
13            colorCode = theColor.charAt(0);  
14    }  
15    }  
16 }
```

Variable Scope

```
1 public class Shirt {  
2     public String description;  
3     public char colorCode;   
4     public double price;  
5  
6     public void setColor (String theColor) {  
7         if (theColor.length() > 0)  
8             colorCode = theColor.charAt(0);  
9     }  
10 }  
  
11  
12     public String getColor () {  
13         return theColor; //Cannot find symbol  
14     }  
15  
16 }
```

Instance variable (field)

Local variable

Scope of theColor

Not scope of theColor

A red circle with a diagonal slash is drawn over the word "theColor" in the line "return theColor;".

Advantages of Using Methods

Methods:

- Are reusable
- Make programs shorter and more readable
- Make development and maintenance quicker
- Allow separate objects to communicate and to distribute the work performed by the program

Exercise 8-1: Declare a `setColor` Method

In this exercise you:

- Declare a `setColor` method that takes a `char` as an argument
- In the `ShoppingCart` class, call the `setColor` method on `item1`
- Test the method with both a valid color and an invalid one



Topics

- Using constructors and methods
- Method arguments and return values
- **Using static methods and variables**
- Understanding how arguments are passed to a method
- Overloading a method

Java Puzzle Ball

Have you played through **Basic Puzzle 8**?

Consider the following:

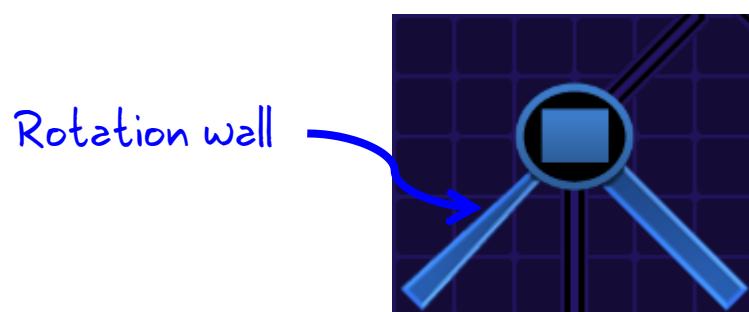
What happens when you rotate the blue wheel?

How else can you affect the rotation of bumpers?



Java Puzzle Ball Debrief

- What happens when you rotate the blue wheel?
 - The orientation of **all** blue bumpers change.
 - **All** blue bumpers share the orientation property of the wheel.
- How else can you affect the rotation of bumpers?
 - After the ball strikes a rotation wall, the rotation of an **individual** bumper changes.



Static Methods and Variables

The `static` modifier is applied to a method or variable.

It means the method/variable:

- Belongs to the *class* and is shared by all objects of that class
- Is *not unique* to an object instance
- Can be accessed without instantiating the class

Comparison:

- A **static variable** is shared by all objects in a class.
- An **instance variable** is unique to an individual object.

Example: Setting the Size for a New Item

```
1 public class ItemSizes {  
2     static final String mSmall = "Men's Small";  
3     static final String mMed = "Men's Medium";  
4 }
```

```
Item item1 = new Item();  
item1.setSize(ItemSizes.mMed);
```

Passing the static mMed variable
to the setSize method

```
1 public class Item {  
2     public String size;  
3     public void setSize(String sizeArg) {  
4         this.size = sizeArg;  
5     }  
6 }
```

Creating and Accessing Static Members

- To create a static variable or method:

```
static String mSmall;
```

```
static void setMSmall(String desc);
```

- To access a static variable or method:

- From another class

```
ItemSizes.mSmall;
```

```
ItemSizes.setMSmall("Men's Small");
```

- From within the class

```
mSmall;
```

```
setMSmall("Men's Small");
```

When to Use Static Methods or Fields

- Performing the operation on an individual object or associating the variable with a specific object type is not important.
- Accessing the variable or method before instantiating an object is important.
- The method or variable does not logically belong to an object, but possibly belongs to a utility class, such as the `Math` class, included in the Java API.
- Using constant values (such as `Math.PI`)

Some Rules About Static Fields and Methods

- Instance methods can access static methods or fields.
- Static methods cannot access instance methods or fields.
Why?

```
1 public class Item{  
2     int itemID;  
3     public Item() {  
4         setId();  
5     }  
6     static int getID() {  
7         // whose itemID??  
8     }
```

Static Fields and Methods vs. Instance Fields and Methods

```
public class Item{  
    static int staticItemID;  
    int instanceItemID;  
    static main() {  
        Item item01 = new Item();  
  
        1 staticItemId = 6; ✓  
        2 instanceItemID = 3 ✗  
        3 showItemID(); ✗  
        4 item01.showItemID(); ✓  
  
    }  
    showItemID() {  
        ...println(staticItemID);  
        ...println(instanceItemID);  
    }  
}
```

Object (instance)
referenced by item01.

```
static int staticItemID;  
int instanceItemID;  
static main() { ... }  
  
showItemID() {  
  
    5 ...println(staticItemID); ✓  
    6 ...println(instanceItemID); ✓  
}
```

Other instances
of Item

Static Methods and Variables in the Java API

Examples:

- Some functionality of the `Math` class:
 - Exponential
 - Logarithmic
 - Trigonometric
 - Random
 - Access to common mathematical constants, such as the value PI (`Math.PI`)
- Some functionality of the `System` class:
 - Retrieving environment variables
 - Access to the standard input and output streams
 - Exiting the current program (`System.exit` method)

Examining Static Variables in the JDK Libraries

A screenshot of a Java class browser interface. On the left, there's a tree view of packages: java.awt, java.awt.im, java.awt.image, java.awt.image.renderable, java.awt.print, java.beans, and java.io. Under java.io, the 'java.lang' package is highlighted with a red box. Inside 'java.lang', several classes are listed: Integer, Long, Math, Number, Object, Package, Process, ProcessBuilder, ProcessBuilder.Redirect, Runtime, RuntimePermission, SecurityManager, Short, StackTraceElement, StrictMath, String, StringBuffer, StringBuilder, and System. The 'System' class is also highlighted with a red box.

Field Detail

in

public static final InputStream in

The "standard" input stream. This stream is already open and ready to s

out

public static final PrintStream out

The "standard" output stream. This stream is already open and ready to a

For simple stand-alone Java applications, a typical way to write a line of c

System.out.println(data)

See the println methods in class PrintStream.

See Also:

PrintStream.println(), PrintStream.println(boolean), Pri
PrintStream.println(int), PrintStream.println(long), Pri

err

public static final PrintStream err

The "standard" error output stream. This stream is already open and rea

Typically this stream corresponds to display output or another output des
should come to the immediate attention of a user even if the principal ou

out is a static
field of System
and contains
end is an
object
reference to a
PrintStream
object.

System is a class in java.lang.

Using Static Variables and Methods: System.out.println

java.lang
Class System
java.lang.Object
 java.lang.System

public final class **System**
extends Object

Field Summary

Fields

Modifier and Type	Field and Description
static PrintStream	err The "standard" error output stream.
static InputStream	in The "standard" input stream.
static PrintStream	out The "standard" output stream.

The field, out, on System is
of type PrintStream.

void	print(Object obj) Prints an object.
void	print(String s) Prints a string.
PrintStream	printf(Locale l, String format, Object... args) A convenience method to write a formatted string to this output
void	println(double x) Prints a double and then terminate the line.
void	println(float x) Prints a float and then terminate the line.
void	println(int x) Prints an integer and then terminate the line.
void	println(long x) Prints a long and then terminate the line.
void	println(Object x) Prints an Object and then terminate the line.
void	println(String x) Prints a String and then terminate the line.

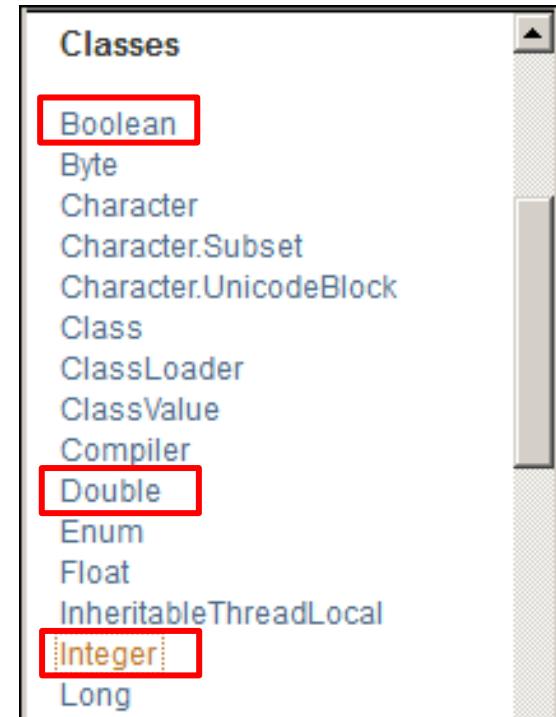
Some of the methods
of PrintStream

More Static Fields and Methods in the Java API

Java provides wrapper classes for each of the primitive data types.

- Boolean: Contains a single field of type boolean
- Double: Contains a single field of type double
- Integer: Contains a single field of type int

They also provide utility methods to work with the data.



Converting Data Values

- Methods often need to convert an argument to a different type.
- Most of the object classes in the JDK provide various conversion methods.

Examples:

- Converting a String to an int

```
int myInt1 = Integer.parseInt(s_Num);
```

- Converting a String to a double

```
double myDbl = Double.parseDouble(s_Num);
```

- Converting a String to boolean

```
boolean myBool = Boolean.valueOf(s_Bool);
```

Topics

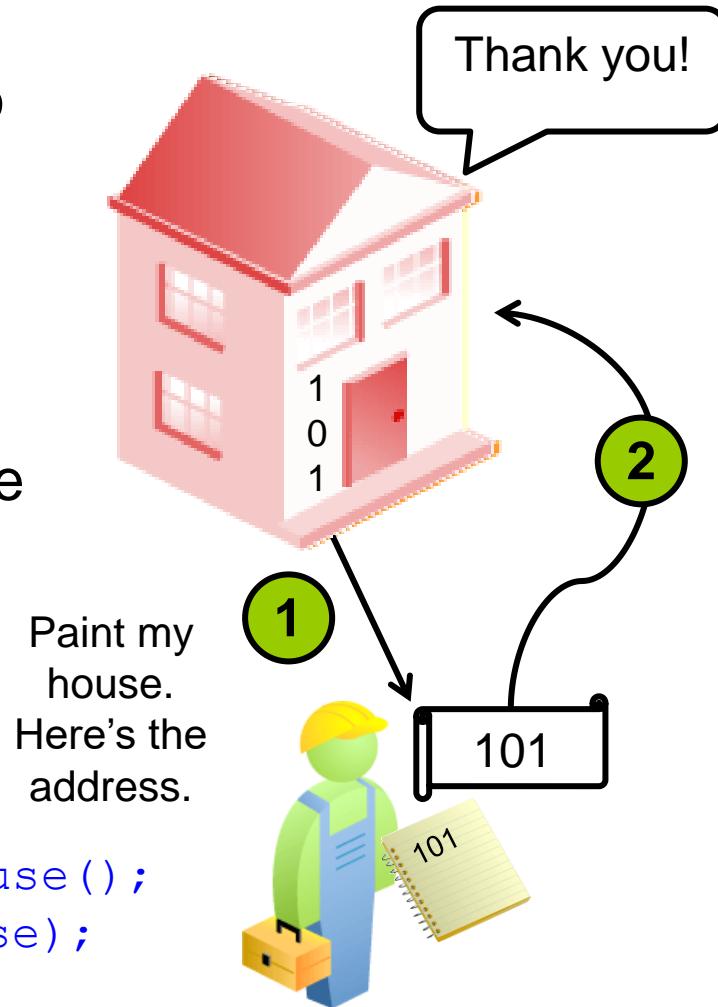
- Using constructors and methods
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Passing an Object Reference

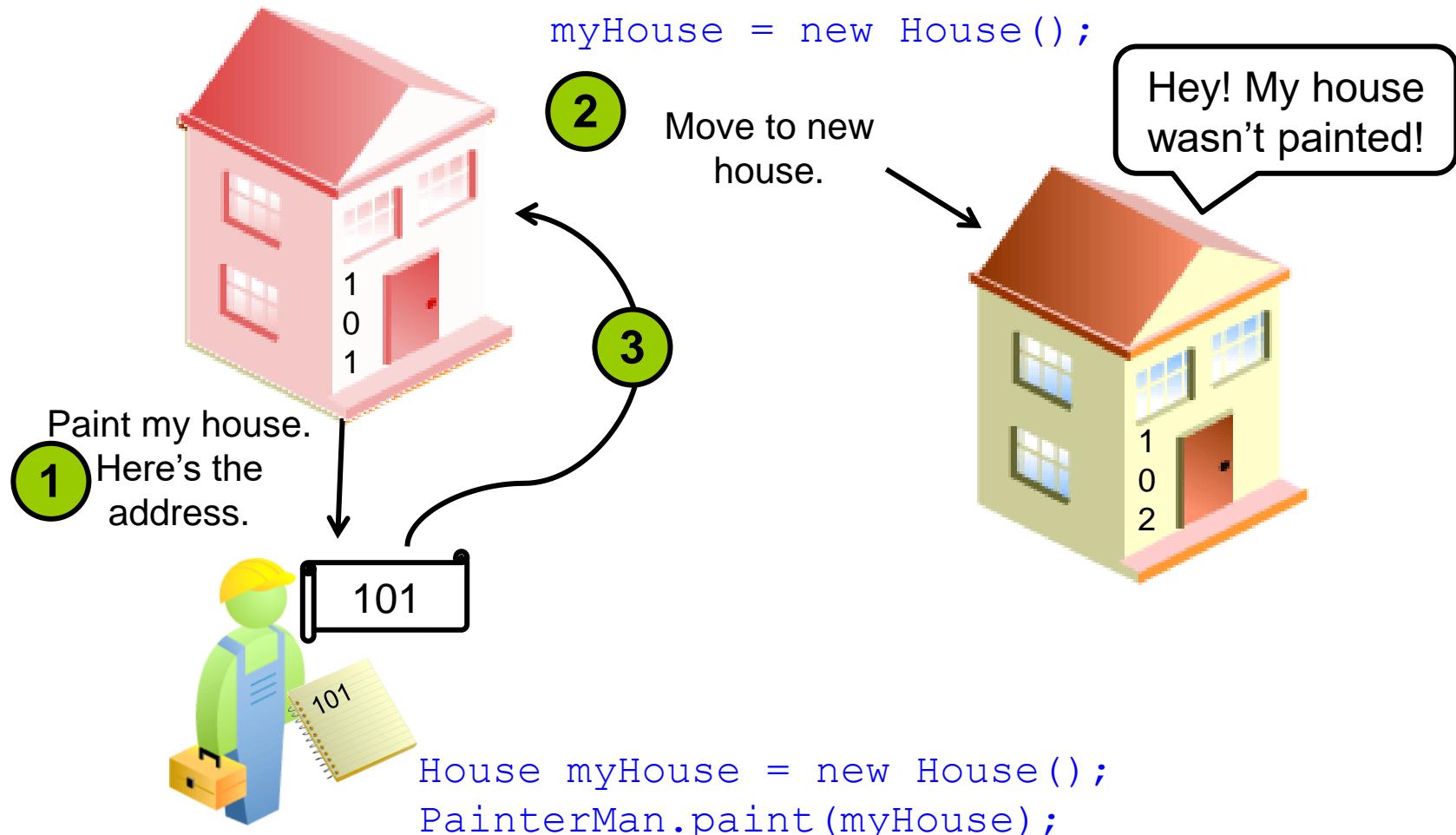
An object reference is similar to a house address. When it is passed to a method:

- The object itself is not passed
- The method can access the object using the reference
- The method can act upon the object

```
House myHouse = new House();  
PainterMan.paint(myHouse);
```



What If There Is a New Object?



A Shopping Cart Code Example

```
1 public class ShoppingCart {  
2     public static void main (String[] args) {  
3         Shirt myShirt = new Shirt();  
4         System.out.println("Shirt color: " + myShirt.colorCode);  
5         changeShirtColor(myShirt, 'B');  
6         System.out.println("Shirt color: " + myShirt.colorCode);  
7     }  
8     public static void changeShirtColor(Shirt theShirt, char color) {  
9         theShirt.colorCode = color;    }  
10 }
```

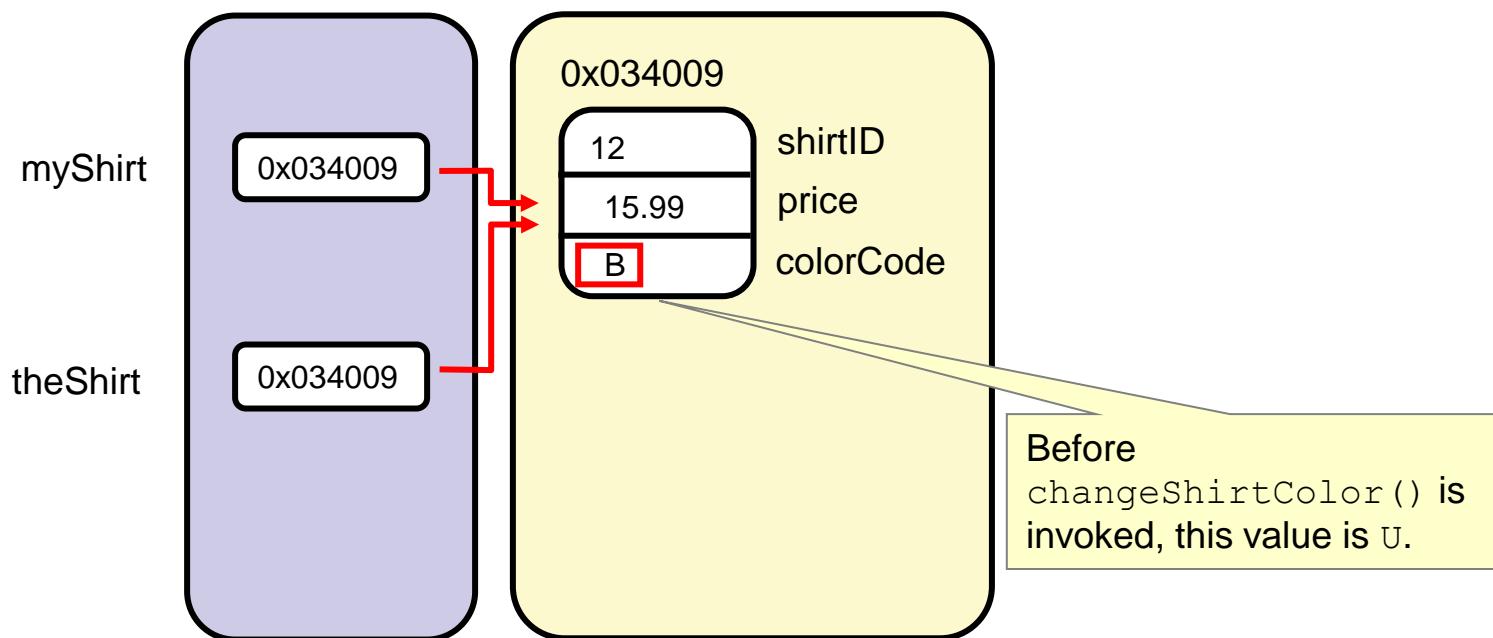
theShirt is a new reference of type Shirt.

Output:

```
Shirt color: U  
Shirt color: B
```

Passing by Value

```
Shirt myShirt = new Shirt();  
changeShirtColor(myShirt, 'B');
```



Reassigning the Reference

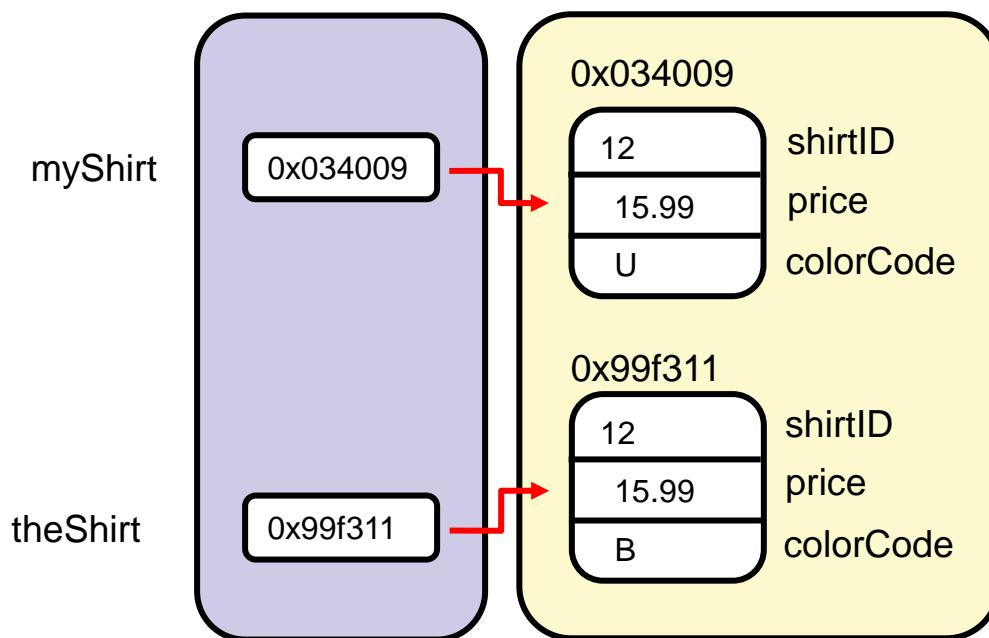
```
1 public class ShoppingCart {  
2     public static void main (String[] args) {  
3         Shirt myShirt = new Shirt();  
4         System.out.println("Shirt color: " + myShirt.colorCode);  
5         changeShirtColor(myShirt, 'B');  
6         System.out.println("Shirt color: " + myShirt.colorCode);  
7     }  
8     public static void changeShirtColor(Shirt theShirt, char color) {  
9         theShirt = new Shirt();  
10        theShirt.colorCode = color;  
11    }  
12 }
```

Output:

```
Shirt color: U  
Shirt color: U
```

Passing by Value

```
Shirt myShirt = new Shirt();  
changeShirtColor(myShirt, 'B');
```



Topics

- Using constructors and methods
- Method arguments and return values
- Using static methods and variables
- Understanding how arguments are passed to a method
- Overloading a method

Method Overloading

Overloaded methods:

- Have the same name
- Have different signatures
 - The **number** of parameters
 - The **types** of parameters
 - The **order** of parameters
- May have different functionality or similar functionality
- Are widely used in the foundation classes

Using Method Overloading

```
1 public final class Calculator {  
2  
3     public static int sum(int num1, int num2) {  
4         System.out.println("Method One");  
5         return num1 + num2;  
6     }  
7  
8     public static float sum(float num1, float num2) {  
9         System.out.println("Method Two");  
10        return num1 + num2;  
11    }  
12    public static float sum(int num1, float num2) {  
13        System.out.println("Method Three");  
14        return num1 + num2;  
15    }  
16}
```

The method type

The method signature

Using Method Overloading

```
1 public class CalculatorTest {  
2  
3     public static void main(String[] args) {  
4  
5         int totalOne = Calculator.sum(2, 3);  
6         System.out.println("The total is " + totalOne);  
7  
8         float totalTwo = Calculator.sum(15.99F, 12.85F);  
9         System.out.println(totalTwo);  
10  
11         float totalThree = Calculator.sum(2, 12.85F);  
12         System.out.println(totalThree);  
13     }  
14 }
```

Method Overloading and the Java API

Method	Use
<code>void println()</code>	Terminates the current line by writing the line separator string
<code>void println(boolean x)</code>	Prints a boolean value and then terminates the line
<code>void println(char x)</code>	Prints a character and then terminates the line
<code>void println(char[] x)</code>	Prints an array of characters and then terminates the line

Exercise 8-2: Overload a `setItemFields` Method

In this exercise, you create an overloaded method in the Item class:

- `setItemFields` with three parameters that returns `void`
- `setItemFields` with four parameters that returns an `int`
- Then you invoke these from `ShoppingCart`.



Quiz

Which method corresponds to the following method call?

```
myPerson.printValues(100, 147.7F, "lavender");
```

- a. public void printValues (int i, float f)
- b. public void printValues (i, float f, s)
- c. public void printValues (int i, float f, String s)
- d. public void printValues (float f, String s, int i)

Summary

In this lesson, you should have learned how to:

- Add an argument to a method
- Instantiate a class and call a method
- Overload a method
- Work with static methods and variables
- Convert data values using Integer, Double, and Boolean object types



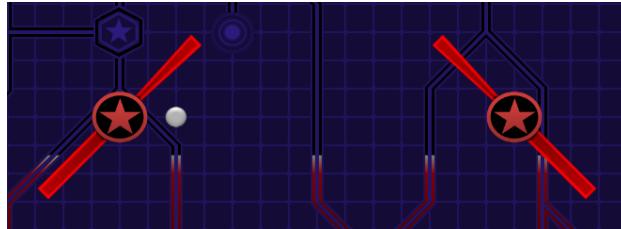
Challenge Questions: Java Puzzle Ball



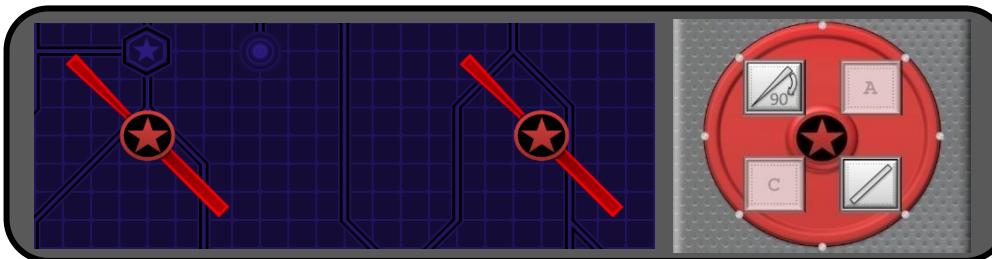
Which of the scenarios below reflect the behavior of:

- A static variable?
- An instance variable?

1. A single bumper rotates after being struck by the ball.



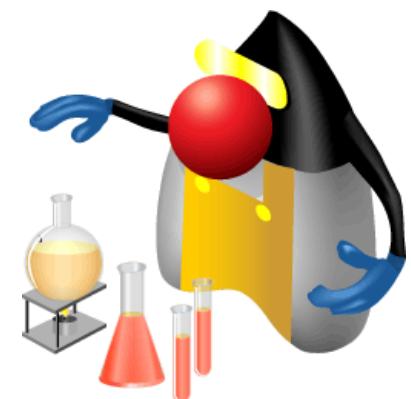
2. Rotating the red wheel changes the orientation of all red bumpers.



Practice 8-1 Overview: Using Methods

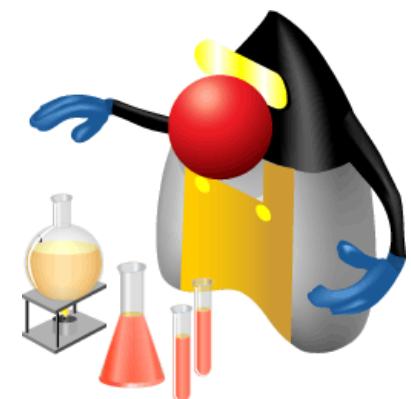
This practice covers the following topics:

- Creating a static method, `createTeams`, to return an array of teams
- Creating another static method, `createGames`, that takes an array of teams and returns an array of games



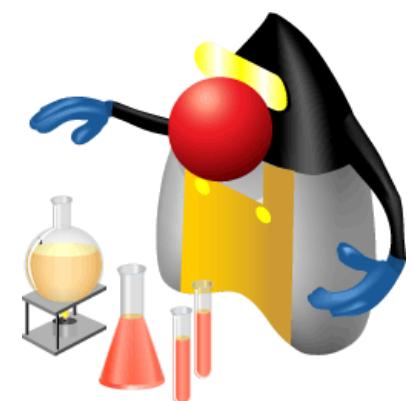
Practice 8-2 Overview: Creating Game Data Randomly

This practice covers creating a method for playing a soccer game that randomly creates Goal objects.



Practice 8-3 Overview: Creating Overloaded Methods

This practice covers overloading a method.



Using Encapsulation

9

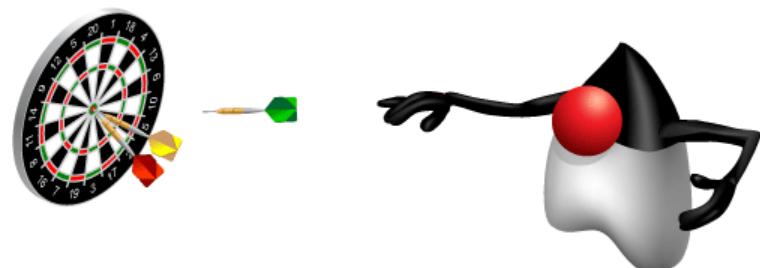
Interactive Quizzes



Objectives

After completing this lesson, you should be able to:

- Use an access modifier to make fields and methods private
- Create get and set methods to control access to private fields
- Define encapsulation as “information hiding”
- Implement encapsulation in a class using the NetBeans refactor feature
- Create an overloaded constructor and use it to instantiate an object



Topics

- Access control
- Encapsulation
- Overloading constructors

What Is Access Control?

Access control allows you to:

- Hide fields and methods from other classes
- Determine how internal data gets changed
- Keep the implementation separate from the public interface

— Public interface:

```
setPrice( Customer cust)
```

— Implementation:

```
public void setPrice(Customer cust) {  
    // set price discount relative to customer  
}
```

Access Modifiers

- `public`: Accessible by anyone
- `private`: Accessible only within the class

```
1 public class Item {  
2     // Base price  
3     private double price = 15.50;  
4  
5     public void setPrice(Customer cust) {  
6         if (cust.hasLoyaltyDiscount()) {  
7             price = price*.85; }  
8     }  
9 }
```



Access from Another Class

```
1 public class Item {  
2     private double price = 15.50;  
3     public void setPrice(Customer cust) {  
4         if (cust.hasLoyaltyDiscount()) {  
5             price = price*.85; }  
6     }  
7 }  
8 public class Order{  
9     public static void main(String args[]) {  
10         Customer cust = new Customer(int ID);  
11         Item item = new Item(); └─ Won't compile  
12         item.price = 10.00; └─  
13         item.setPrice(cust); └─ You don't need to know  
14     }                                         how setPrice works in  
15 }
```

Won't compile

You don't need to know how setPrice works in order to use it.

Another Example

The data type of the field does not match the data type of the data used to set the field.

```
1 private int phone;
2 public void setPhoneNumber(String s_num) {
3     // parse out the dashes and parentheses from the
4     // String first
5     this.phone = Integer.parseInt(s_num);
6 }
```

Using Access Control on Methods

```
1 public class Item {  
2     private int id;  
3     private String desc;  
4     private double price;  
5     private static int nextId = 1;  
6  
7     public Item(){  
8         setId();      
9         desc = "--description required--";  
10        price = 0.00;  
11    }  
12  
13    private void setId() {  
14        id = Item.nextId++;  
15    }  
16}
```

Called from within a public method

Private method

Topics

- Access control
- Encapsulation
- Overloading constructors

Encapsulation

- Encapsulation means hiding object fields. It uses access control to hide the fields.
 - Safe access is provided by getter and setter methods.
 - In setter methods, use code to ensure that values are valid.
- Encapsulation mandates programming to the interface:
 - A method can change the data type to match the field.
 - A class can be changed as long as interface remains same.
- Encapsulation encourages good object-oriented (OO) design.

Get and Set Methods

```
1 public class Shirt {
2     private int shirtID = 0;          // Default ID for the shirt
3     private String description = "-description required-"; // default
4     private char colorCode = 'U'; //R=Red, B=Blue, G=Green, U=Unset
5     private double price = 0.0;      // Default price for all items
6
7     public char getColorCode() {
8         return colorCode;
9     }
10    public void setColorCode(char newCode) {
11        colorCode = newCode;
12    }
13    // Additional get and set methods for shirtID, description,
14    // and price would follow
15
16 } // end of class
```

Why Use Setter and Getter Methods?

```
1 public class ShirtTest {  
2     public static void main (String[] args) {  
3         Shirt theShirt = new Shirt();  
4         char colorCode;  
5         // Set a valid colorCode  
6         theShirt.setColorCode ('R');  
7         colorCode = theShirt.getColorCode ();  
8         System.out.println ("Color Code: " + colorCode);  
9         // Set an invalid color code  
10        theShirt.setColorCode ('Z');  Not a valid color code  
11        colorCode = theShirt.getColorCode ();  
12        System.out.println ("Color Code: " + colorCode);  
13    }  
14 ...
```

Output:

```
Color Code: R  
Color Code: Z
```

Setter Method with Checking

```
15  public void setColorCode(char newCode) {  
16      if (newCode == 'R') {  
17          colorCode = newCode;  
18          return;  
19      }  
20      if (newCode == 'G') {  
21          colorCode = newCode;  
22          return;  
23      }  
24      if (newCode == 'B') {  
25          colorCode = newCode;  
26          return;  
27      }  
28      System.out.println("Invalid colorCode. Use R, G, or B");  
29  }  
30}
```

Using Setter and Getter Methods

```
1 public class ShirtTest {  
2     public static void main (String[] args) {  
3         Shirt theShirt = new Shirt();  
4         System.out.println("Color Code: " + theShirt.getColorCode());  
5  
6         // Try to set an invalid color code  
7         Shirt1.setColorCode('Z');      ————— Not a valid color code  
8         System.out.println("Color Code: " + theShirt.getColorCode());  
9     }  
}
```

Output:

Color Code: U ————— Before call to setColorCode() – shows default value
Invalid colorCode. Use R, G, or B ————— call to setColorCode prints error message
Color Code: U ————— colorCode not modified by invalid argument passed to setColorCode()

Exercise 9-1: Encapsulate a Class

In this exercise, you encapsulate the `Customer` class.

- Change access modifiers so that fields can be read or changed only through public methods.
- Allow the `ssn` field to be read but not modified.



Topics

- Access control
- Encapsulation
- Overloading constructors

Initializing a Shirt Object

Explicitly:

```
1 public class ShirtTest {  
2     public static void main (String[] args) {  
3         Shirt theShirt = new Shirt();  
4  
5         // Set values for the Shirt  
6         theShirt.setColorCode ('R');  
7         theShirt.setDescription ("Outdoors shirt");  
8         theShirt.price (39.99);  
9     }  
10 }
```

Using a constructor:

```
Shirt theShirt = new Shirt('R', "Outdoors shirt", 39.99);
```

Constructors

- Constructors are usually used to initialize fields in an object.
 - They can receive arguments.
 - When you create a constructor with arguments, it removes the default no-argument constructor.

Shirt Constructor with Arguments

```
1 public class Shirt {  
2     public int shirtID = 0;          // Default ID for the shirt  
3     public String description = "-description required-"; // default  
4     private char colorCode = 'U'; //R=Red, B=Blue, G=Green, U=Unset  
5     public double price = 0.0;      // Default price all items  
6  
7     // This constructor takes three argument  
8     public Shirt(char colorCode, String desc, double price ) {  
9         setColorCode(colorCode);  
10        setDescription(desc);  
11        setPrice(price);  
12    }
```

Default Constructor and Constructor with Args

When you create a constructor with arguments, the default constructor is no longer created by the compiler.

```
// default constructor  
public Shirt ()
```



```
// Constructor with args
```

```
public Shirt (char color, String desc, double price)
```

This constructor is not in the source code. It only exists if no constructor is explicitly defined.

```
6  / **  
7  *  
8  cannot find symbol  
9  symbol: constructor Shirt()  
10 location: class Shirt  
11 --  
12 (Alt-Enter shows hints)  
13  
14  
15
```

```
main (String args[]) {  
    myShirt = new Shirt();
```

Overloading Constructors

```
1 public class Shirt {  
2     ... //fields  
3  
4     // No-argument constructor  
5     public Shirt() {  
6         setColorCode('U');  
7     }  
8     // 1 argument constructor  
9     public Shirt(char colorCode) {  
10        setColorCode(colorCode);  
11    }  
12    // 2 argument constructor  
13    public Shirt(char colorCode, double price) {  
14        this(colorCode);  
15        setPrice(price);  
16    }
```

If required, must be added explicitly

Calling the 1 argument
constructor

Quiz

What is the default constructor for the following class?

```
public class Penny {  
    String name = "lane";  
}
```

- a. public Penny(String name)
- b. public Penny()
- c. class()
- d. String()
- e. private Penny()

Exercise 9-2: Create an Overloaded Constructor

In this exercise, you:

- Add an overloaded constructor to the Customer class
- Create a new Customer object by calling the overloaded constructor



Summary

In this lesson, you should have learned how to:

- Use public and private access modifiers
- Restrict access to fields and methods using encapsulation
- Implement encapsulation in a class
- Overload a constructor by adding method parameters to a constructor



Play Time!

Play **Basic Puzzle 12** before the next lesson titled “More on Conditionals.”

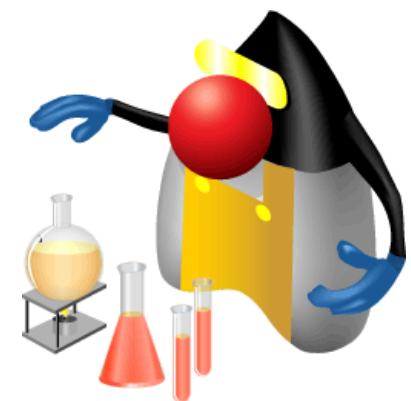
Consider the following:

[What happens if the ball strikes the blade?](#)



Practice 9-1 Overview: Encapsulating Fields

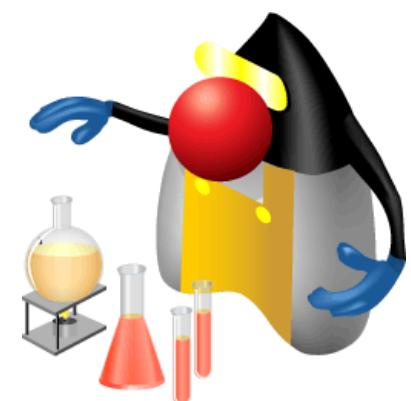
This practice covers using the NetBeans refactor feature to encapsulate the fields of several classes from the Soccer application.



Practice 9-2 Overview: Creating Overloaded Constructors

This practice covers the following topics:

- Creating overloaded constructors for several classes of the Soccer application
- Initializing fields within the custom constructor methods



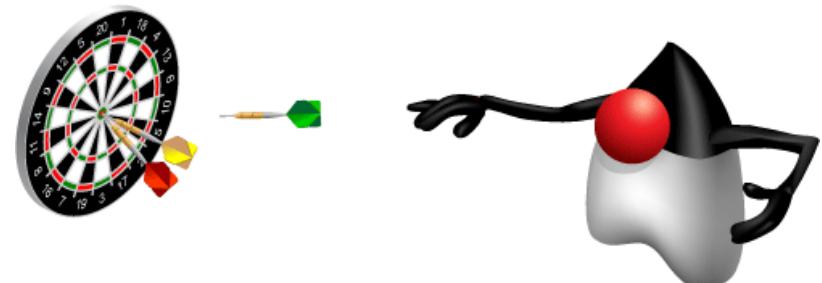
10

More on Conditionals

Objectives

After completing this lesson, you should be able to:

- Correctly use all of the conditional operators
- Test equality between string values
- Chain an `if/else` statement to achieve the desired result
- Use a `switch` statement to achieve the desired result
- Debug your Java code by using the NetBeans debugger to step through code line by line and view variable values



Topics

- Relational and conditional operators
- More ways to use if/else statements
- Using a switch statement
- Using the NetBeans debugger

Review: Relational Operators

Condition	Operator	Example
Is equal to	<code>==</code>	<code>int i=1; (i == 1)</code>
Is not equal to	<code>!=</code>	<code>int i=2; (i != 1)</code>
Is less than	<code><</code>	<code>int i=0; (i < 1)</code>
Is less than or equal to	<code><=</code>	<code>int i=1; (i <= 1)</code>
Is greater than	<code>></code>	<code>int i=2; (i > 1)</code>
Is greater than or equal to	<code>>=</code>	<code>int i=1; (i >= 1)</code>

Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = "Fred Smith";  
    public String name2 = "Sam Smith";  
  
    public void areNamesEqual() {  
        if (name1.equals(name2)) {  
            System.out.println("Same name.");  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = "Fred Smith";  
    public String name2 = "fred smith";  
  
    public void areNamesEqual() {  
        if (name1.equalsIgnoreCase(name2)) {  
            System.out.println("Same name."); ✓  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = "Fred Smith";  
    public String name2 = "Fred Smith";  
  
    public void areNamesEqual() {  
        if (name1 == name2) {  
            System.out.println("Same name."); ✓  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



Testing Equality Between String variables

Example:

```
public class Employees {  
  
    public String name1 = new String("Fred Smith");  
    public String name2 = new String("Fred Smith");  
  
    public void areNamesEqual() {  
        if (name1 == name2) {  
            System.out.println("Same name.");  
        }  
        else {  
            System.out.println("Different name.");  
        }  
    }  
}
```



Common Conditional Operators

Operation	Operator	Example
If one condition AND another condition	&&	<pre>int i = 2; int j = 8; ((i < 1) && (j > 6))</pre>
If either one condition OR another condition		<pre>int i = 2; int j = 8; ((i < 1) (j > 10))</pre>
NOT	!	<pre>int i = 2; (! (i < 3))</pre>

Ternary Conditional Operator

Operation	Operator	Example
If some condition is true, assign the value of value1 to the result. Otherwise, assign the value of value2 to the result.	?:	condition ? value1 : value2 Example: int x = 2, y = 5, z = 0; z = (y < x) ? x : y;

Equivalent statements

```
z = (y < x) ? x : y;
```

```
if(y<x) {  
    z=x;  
}  
else{  
    z=y;  
}
```

Using the Ternary Operator

Advantage: Usable in a single line

```
int numberOfGoals = 1;  
String s = (numberOfGoals==1 ? "goal" : "goals");  
  
System.out.println("I scored " +numberOfGoals + " "  
+s );
```

Advantage: Place the operation directly within an expression

```
int numberOfGoals = 1;  
  
System.out.println("I scored " +numberOfGoals + " "  
+(numberOfGoals==1 ? "goal" : "goals") );
```

Disadvantage: Can have only two potential results

```
(numberOfGoals==1 ? "goal" : "goals" : "More goals");
```

The diagram illustrates the structure of the ternary operator. It consists of four parts separated by colons: 1) A boolean expression underlined with a blue bracket. 2) The value to return if the boolean is true, also underlined with a blue bracket. 3) The value to return if the boolean is false, also underlined with a blue bracket. 4) A fourth part, which is the value returned if the boolean is neither true nor false (implied by the question marks). This fourth part is crossed out with a large red circle and a diagonal slash.

Exercise 10-1: Using the Ternary Operator

In this exercise, you use a ternary operator to duplicate the same logic shown in this `if/else` statement:

```
01     int x = 4, y = 9;  
02     if ((y / x) < 3) {  
03         x += y;  
04     }  
05     else x *= y;
```



Topics

- Relational and conditional operators
- More ways to use if/else statements
- Using a switch statement
- Using the NetBeans debugger

Java Puzzle Ball

Have you played through **Basic Puzzle 12**?

Consider the following:

What happens *if* the ball strikes the blade?



Java Puzzle Ball Debrief

- What happens if the ball strikes the blade?
 - if the ball strikes the blade:
 - Transform the ball into a blade
 - if the ball is a blade && it strikes the fan:
 - The ball is blown in the direction of the fan
 - if the ball is a blade && it strikes any object other than the fan || blade:
 - Destroy that object
 - Transform the ball back into a ball



Handling Complex Conditions with a Chained if Construct

The chained `if` statement:

- Connects multiple conditions together into a single construct
- Often contains nested `if` statements
- Tends to be confusing to read and hard to maintain

Determining the Number of Days in a Month

```
01 if (month == 1 || month == 3 || month == 5 || month == 7  
02     || month == 8 || month == 10 || month == 12) {  
03     System.out.println("31 days in the month.");  
04 }  
05 else if (month == 2) {  
06     if (!isLeapYear) {  
07         System.out.println("28 days in the month.");  
08     } else System.out.println("29 days in the month.");  
09 }  
10 else if (month == 4 || month == 6 || month == 9  
11     || month == 11) {  
12     System.out.println("30 days in the month.");  
13 }  
14 else  
15     System.out.println("Invalid month.");
```

Chaining if/else Constructs

Syntax:

```
01  if <condition1> {  
02      //code_block1  
03  }  
04  else if <condition2> {  
05      // code_block2  
06  }  
07  else {  
08      // default_code  
09  }
```

Exercise 10-2: Chaining if Statements

In this exercise, you write a `calcDiscount` method that determines the discount for three different customer types:

- Nonprofits get a discount of 10% if total > 900, else 8%.
- Private customers get a discount of 7% if total > 900, else no discount.
- Corporations get a discount of 8% if total > 500, else 5%.



Topics

- Relational and conditional operators
- More ways to use `if/else` statements
- **Using a `switch` statement**
- Using the NetBeans debugger

Handling Complex Conditions with a switch Statement

The `switch` statement:

- Is a streamlined version of chained `if` statements
- Is easier to read and maintain
- Offers better performance

Coding Complex Conditions: switch

```
01 switch (month) {  
02     case 1: case 3: case 5: case 7:  
03     case 8: case 10: case 12:  
04         System.out.println("31 days in the month.");  
05         break;  
06     case 2:  
07         if (!isLeapYear) {  
08             System.out.println("28 days in the month.");  
09         } else  
10             System.out.println("29 days in the month.");  
11         break;  
12     case 4: case 6: case 9: case 11:  
14         System.out.println("30 days in the month.");  
15         break;  
16     default:  
17         System.out.println("Invalid month.");  
18 }
```

switch Statement Syntax

Syntax:

```
01  switch (<variable or expression>) {  
02      case <literal value>:  
03          //code_block1  
04          [break;]  
05      case <literal value>:  
06          // code_block2  
07          [break;]  
08      default:  
09          //default_code  
10  {
```

When to Use switch Constructs

Use when you are testing:

- Equality (not a range)
- A *single* value
- Against fixed known values at compile time
- The following data types:
 - Primitive data types: int, short, byte, char
 - String or enum (enumerated types)
 - Wrapper classes (special classes that wrap certain primitive types): Integer, Short, Byte and Character

Only a single value can be tested.

```
01 switch (month) {  
02     case 1: case 3: case 5: case 7:  
03     case 8: case 10: case 12:  
04         System.out.println("31 days in the month."); } Known values  
05         break;  
06     case 2:  
07         if (!isLeapYear) {
```

Exercise 10-3: Using switch Construct

In this exercise, you modify the `calcDiscount` method to use a `switch` construct, instead of a chained `if` construct:

- Use a ternary operator instead of a nested `if` within each case block.



Quiz

Which of the following sentences describe a valid case to test in a switch construct?

- a. The switch construct tests whether values are greater than or less than a single value.
- b. Variable or expression where the expression returns a supported switch type.
- c. The switch construct can test the value of a float, double, boolean, or String.
- d. The switch construct tests the outcome of a boolean expression.

Topics

- Relational and conditional operators
- More ways to use if/else statements
- Using a switch statement
- Using the NetBeans debugger

Working with an IDE Debugger

Most IDEs provide a debugger. They are helpful to solve:

- Logic problems
 - (Why am I not getting the result I expect?)
- Runtime errors
 - (Why is there a `NullPointerException`?)



Debugger Basics

- Breakpoints:
 - Are stopping points that you set on a line of code
 - Stop execution at that line so you can view the state of the application
- Stepping through code:
 - After stopping at a break point, you can “walk” through your code, line by line to see how things change.
- Variables:
 - You can view or change the value of a variable at run time.
- Output:
 - You can view the System output at any time.

Setting Breakpoints

- To set breakpoints, click in the margin of a line of code.
- You can set multiple breakpoints in multiple classes.

```
3  public class DebugTestIfElse {  
4      public static void main(String[] args) {  
5          int month =11;  
6          boolean isLeapYear = true;  
7  
8          if(month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 || month == 12){  
9              System.out.println("31 days in the month.");  
10         }  
11         else if(month == 2){  
12             if(!isLeapYear){  
13                 System.out.println("28 days in the month.");  
14             }  
15             else{  
16                 System.out.println("29 days in the month.");  
17             }  
18         }  
19         else if(month == 4 || month ==6 || month == 9 || month ==11){  
20             System.out.println("30 days in the month.");  
21         }  
22         else{  
23             System.out.println("Invalid month");  
24         }  
25     }  
26 }
```

The Debug Toolbar

1. Start debugger
2. Stop debug session
3. Pause debug session
4. Continue running
5. Step over
6. Step over an expression
7. Step into
8. Step out of



Viewing Variables

Breakpoint

Current line of execution

```
switch(month) {  
    case 1: case 3: case 5: case 7:  
    case 8: case 10: case 12:  
        System.out.println("31 days in the month.");  
        break;  
    case 2:  
        if(!isLeapYear){  
            System.out.println("28 days in the month.");  
        }  
        else{  
            System.out.println("29 days in the month.");  
        }  
        break;  
    case 4: case 6: case 9: case 11:  
}
```

DebutTestSwitch > main > switch (month) > case 2: >

Name	Type	Value
Static	String[]	#72(length=0)
args	String[]	2
month	int	true
isLeapYear	boolean	

Value of variables

Summary

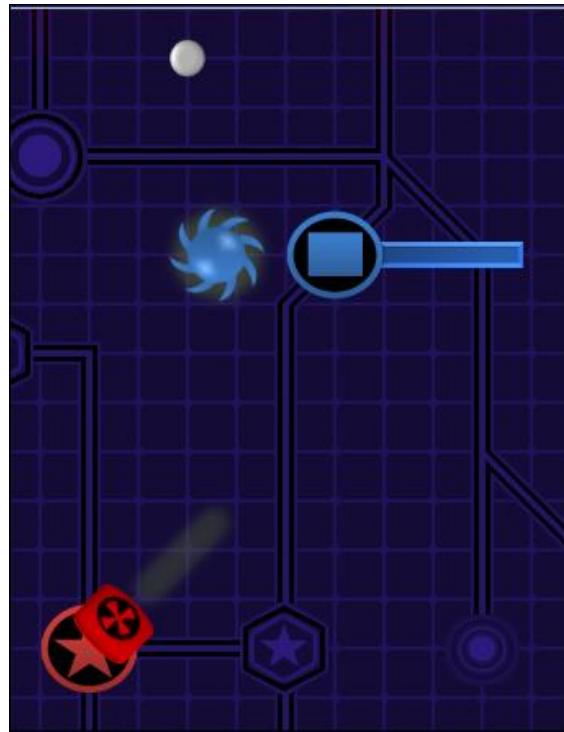
In this lesson, you should have learned how to:

- Use a `ternary` statement
- Test equality between strings
- Chain an `if/else` statement
- Use a `switch` statement
- Use the NetBeans debugger



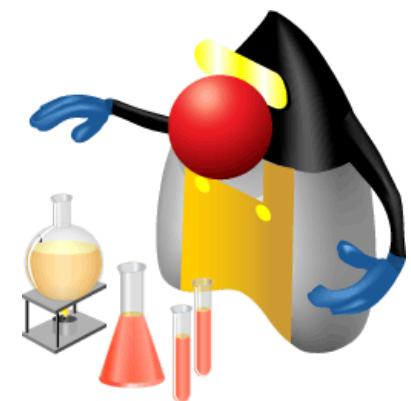
Challenge Question: Java Puzzle Ball

What type of conditional construct would you use to handle the behavior of the blade?



Practice 10-1 Overview: Using Conditional Statements

This practice covers enhancing the `getDescription` method of the Game class to announce the name of the winning team.



Practice 10-2 Overview: Debugging

This practice covers the following topics:

- Enhancing the `showBestTeam` method to differentiate between teams with the same number of points
- Using the NetBeans debugger to step through the code line by line

Working with Arrays, Loops, and Dates

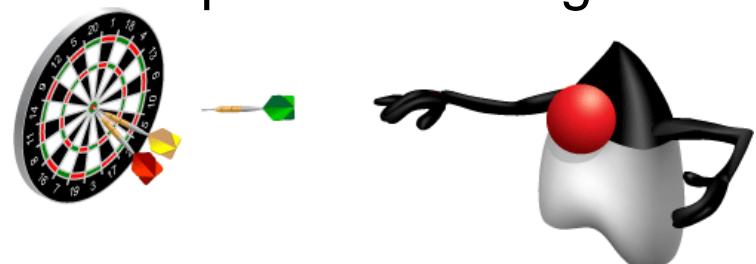
Interactive Quizzes



Objectives

After completing this lesson, you should be able to:

- Create a `java.time.LocalDateTime` object to show the current date and time
- Parse the `args` array of the `main` method
- Correctly declare and instantiate a two-dimensional array
- Code a nested `while` loop to achieve the desired result
- Use a nested `for` loop to process a two-dimensional array
- Code a `do/while` loop to achieve the desired result
- Use an `ArrayList` to store and manipulate lists of Objects
- Evaluate and select the best type of loop to use for a given programming requirement



Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- The `ArrayList` class



Displaying a Date

```
LocalDate myDate = LocalDate.now();  
System.out.println("Today's date: "+ myDate);
```

Output: 2013-12-20

- `LocalDate` belongs to the package `java.time`.
- The `now` method returns today's date.
- This example uses the default format for the default time zone.



Class Names and the Import Statement

- Date classes are in the package `java.time`.
- To refer to one of these classes in your code, you can fully qualify

`java.time.LocalDate`

or, add the import statement at the top of the class.

```
import java.time.LocalDate;  
public class DateExample {  
    public static void main (String[] args) {  
        LocalDate myDate;  
    }  
}
```

Working with Dates

`java.time`

- Main package for date and time classes

`java.time.format`

- Contains classes with static methods that you can use to format dates and times

Some notable classes:

- `java.time.LocalDate`
- `java.time.LocalDateTime`
- `java.time.LocalTime`
- `java.time.format.DateTimeFormatter`

Formatting example:

```
myDate.format(DateTimeFormatter.ISO_LOCAL_DATE);
```

Working with Different Calendars

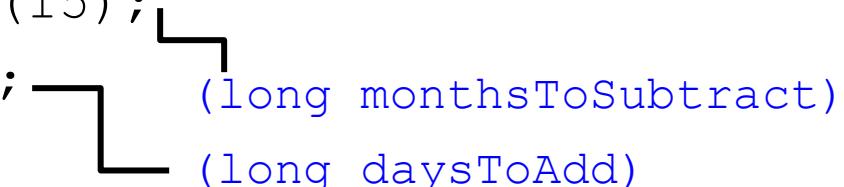
- The default calendar is based on the Gregorian calendar.
- If you need non-Gregorian type dates:
 - Use the `java.time.chrono` classes
 - They have conversion methods.
- Example: Convert a `LocalDate` to a Japanese date:

```
LocalDate myDate = LocalDate.now();  
JapaneseDate jDate = JapaneseDate.from(mydate);  
System.out.println("Japanese date: "+ jDate);
```

- Output:
Japanese date: Japanese Heisei 26-01-16

Some Methods of LocalDate

LocalDate overview: A few notable methods and fields

- Instance methods:
 - myDate.minusMonths (15);
 - myDate.plusDays (8);
 - Static methods:
 - of(int year, Month month, int dayOfMonth)
 - parse(CharSequence text, DateTimeFormatter formatter)
 - now()
- 
- ```
myDate.minusMonths (15); ← long monthsToSubtract
myDate.plusDays (8); ← long daysToAdd
```

# Formatting Dates

```
1 LocalDateTime today = LocalDateTime.now();
2 System.out.println("Today's date time (no formatting): "
3 + today);
4
5
6 String sdate =
7 today.format(DateTimeFormatter.ISO_DATE_TIME);
8 System.out.println("Date in ISO_DATE_TIME format: "
9 + sdate);
10
11 String fdate =
12 today.format(DateTimeFormatter.ofLocalizedDateTime
14 (FormatStyle.MEDIUM));
15 System.out.println("Formatted with MEDIUM FormatStyle: "
16 + fdate);
```

Format the date in standard ISO format.

Localized date time in Medium format

## Output:

|                                    |                         |
|------------------------------------|-------------------------|
| Today's date time (no formatting): | 2013-12-23T16:51:49.458 |
| Date in ISO_DATE_TIME format:      | 2013-12-23T16:51:49.458 |
| Formatted with MEDIUM FormatStyle: | Dec 23, 2013 4:51:49 PM |

# Exercise 11-1: Declare a LocalDateTime Object

1. Declare and initialize a `LocalDateTime` object.
2. Print and format the `OrderDate`.
3. Run the code.



# Topics

- Working with dates
- Parsing the args array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- The ArrayList class

# Using the args Array in the main Method

- Parameters can be typed on the command line:

```
> java ArgsTest Hello World!
args[0] is Hello
args[1] is World!
```

*Goes into args[1]*

*Goes into args[0]*

- Code for retrieving the parameters:

```
public class ArgsTest {
 public static void main (String[] args) {
 System.out.println("args[0] is " + args[0]);
 System.out.println("args[1] is " + args[1]);
 }
}
```

# Converting String Arguments to Other Types

- Numbers can be typed as parameters:

```
> java ArgsTest 2 3
```

Total is: 23

Total is: 5

Concatenation, not addition!

- Conversion of String to int:

```
public class ArgsTest {
 public static void main (String[] args) {
 System.out.println("Total is:"+ (args[0]+args[1]));
 int arg1 = Integer.parseInt(args[0]);
 int arg2 = Integer.parseInt(args[1]);
 System.out.println("Total is: " + (arg1+arg2));
 }
}
```

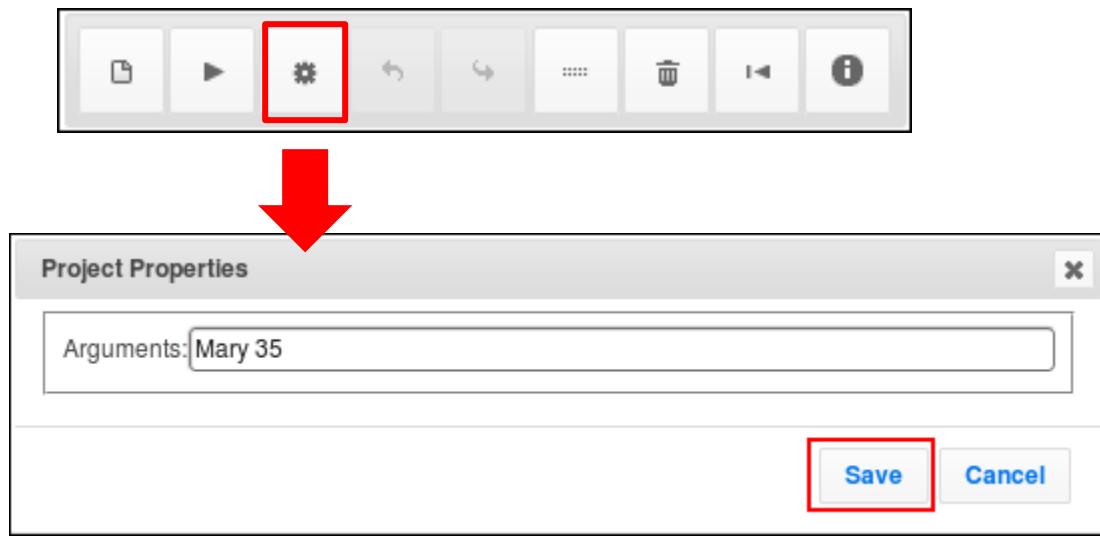
Strings

Note the parentheses.

## Exercise 11-2: Parsing the args Array

In this exercise, you parse the `args` array in the `main` method to get the command-line arguments and assign them to local variables.

- To enter command-line arguments, click the Configure button and enter the values, separated by a space.



# Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- The `ArrayList` class

# Describing Two-Dimensional Arrays

|        | Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
|--------|--------|--------|---------|-----------|----------|--------|----------|
| Week 1 |        |        |         |           |          |        |          |
| Week 2 |        |        |         |           |          |        |          |
| Week 3 |        |        |         |           |          |        |          |
| Week 4 |        |        |         |           |          |        |          |

# Declaring a Two-Dimensional Array

Example:

```
int [][] yearlySales;
```

Syntax:

```
type [][] array_identifier;
```

# Instantiating a Two-Dimensional Array

Example:

```
// Instantiates a 2D array: 5 arrays of 4 elements each
yearlySales = new int[5] [4];
```

Syntax:

```
array_identifier = new type [number_of_arrays] [length];
```

|        | Quarter 1 | Quarter 2 | Quarter 3 | Quarter 4 |
|--------|-----------|-----------|-----------|-----------|
| Year 1 |           |           |           |           |
| Year 2 |           |           |           |           |
| Year 3 |           |           |           |           |
| Year 4 |           |           |           |           |
| Year 5 |           |           |           |           |

# Initializing a Two-Dimensional Array

Example:

```
int[][] yearlySales = new int[5][4];
yearlySales[0][0] = 1000;
yearlySales[0][1] = 1500;
yearlySales[0][2] = 1800;
yearlySales[1][0] = 1000;
yearlySales[3][3] = 2000;
```

|        | Quarter 1 | Quarter 2 | Quarter 3 | Quarter 4 |
|--------|-----------|-----------|-----------|-----------|
| Year 1 | 1000      | 1500      | 1800      |           |
| Year 2 | 1000      |           |           |           |
| Year 3 |           |           |           |           |
| Year 4 |           |           |           | 2000      |
| Year 5 |           |           |           |           |

# Quiz

A two-dimensional array is similar to a \_\_\_\_\_.

- a. Shopping list
- b. List of chores
- c. Matrix
- d. Bar chart containing the dimensions for several boxes



# Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- The `ArrayList` class

# Some New Types of Loops

Loops are frequently used in programs to repeat blocks of code while some condition is true.

There are three main types of loops:

- A `while` loop repeats *while* an expression is true.
- A `for` loop simply repeats a *set number* of times.
  - \* A variation of this is the **enhanced** `for` loop. This loops through the elements of an array.
- A `do/while` loop executes once and then continues to repeat *while* an expression is true.

\**You have already learned this one!*

# Repeating Behavior



```
while (!areWeThereYet) {

 read book;
 argue with sibling;
 ask, "Are we there yet?";

}

Woohoo!;
Get out of car;
```

# A while Loop Example

```
01 public class Elevator {
02 public int currentFloor = 1;
03
04 public void changeFloor(int targetFloor){
05 while (currentFloor != targetFloor) {
06 if(currentFloor < targetFloor)
07 goUp();
08 else
09 goDown();
10 }
11 }

 Boolean expression
 Body of the loop
```

# Coding a while Loop

Syntax:

```
while (boolean_expression) {
 code_block;
}
```

# while Loop with Counter

```
01 System.out.println("/*");
02 int counter = 0;
03 while (counter < 3) {
04 System.out.println(" *");
05 counter++;
06 }
07 System.out.println("*/");
```

## Output:

```
/*
 *
 *
 *
 */

```

# Coding a Standard for Loop

The standard `for` loop repeats its code block for a set number of times using a counter.

Example:

```
01 for(int i = 1; i < 5; i++) {
02 System.out.print("i = " + i + "; ");
03 }
```

Output: i = 1; i = 2; i = 3; i = 4;

Syntax:

```
01 for (<type> counter = n;
02 <boolean_expression>;
03 <counter_increment>) {
04 code_block;
05 }
```

# Standard `for` Loop Compared to a `while` loop

`while` loop

```
01 int i = 0;
02 while (i < 3) {
03 System.out.println(" *");
04 i++;
05 }
```

Initialize  
counter

Increment  
counter

`for` loop

```
01 for (int num = 0; num < 3; num++) {
02 System.out.println(" *");
03 }
```

# Standard for Loop Compared to an Enhanced for Loop

Enhanced for loop

```
01 for(String name: names) {
02 System.out.println(name);
03 }
```

Standard for loop

```
01 for (int idx = 0; idx < names.length; idx++) {
02 System.out.println(names[idx]);
03 }
```

boolean expression

Counter used as the index of the array

# do/while Loop to Find the Factorial Value of a Number

```
1 // Finds the product of a number and all integers below it
2 static void factorial(int target) {
3 int save = target;
4 int fact = 1;
5 do {
6 fact *= target--;
7 }while(target > 0);
8 System.out.println("Factorial for "+save+": "+ fact);
9 }
```

Executed once before evaluating the condition

Outputs for two different targets:

Factorial value for 5: 120

Factorial value for 6: 720

# Coding a do/while Loop

Syntax:

```
do {
 code_block; } └─ This block executes at least once.
 }
 while (boolean_expression); // Semicolon is mandatory.
```

# Comparing Loop Constructs

- Use the `while` loop to iterate indefinitely through statements and to perform the statements zero or more times.
- Use the standard `for` loop to step through statements a predefined number of times.
- Use the enhanced `for` loop to iterate through the elements of an `Array` or `ArrayList` (discussed later).
- Use the `do/while` loop to iterate indefinitely through statements and to perform the statements *one* or more times.

# The continue Keyword

There are two keywords that enable you to interrupt the iterations in a loop of any type:

- break causes the loop to exit. \*
- continue causes the loop to skip the current iteration and go to the next.

```
01 for (int idx = 0; idx < names.length; idx++) {
02 if (names[idx].equalsIgnoreCase("Unavailable"))
03 continue;
04 System.out.println(names[idx]);
05 }
```

\* Or any block of code to exit

# Exercise 11-3: Processing an Array of Items

In this exercise you:

- Process an array of items to calculate the Shopping Cart total
- Skip any items that are back ordered
- Display the total

|           |                   |
|-----------|-------------------|
| Item.java | ShoppingCart.java |
|-----------|-------------------|

```
1 package ex11_2_exercise;
2
3 public class ShoppingCart {
4 Item[] items = {new Item("Shirt",25.60),
5 new Item("WristBand",1.00),
6 new Item("Pants",35.99)};
7
8 public static void main(String[] args){
9 ShoppingCart cart = new ShoppingCart();
10 cart.displayTotal();
11 }
12
13 // Use a standard for loop to iterate through the items array, adding up the total price
14 // Skip any items that are back ordered. Display the Shopping Cart total.
15 public void displayTotal(){
16
17 }
18
19 }
20
```



# Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- **Nesting loops**
- The `ArrayList` class

# Nesting Loops

All types of loops can be nested within the body of another loop. This is a powerful construct used to:

- Process multidimensional arrays
- Sort or manipulate large amounts of data



How it works:

1<sup>st</sup> iteration of outer loop triggers:

    Inner loop

2<sup>nd</sup> iteration of outer loop triggers:

    Inner loop

3<sup>rd</sup> iteration of outer loop triggers:

    Inner loop

and so on...

# Nested for Loop

Example: Print a table with 4 rows and 10 columns:

```
01 int height = 4, width = 10;
02
03 for(int row = 0; row < height; row++) {
04 for (int col = 0; col < width; col++) {
05 System.out.print("@");
06 }
07 System.out.println();
08 }
```

Output:

```
run:
@ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @
@ @ @ @ @ @ @ @ @ @
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Nested while Loop

Example:

```
01 String name = "Lenny";
02 String guess = "";
03 int attempts = 0;
04 while (!guess.equalsIgnoreCase(name)) {
05 guess = "";
06 while (guess.length() < name.length()) {
07 char asciiChar = (char) (Math.random() * 26 + 97);
08 guess += asciiChar;
09 }
10 attempts++;
11 }
12 System.out.println(name+" found after "+attempts+" tries!");
```

Output:

Lenny found after 20852023 tries!

# Processing a Two-Dimensional Array

## Example: Quarterly Sales per Year

```
01 int sales[][] = new int[3][4];
02 initArray(sales); //initialize the array
03 System.out.println
04 ("Yearly sales by quarter beginning 2010:");
05 for(int i=0; i < sales.length; i++) {
06 for(int j=0; j < sales[i].length; j++) {
07 System.out.println("\tQ"+(j+1)+" "+sales[i][j]);
08 }
09 System.out.println();
10 }
```

# Output from Previous Example

```
Yearly sales by quarter beginning 2010:
```

```
Q1 36631
```

```
Q2 62699
```

```
Q3 60795
```

```
Q4 11975
```

```
Q1 72535
```

```
Q2 37363
```

```
Q3 20527
```

```
Q4 36670
```

```
Q1 3195
```

```
Q2 98608
```

```
Q3 21433
```

```
Q4 98519
```

# Quiz

---

\_\_\_\_\_ enable you to check and recheck a decision to execute and re-execute a block of code.

- a. Classes
- b. Objects
- c. Loops
- d. Methods

# Quiz

Which of the following loops always executes at least once?

- a. The while loop
- b. The nested while loop
- c. The do/while loop
- d. The for loop

# Topics

- Working with dates
- Parsing the `args` array
- Two-dimensional arrays
- Alternate looping constructs
- Nesting loops
- **The ArrayList class**

# ArrayList Class

Arrays are not the only way to store lists of related data.

- ArrayList is one of several list management classes.
- It has a set of useful methods for managing its elements:
  - add, get, remove, indexOf, and many others
- It can store *only objects*, not primitives.
  - Example: an ArrayList of Shirt objects:
    - shirts.add(shirt04);
  - Example: an ArrayList of String objects:
    - names.remove ("James");
  - Example: an ArrayList of ages:
    - ages.add(5) //NOT ALLOWED!
    - ages.add(new Integer(5)) // OK

# Benefits of the ArrayList Class

- Dynamically resizes:
  - An ArrayList grows as you add elements.
  - An ArrayList shrinks as you remove elements.
  - You can specify an initial capacity, but it is not mandatory.
- Option to designate the object type it contains:

```
ArrayList<String> states = new ArrayList();
```

Contains only String objects

- Call methods on an ArrayList or its elements:

```
states.size(); //Size of list
```

```
states.get(49).length(); //Length of 49th element
```

# Importing and Declaring an ArrayList

- You must `import java.util.ArrayList` to use an `ArrayList`.
- An `ArrayList` may contain any object type, including a type that you have created by writing a class.

```
import java.util.ArrayList;
```

```
public class ArrayListExample {
 public static void main (String[] args) {
 ArrayList<Shirt> myList;
 }
}
```

You may specify any object type.

# Working with an ArrayList

```
01 ArrayList<String> names; _____ Declare an ArrayList of
02 names = new ArrayList(); _____ Strings.
03
04 names.add("Jamie");
05 names.add("Gustav");
06 names.add("Alisa");
07 names.add("Jose");
08 names.add(2, "Prashant"); _____ Initialize it.
09
10 names.remove(0);
11 names.remove(names.size() - 1);
12 names.remove("Gustav"); _____ Modify it.
13
14 System.out.println(names);
```

# Exercise 11-4: Working with an ArrayList

In this exercise, you:

- Declare, instantiate, and initialize an `ArrayList` of `Strings`
- Use two different `add` methods
  - `add(E element)`
  - `add(int index, E element)`
- Use the following methods to find and remove a specific element if it exists
  - `contains (Object element)`
  - `remove (Object element)`



# Summary

In this lesson, you should have learned how to:

- Create a `java.time.LocalDateTime` object to show the current date and time
- Parse the `args` array of the `main` method
- Nest a `while` loop
- Develop and nest a `for` loop
- Code and nest a `do/while` loop
- Use an `ArrayList` to store and manipulate objects



# Play Time!

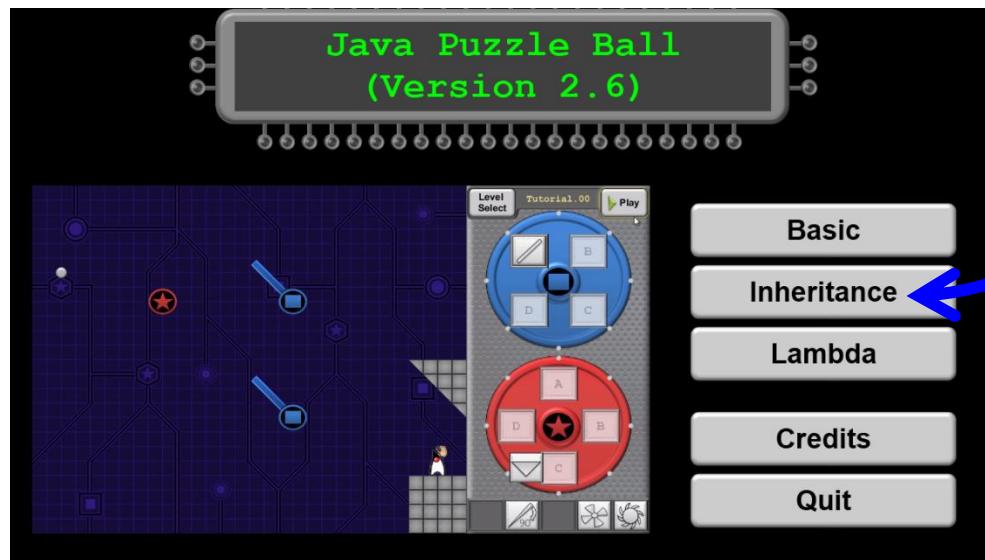


Play **Inheritance Puzzles 1, 2, and 3** before the next lesson titled “Using Inheritance.”

Consider the following:

What is inheritance?

Why are these considered “Inheritance” puzzles?

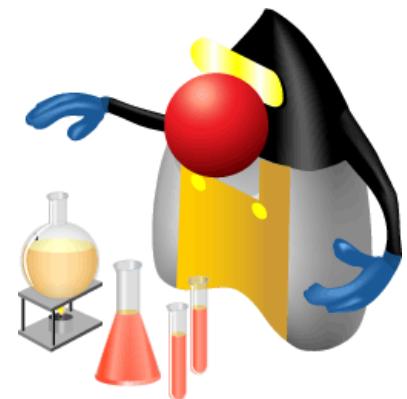


Play this  
game-mode.

# Practice 11-1 Overview: Iterating Through Data

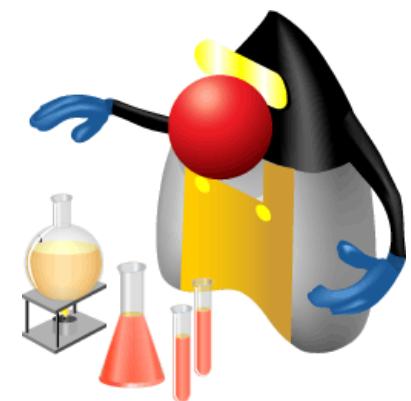
This practice covers the following topics:

- Converting a comma-separated list of names to an array of names
- Processing the array using a `for` loop
- Using a nested loop to populate an `ArrayList` of games



## **Practice 11-2 Overview: Working with LocalDateTime**

This practice covers working with a few classes and methods from the `java.time` package in order to show date information for games played.



# 12

## Using Inheritance

# Objectives

After completing this lesson, you should be able to:

- Define inheritance in the context of a Java class hierarchy
- Create a subclass
- Override a method in the superclass
- Use a keyword to reference the superclass
- Define polymorphism
- Use the `instanceof` operator to test an object's type
- Cast a superclass reference to the subclass type
- Explain the difference between abstract and nonabstract classes
- Create a class hierarchy by extending an abstract class

# Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

# Java Puzzle Ball

Have you played through **Inheritance Puzzle 3?**

Consider the following:

What is inheritance?

Why are these considered “Inheritance” puzzles?



# Java Puzzle Ball Debrief

What is inheritance?

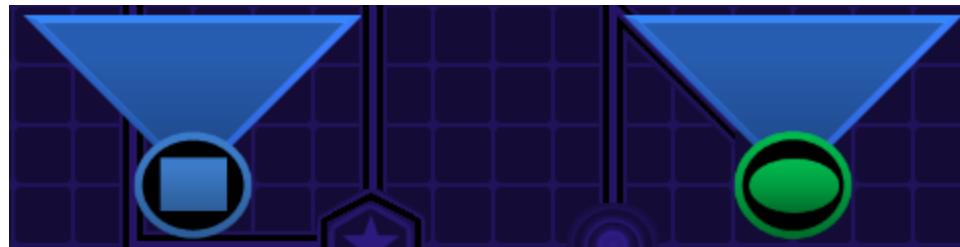
- **Inheritance** allows one class to be derived from another.
  - A child inherits properties and behaviors of the parent.
  - A child *class* inherits the fields and method of a parent *class*.
- In the game:
  - Blue shapes also appear on *green* bumpers



# Inheritance in Java Puzzle Ball

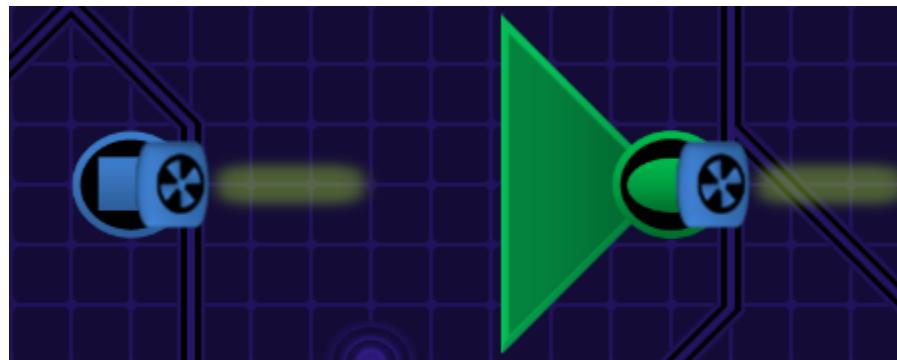
## Inheritance Puzzle 1:

- Methods for deflecting the ball that were originally assigned to Blue Bumpers are also found on Green Bumpers.



## Inheritance Puzzle 2:

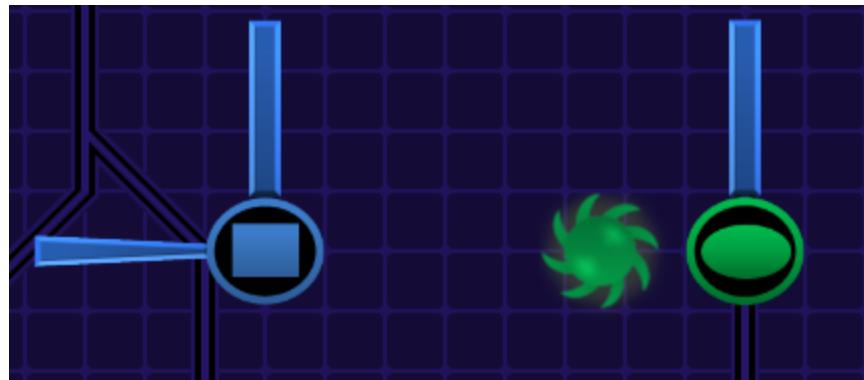
- Green Bumpers contain methods from Blue Bumpers, PLUS methods unique to Green Bumpers.



# Inheritance in Java Puzzle Ball

## Inheritance Puzzle 3:

- If Green Bumpers inherit unwanted Blue Bumper methods, it is possible to **override**, or replace those methods.



# Implementing Inheritance

```
public class Clothing {
 public void display() {...}
 public void setSize(char size) {...}
}
```

```
public class Shirt extends Clothing { ... }
```



Use the **extends** keyword.

```
Shirt myShirt = new Shirt();
myShirt.setSize ('M');
```

This code works!

# More Inheritance Facts

- The parent class is the **superclass**.
- The child class is the **subclass**.
- A subclass may have unique fields and methods not found in the superclass.

The diagram illustrates the inheritance relationship between the `Shirt` class and the `Clothing` class. The `Shirt` class is labeled as a **sub class** and the `Clothing` class is labeled as a **super class**. A blue bracket underlines the word `extends` in the code, indicating the mechanism by which `Shirt` inherits from `Clothing`.

```
public class Shirt extends Clothing {
 private int neckSize;
 public int getNeckSize() {
 return neckSize;
 }
 public void setNeckSize(int nSize) {
 this.neckSize = nSize;
 }
}
```

# Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

# Duke's Choice Classes: Common Behaviors

| Shirt                                                                                                                         | Trousers                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>getId()</code><br><code>getPrice()</code><br><code>getSize()</code><br><code>getColor()</code><br><code>getFit()</code> | <code>getId()</code><br><code>getPrice()</code><br><code>getSize()</code><br><code>getColor()</code><br><code>getFit()</code><br><code>getGender()</code> |
| <code>setId()</code><br><code>setPrice()</code><br><code>setSize()</code><br><code>setColor()</code><br><code>setFit()</code> | <code>setId()</code><br><code>setPrice()</code><br><code>setSize()</code><br><code>setColor()</code><br><code>setFit()</code><br><code>setGender()</code> |
| <code>display()</code>                                                                                                        | <code>display()</code>                                                                                                                                    |

# Code Duplication

**Shirt**

```
getId()
display()
getPrice()
getSize()
getColor()
getFit()
```

**Trousers**

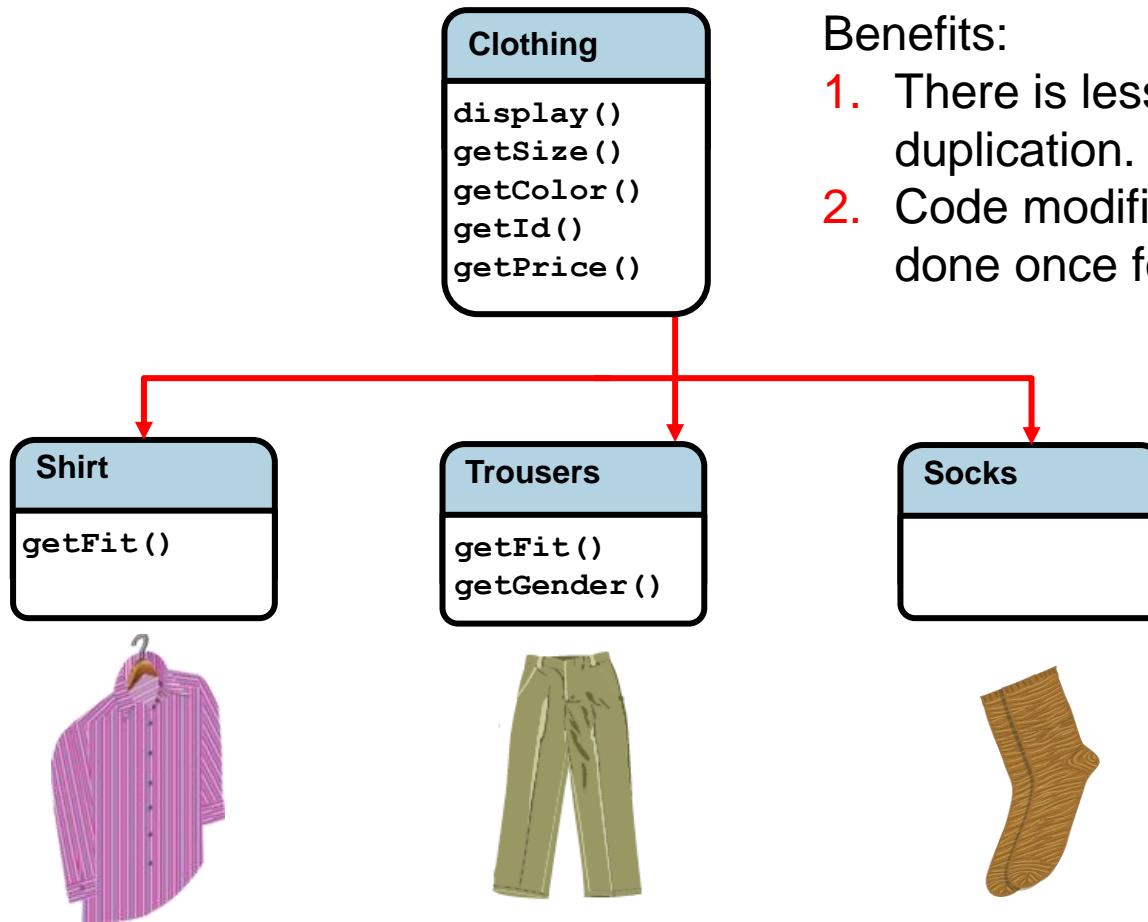
```
getId()
display()
getPrice()
getSize()
getColor()
getFit()
getGender()
```

**Socks**

```
getId()
display()
getPrice()
getSize()
getColor()
```



# Inheritance



Benefits:

1. There is less code duplication.
2. Code modification can be done once for all subclasses.

# Clothing Class: Part 1

```
01 public class Clothing {
02 // fields given default values
03 private int itemID = 0;
04 private String desc = "-description required-";
05 private char colorCode = 'U';
06 private double price = 0.0;
07
08 // Constructor
09 public Clothing(int itemID, String desc, char color,
10 double price) {
11 this.itemID = itemID;
12 this.desc = desc;
13 this.colorCode = color;
14 this.price = price;
15 }
16 }
```

# Shirt Class: Part 1

```
01 public class Shirt extends Clothing {
03 private char fit = 'U';
04
05 public Shirt(int itemID, String description, char
06 colorCode, double price, char fit) {
07 super(itemID, description, colorCode, price);
08
09 this.fit = fit; Reference to the
superclass constructor
10 }
12 public char getFit() { Reference to
this object
13 return fit;
14 }
15 public void setFit(char fit) {
16 this.fit = fit;
17 }
```

# Constructor Calls with Inheritance

```
public static void main(String[] args) {
 Shirt shirt01 = new Shirt(20.00, 'M'); }
```

```
public class Shirt extends Clothing {
 private char fit = 'U';

 public Shirt(double price, char fit) {
 super(price); //MUST call superclass constructor
 this.fit = fit; } }
```



```
public class Clothing{
 private double price;

 public Clothing(double price){
 this.price = price;
 } }
```

# Inheritance and Overloaded Constructors

```
public class Shirt extends Clothing {
 private char fit = 'U';

 public Shirt(char fit){
 this(15.00, fit); //Call constructor in same class
 } //Constructor is overloaded

 public Shirt(double price, char fit) {
 super(price); //MUST call superclass constructor
 this.fit = fit;
 } }
```



```
public class Clothing{
 private double price;

 public Clothing(double price){
 this.price = price;
 } }
```

# Exercise 12-1: Creating a Subclass

In this exercise, you create the `Shirt` class, which extends the `Item` class.

- Add two fields that are unique to the `Shirt` class.
- Invoke the superclass constructor from the `Shirt` constructor.
- Instantiate a `Shirt` object and call the `display` method.

|                                                       |   |
|-------------------------------------------------------|---|
| Output                                                | ✖ |
| <pre>Item description: Shirt ID: 1 Price: 25.99</pre> |   |



# Topics

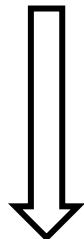
- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

# More on Access Control

Access level modifiers determine whether other classes can use a particular field or invoke a particular method

- At the top level—public, or *package-private* (no explicit modifier).
- At the member level—public, private, protected, or *package-private* (no explicit modifier).

Stronger  
access  
privileges



| Modifier    | Class | Package | Subclass | World |
|-------------|-------|---------|----------|-------|
| public      | Y     | Y       | Y        | Y     |
| protected   | Y     | Y       | Y        | N     |
| No modifier | Y     | Y       | N        | N     |
| private     | Y     | N       | N        | N     |

# Overriding Methods

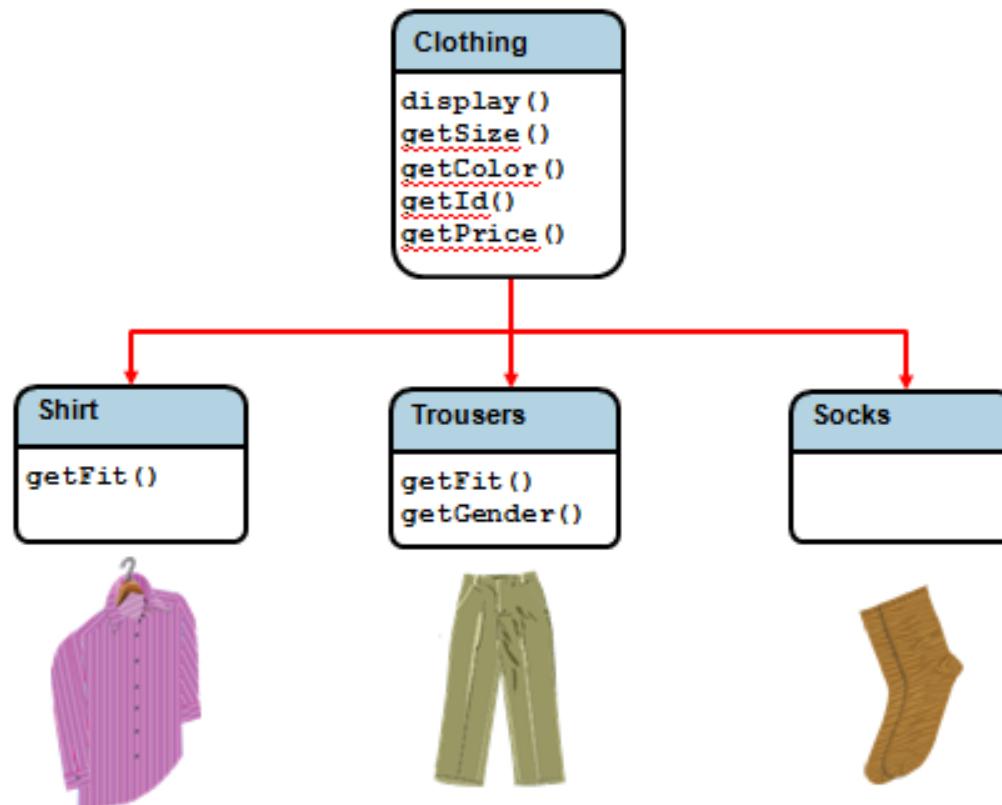
**Overriding:** A subclass implements a method that already has an implementation in the superclass.

## Access Modifiers:

- The method can only be overridden if it is accessible from the subclass
- The method signature in the subclass cannot have a more restrictive (stronger) access modifier than the one in the superclass

# Review: Duke's Choice Class Hierarchy

Now consider these classes in more detail.



# Clothing Class: Part 2

```
29 public void display() {
30 System.out.println("Item ID: " + getItemID());
31 System.out.println("Item description: " + getDesc());
32 System.out.println("Item price: " + getPrice());
33 System.out.println("Color code: " + getColorCode());
34 }
35 public String getDesc (){
36 return desc;
37 }
38 public double getPrice() {
39 return price;
40 }
41 public int getItemID() {
42 return itemID;
43 }
44 protected void setColorCode(char color) {
45 this.colorCode = color; }
```

*Assume that the remaining  
get/set methods are  
included in the class.*

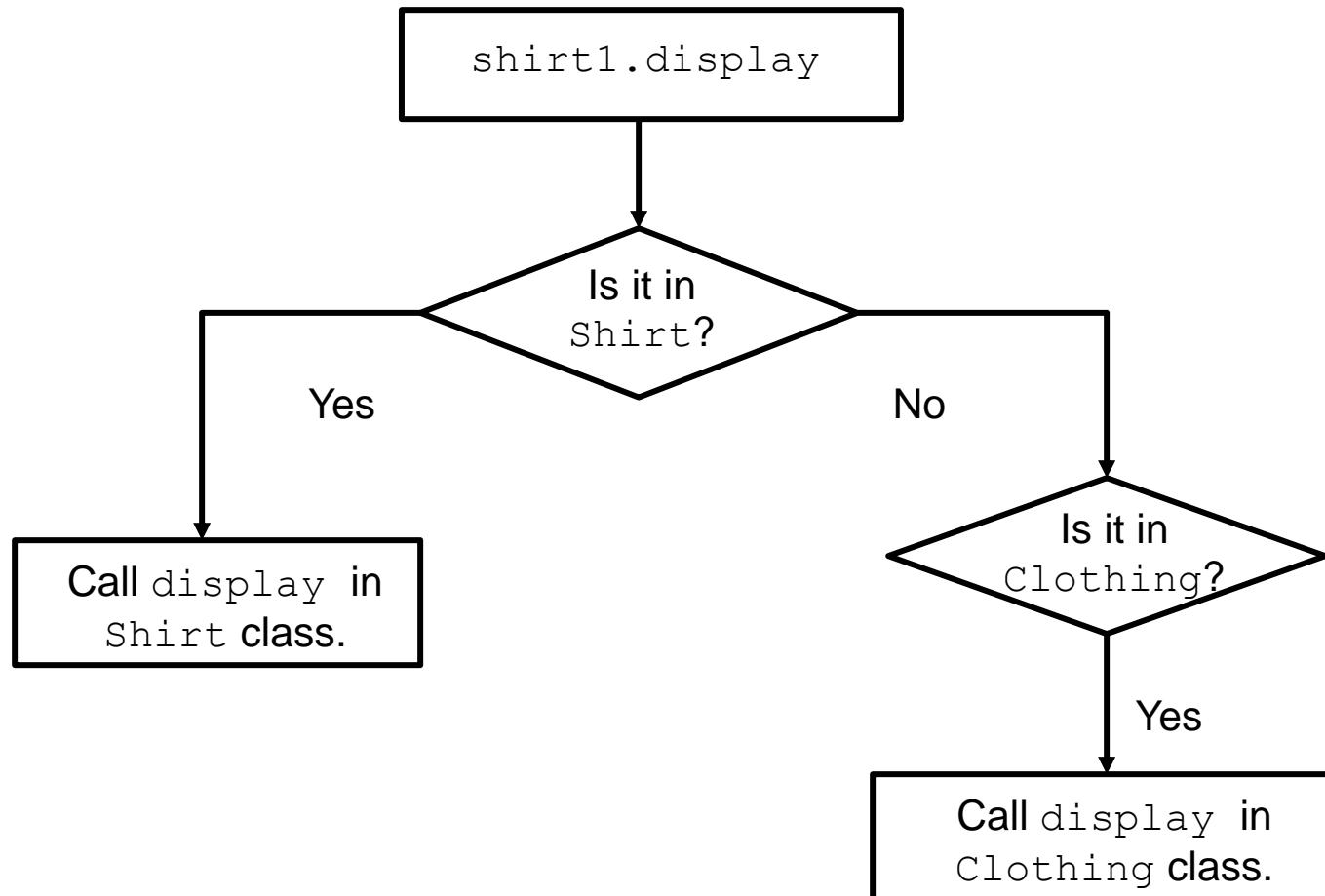
# Shirt Class: Part 2

```
17 // These methods override the methods in Clothing
18 public void display() {
19 System.out.println("Shirt ID: " + getItemID());
20 System.out.println("Shirt description: " + getDesc());
21 System.out.println("Shirt price: " + getPrice());
22 System.out.println("Color code: " + getColorCode());
23 System.out.println("Fit: " + getFit());
24 }
25
26 protected void setColorCode(char colorCode) {
27 //Code here to check that correct codes used
28 super.setColorCode(colorCode);
29 }
30}
```



Call the superclass's version of setColorCode.

# Overriding a Method: What Happens at Run Time?



## Exercise 12-2: Overriding a Method in the Superclass

In this exercise, you override a method in the Item class.

- Override the `display` method to show the additional fields from the `Shirt` class.
- Run the `ShoppingCart` to see the result.

```
Output ✖
Item description: Shirt
ID: 1
Price: 25.99
Size: M
Color Code: P
```



# Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

# Polymorphism

Polymorphism means that the same message to two different objects can have different results.

- “Good night” to a child means “Start getting ready for bed.”
- “Good night” to a parent means “Read a bedtime story.”

In Java, it means the same method is implemented differently by different classes.

- This is especially powerful in the context of inheritance.
- It relies upon the “*is a*” relationship.



# Superclass and Subclass Relationships

Use inheritance only when it is completely valid or unavoidable.

- Use the “*is a*” test to decide whether an inheritance relationship makes sense.
- Which of the phrases below expresses a valid inheritance relationship within the Duke’s Choice hierarchy?



A Shirt *is a* piece of Clothing.

- A Hat *is a* Sock.
- Equipment *is a* piece of Clothing.



Clothing and Equipment *are* Items.

# Using the Superclass as a Reference

So far, you have referenced objects only with a reference variable of the same class:

- To use the `Shirt` class as the reference type for the `Shirt` object:

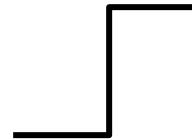
```
Shirt myShirt = new Shirt();
```

- But you can also use the superclass as the reference:

```
Clothing garment1 = new Shirt();
```

```
Clothing garment2 = new Trousers();
```

Shirt is a (type of) Clothing.  
Trousers is a (type of) Clothing.



# Polymorphism Applied

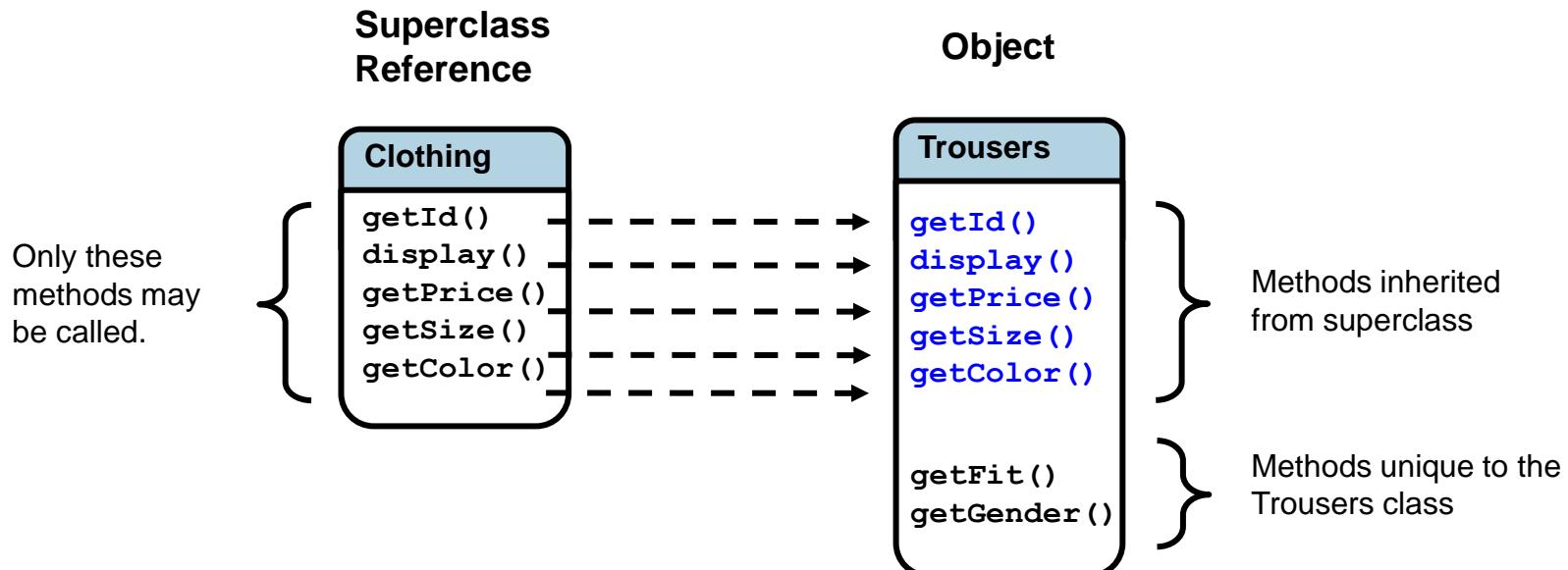
```
Clothing c1 = new ??();
c1.display();
c1.setColorCode('P');
```

c1 could be a Shirt,  
Trousers, or Socks object.

The method will be implemented differently on different types of objects. For example:

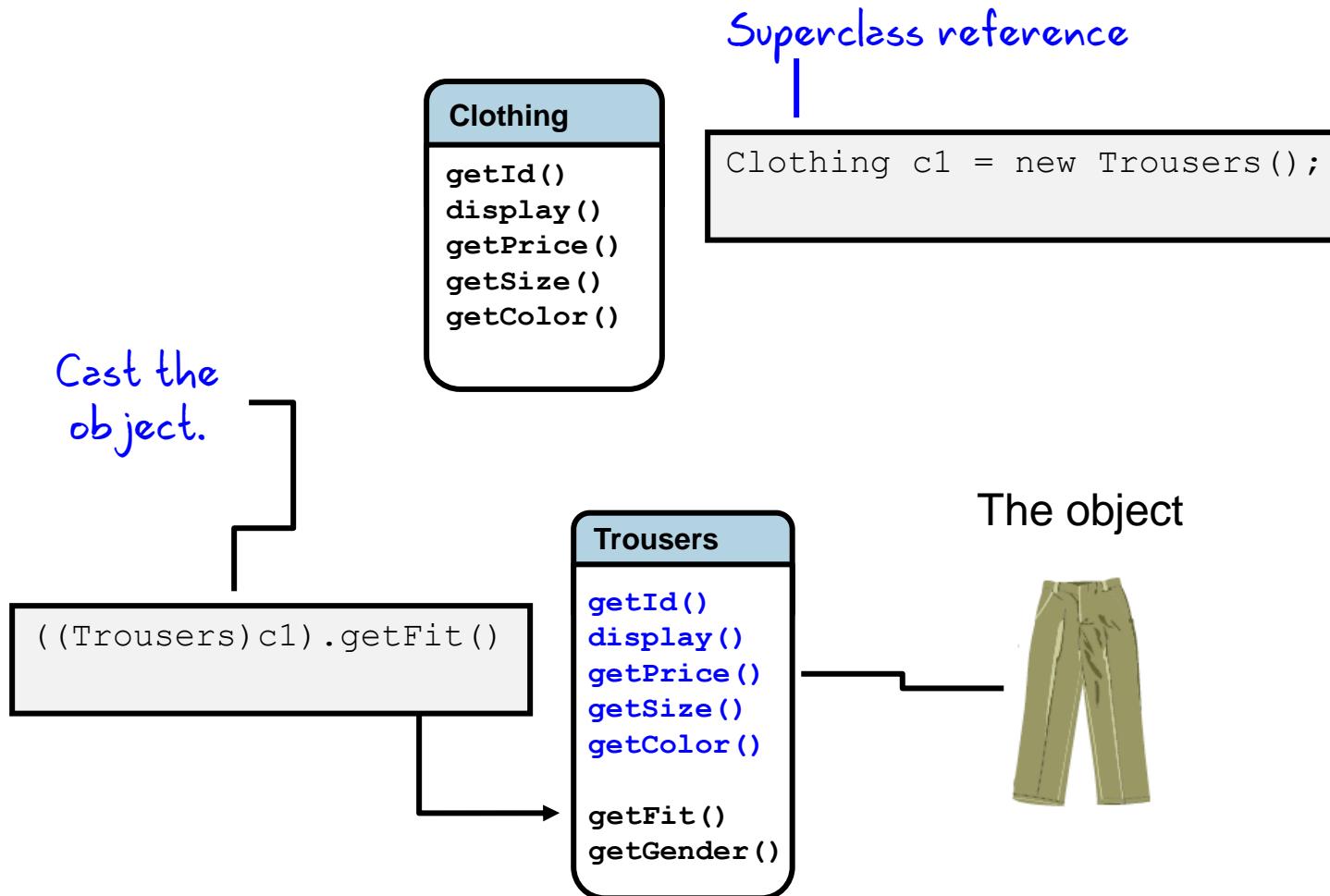
- Trousers objects show more fields in the display method.
- Different subclasses accept a different subset of valid color codes.

# Accessing Methods Using a Superclass Reference



```
Clothing c1 = new Trouzers();
c1.getId(); OK
c1.display(); OK
c1.getFit(); NO!
```

# Casting the Reference Type



# instanceof Operator

Possible casting error:

```
public static void displayDetails(Clothing cl) {
 cl.display();
 char fitCode = ((Trousers)cl).getFit();
 System.out.println("Fit: " + fitCode);
}
```

What if cl is not a  
Trousers object?

instanceof operator used to ensure there is no casting error:

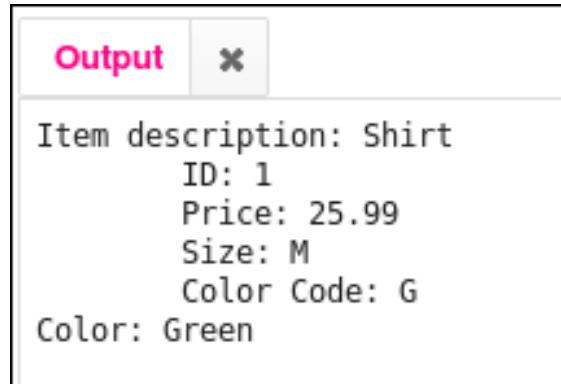
```
public static void displayDetails(Clothing cl) {
 cl.display();
 if (cl instanceof Trousers) {
 char fitCode = ((Trousers)cl).getFit();
 System.out.println("Fit: " + fitCode);
 }
 else { // Take some other action }
}
```

instanceof returns true  
if cl is a Trousers object.

## Exercise 12-3: Using the instanceof Operator

In this exercise, you use the `instanceof` operator to test the type of an object before casting it to that type.

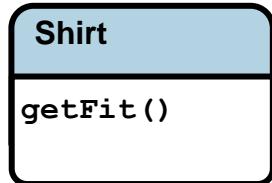
- Add a `getColor` method to the `Shirt` class.
- Instantiate a `Shirt` object using an `Item` reference type.
- Call the `display` method on the object.
- Cast the `Item` reference as a `Shirt` and call `getColor`.



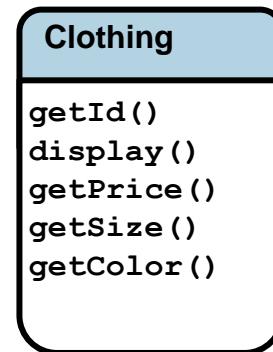
# Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Overriding superclass methods
- Introducing polymorphism
- Creating and extending abstract classes

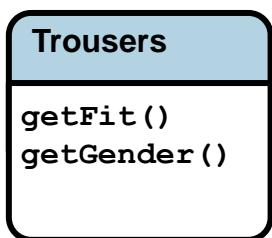
# Abstract Classes



=



=



=



=



# Abstract Classes

Use the `abstract` keyword to create a special class that:

- Cannot be instantiated  `Clothing cloth01 = new Clothing()`
- May contain concrete methods
- May contain abstract methods that **must** be implemented later by any nonabstract subclasses

```
public abstract class Clothing{
 private int id;
```

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

Concrete  
method

```
 }
 public abstract double getPrice();
 public abstract void display();
```

Abstract  
methods

# Extending Abstract Classes

```
public abstract class Clothing{
 private int id;

 public int getId(){
 return id;
 }
 protected abstract double getPrice(); //MUST be implemented
 public abstract void display(); } //MUST be implemented
```

```
public class Socks extends Clothing{
 private double price;

 protected double getPrice(){
 return price;
 }
 public void display(){
 System.out.println("ID: " +getID());
 System.out.println("Price: $" +getPrice());
 } }
```

# Summary

In this lesson, you should have learned the following:

- Creating class hierarchies with subclasses and superclasses helps to create extensible and maintainable code by:
  - Generalizing and abstracting code that may otherwise be duplicated
  - Allowing you to override the methods in the superclass
  - Allowing you to use less-specific reference types
- An abstract class cannot be instantiated, but it can be used to impose a particular interface on its descendants.



# Challenge Questions: Java Puzzle Ball



Bumpers share many of the same properties and behaviors. Is there a way to design code for these objects that minimizes duplicate code and take advantage of polymorphism?

- Common properties: color, shape, x-position, y-position...
- Common behaviors: flash(), chime(), destroy() ...



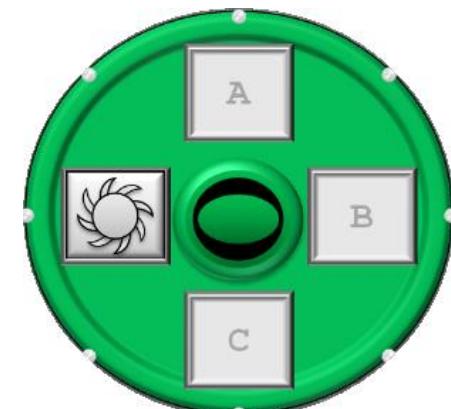
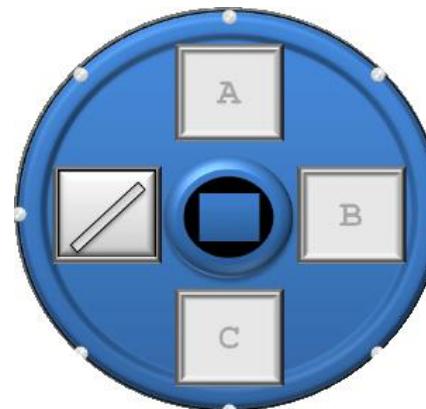
# Challenge Questions: Java Puzzle Ball



For a method to be overridden, a subclass must provide a method with the same name and signature as the superclass's method. Only the implementation may differ.

To make overriding possible, which game components best represent:

- A method name and signature?
- A method implantation?



## **Practice 12-1 Overview: Creating a Class Hierarchy**

This practice covers rewriting the playGame method so that it will eventually be able to work with GameEvent objects and not just Goal objects, which is the case at present.

## **Practice 12-2 Overview: Creating a GameEvent Hierarchy**

This practice covers adding game events to the application. To do this, you create a hierarchy of game event classes that extend an abstract superclass. Some of the game events are:

- Goal
- Pass
- Kickoff

# 13

## Using Interfaces

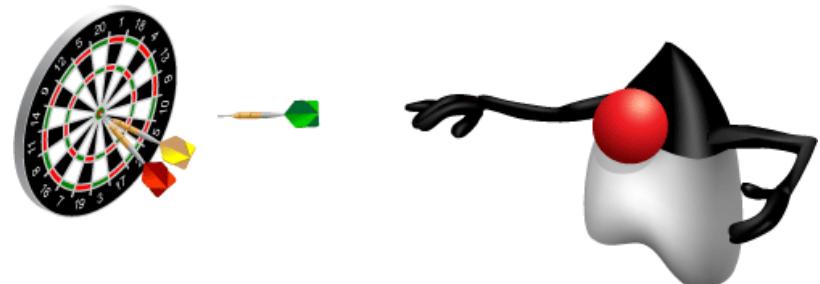
# Interactive Quizzes



# Objectives

After completing this lesson, you should be able to:

- Override the `toString` method of the `Object` class
- Implement an interface in a class
- Cast to an interface reference to allow access to an object method
- Write a simple lambda expression that consumes a Predicate



# Topics

- Polymorphism in the JDK foundation classes
- Using interfaces
- Using the List interface
- Introducing lambda expressions

# The Object Class

compact1, compact2, compact3  
java.util

**Class ArrayList<E>**

java.lang.Object

java.util.AbstractCollection<E>  
java.util.AbstractList<E>  
java.util.ArrayList<E>

The Object class is the base class.

All Implemented Interfaces:

Serializable

Direct Known Subclasses

Attribute

compact1, compact2, compact3  
java.lang

**Class Object**

java.lang.Object

public class **Object**

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

This diagram illustrates the inheritance hierarchy of the `ArrayList` class. It shows that `ArrayList` extends `java.lang.Object`, which is highlighted with a red box. A callout box points to the `Object` class with the text "The Object class is the base class." The `Object` class itself is shown with its direct known subclasses: `compact1, compact2, compact3` and the `java.lang` package. The `Object` class is described as the root of the class hierarchy, implemented by all objects, including arrays. It was introduced in JDK 1.0.

# Calling the `toString` Method

Object's `toString` method is used.

StringBuilder overrides Object's `toString` method.

First inherits Object's `toString` method.

Second overrides Object's `toString` method.

```
1 public class Main {
2 public static void main(String[] args) {
3 // Output an Object to the console
4 System.out.println(new Object());
5
6 // Output this StringBuilder object to the console
7 System.out.println(new StringBuilder("Some text for StringBuilder"));
8
9 //Output a class that does not override the toString() method
10 System.out.println(new First());
11
12 //Output a class that *does* override the toString() method
13 System.out.println(new Second());
14 }
15 }
16 }
```

Output - TestCode (run)

```
run:
java.lang.Object@3e25a5
Some text for StringBuilder
First@19821f
This class named Second has overridden the toString() method of Object
BUILD SUCCESSFUL (total time: 1 second)
```

The output for the calls to the `toString` method of each object

# Overriding `toString` in Your Classes

## Shirt class example

```
1 public String toString() {
2 return "This shirt is a " + desc + ";" +
3 + " price: " + getPrice() + "," +
4 + " color: " + getColor(getColorCode());
5 }
```

Output of `System.out.println(shirt)`:

- **Without overriding `toString`**

`examples.Shirt@73d16e93`

- **After overriding `toString` as shown above**

`This shirt is a T Shirt; price: 29.99, color: Green`

# Topics

- Polymorphism in the JDK foundation classes
- Using interfaces
- Using the List interface
- Introducing lambda expressions

# The Multiple Inheritance Dilemma

Can I inherit from *two* different classes? I want to use methods from both classes.

- Class Red:

```
public void print() {System.out.print("I am Red");}
```

- Class Blue:

```
public void print() {System.out.print("I am Blue");}
```

```
public class Purple extends Red, Blue{
 public void printStuff() {
 print();
 }
}
```

Which  
implementation of  
print() will  
occur?

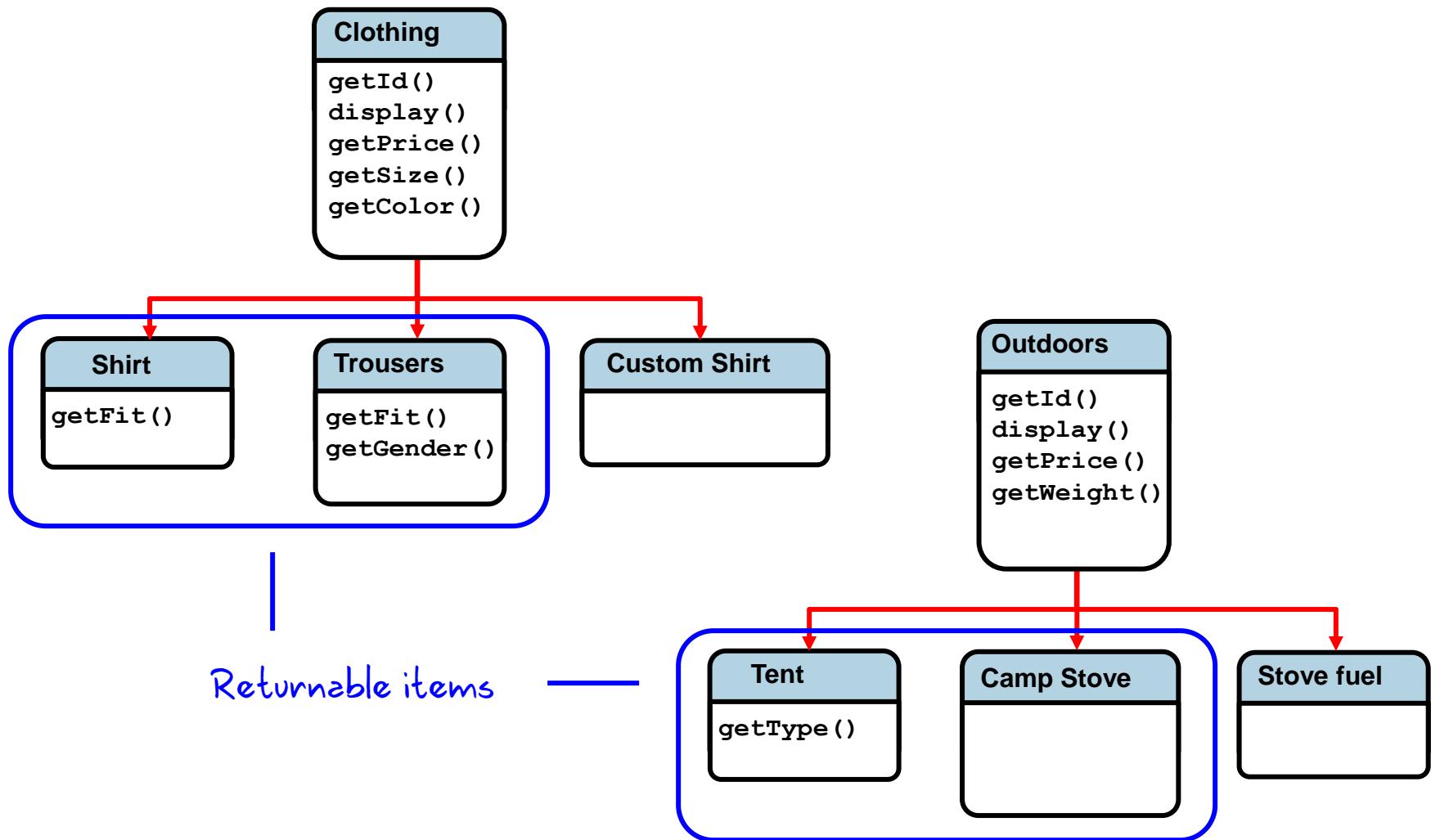
# The Java Interface

- An interface is similar to an abstract class, except that:
  - Methods are implicitly abstract (except default methods)
  - A class does not *extend* it, but *implements* it
  - A class may implement more than one interface
- All abstract methods from the interface must be implemented by the class.

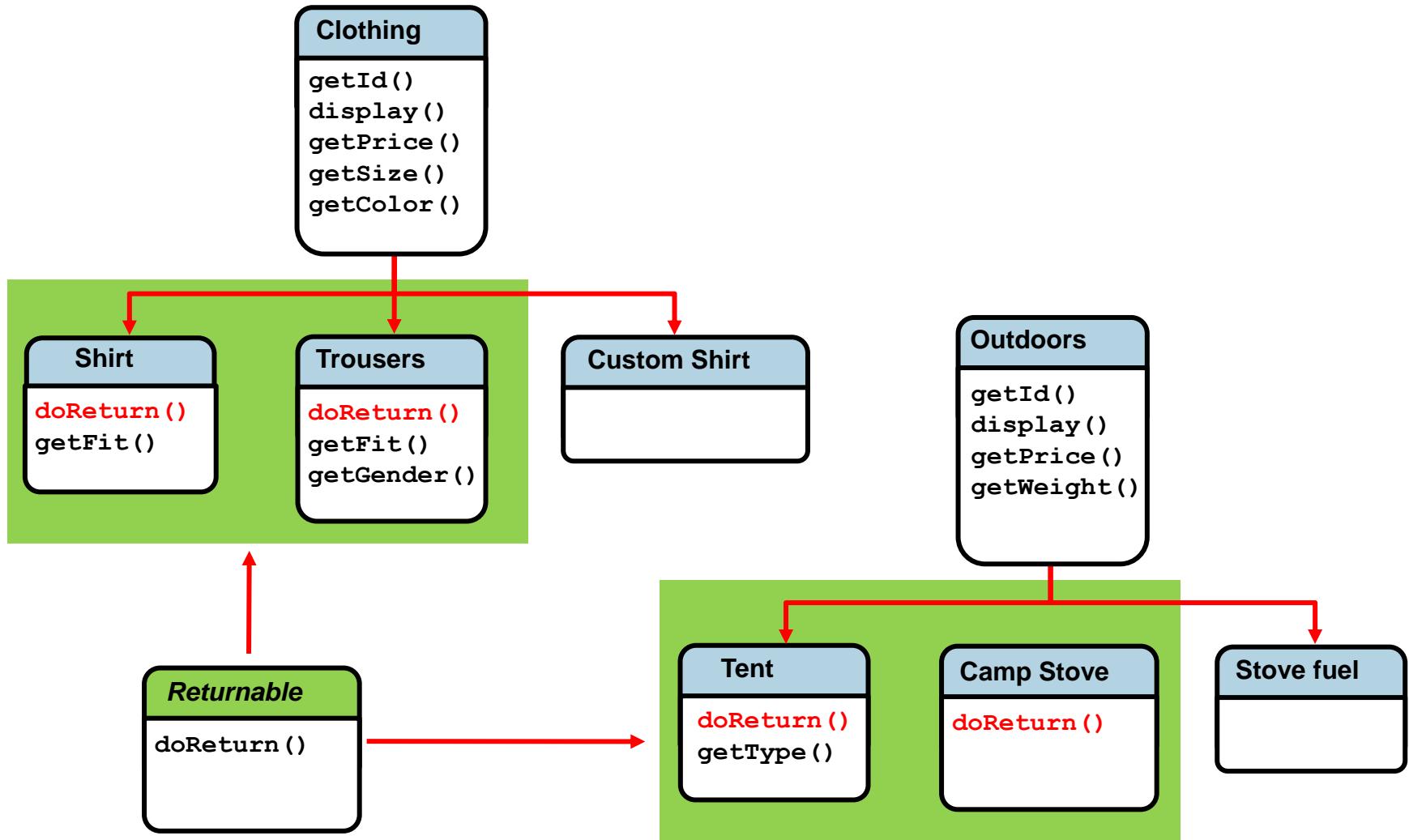
```
1 public interface Printable {
2 public void print(); _____ Implicitly
abstract
3 }
```

```
1 public class Shirt implements Printable { Implements the
print()
method.
2 ...
3 public void print(){
4 System.out.println("Shirt description");
5 }
6 }
```

# Multiple Hierarchies with Overlapping Requirements



# Using Interfaces in Your Application



# Implementing the Returnable Interface

## Returnable interface

```
01 public interface Returnable {
02 public String doReturn(); └─ Implicitly abstract method
03 }
```

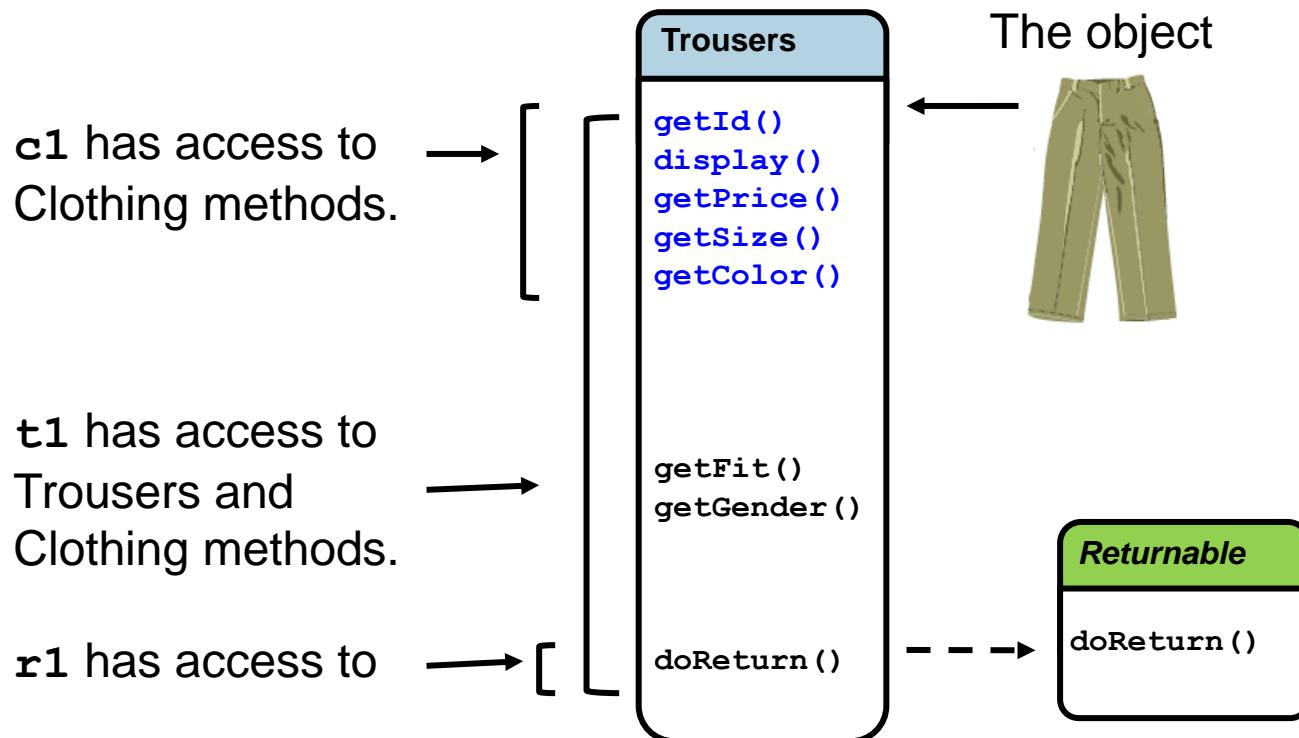
## Shirt class

Now, Shirt 'is a' Returnable.

```
01 public class Shirt extends Clothing implements Returnable {
02 public Shirt(int itemID, String description, char colorCode,
03 double price, char fit) {
04 super(itemID, description, colorCode, price);
05 this.fit = fit;
06 }
07 public String doReturn() { └─ Shirt implements the method
08 // See notes below
09 return "Suit returns must be within 3 days";
10 }
11 ...< other methods not shown > ... } // end of class
```

# Access to Object Methods from Interface

```
Clothing c1 = new Trouzers();
Trousers t1 = new Trouzers();
Returnable r1 = new Trouzers();
```



# Casting an Interface Reference

```
Clothing c1 = new Trousers();
Trousers t1 = new Trousers();
Returnable r1 = new Trousers();
```

- The Returnable interface does not know about Trousers methods:

```
r1.getFit() //Not allowed
```

- Use **casting** to access methods defined outside the interface.

```
((Trousers)r1).getFit();
```

- Use **instanceof** to avoid inappropriate casts.

```
if(r1 instanceof Trousers) {
 ((Trousers)r1).getFit();
}
```

# Quiz

Which methods of an object can be accessed via an interface that it implements?

- a. All the methods implemented in the object's class
- b. All the methods implemented in the object's superclass
- c. The methods declared in the interface

# Quiz

How can you change the reference type of an object?

- a. By calling `getReference`
- b. By casting
- c. By declaring a new reference and assigning the object

# Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- **Using the List interface**
- Introducing lambda expressions

# The Collections Framework

The collections framework is located in the `java.util` package. The framework is helpful when working with lists or collections of objects. It contains:

- Interfaces
- Abstract classes
- Concrete classes (Example: `ArrayList`)

# ArrayList Example

compact1, compact2, compact3

java.util

## Class ArrayList<E>

ArrayList **extends**  
AbstractList, which in turn  
**extends** AbstractCollection.

java.lang.Object

    java.util.AbstractCollection<E>

        java.util.AbstractList<E>

            java.util.ArrayList<E>

### All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

ArrayList  
**implements** a  
number of interfaces.

### Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

---

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

The List interface is  
principally what is used when  
working with ArrayList.

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

# List Interface

```
compact1, compact2, compact3
java.util
```

## Interface List<E>

### Type Parameters:

E - the type of elements in this list

### All Superinterfaces:

Collection<E>, Iterable<E>

### All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

Many classes implement the List interface.

All of these object types can be assigned to a List variable:

```
1 ArrayList<String> words = new ArrayList();
2 List<String> mylist = words;
```

## Example: `Arrays.asList`

The `java.util.Arrays` class has many static utility methods that are helpful in working with arrays.

- Converting an array to a List:

```
1 String[] nums = {"one", "two", "three"};
2 List<String> myList = Arrays.asList(nums);
```

List objects can be of many different types. What if you need to invoke a method belonging to `ArrayList`?

mylist.replaceAll()

This works! `replaceAll` comes from `List`.

mylist.removeIf()

Error! `removeIf` comes from `Collection` (superclass of `ArrayList`).

## Example: `Arrays.asList`

Converting an array to an `ArrayList`:

```
1 String[] nums = {"one", "two", "three"};
2 List<String> myList = Arrays.asList(nums);
3 ArrayList<String> myArrayList = new ArrayList(myList);
```

Shortcut:

```
1 String[] nums = {"one", "two", "three"};
2 ArrayList<String> myArrayList =
 new ArrayList(Arrays.asList(nums));
```

# **Exercise 13-1: Converting an Array to an ArrayList**

In this exercise, you:

- Convert a String array to an ArrayList
- Work with the ArrayList reference to manipulate list values



# Topics

- Polymorphism in the JDK foundation classes
- Using Interfaces
- Using the `List` interface
- Introducing lambda expressions



# Example: Modifying a List of Names

Suppose you want to modify a List of names, changing them all to uppercase. Does this code change the elements of the List?

```
1 String[] names = {"Ned", "Fred", "Jessie", "Alice", "Rick"};
2 List<String> mylist = new ArrayList((Arrays.asList(names)) ;
3
4 // Display all names in upper case
5 for(String s: mylist){
6 System.out.print(s.toUpperCase() + ", "); ┌ Returns a new
7 } ┌ String to print
8 System.out.println("After for loop: " + mylist);
```

Output:

NED, FRED, JESSIE, ALICE, RICK,

After for loop: [Ned, Fred, Jessie, Alice, Rick]

The list  
elements are  
unchanged.

# Using a Lambda Expression with `replaceAll`

`replaceAll` is a default method of the `List` interface. It takes a lambda expression as an argument.

```
mylist.replaceAll(s -> s.toUpperCase());
```

```
System.out.println("List.replaceAll lambda: "+ mylist);
```

Output:

```
List.replaceAll lambda: [NED, FRED, JESSIE, ALICE, RICK]
```

# Lambda Expressions

Lambda expressions are like methods used as the argument for another method. They have:

- Input parameters
- A method body
- A return value

Long version:

```
mylist.replaceAll((String s) -> {return s.toUpperCase();});
```

Declare input  
parameter

Arrow  
token

Method body

Short version:

```
mylist.replaceAll(s -> s.toUpperCase());
```

# The Enhanced APIs That Use Lambda

There are three enhanced APIs that take advantage of lambda expressions:

- `java.util.functions` – *New*
  - Provides target types for lambda expressions
- `java.util.stream` – *New*
  - Provides classes that support operations on streams of values
- `java.util` – *Enhanced*
  - Interfaces and classes that make up the collections framework
    - Enhanced to use lambda expressions
    - Includes List and ArrayList

# Lambda Types

A lambda *type* specifies the type of expression a method is expecting.

- `replaceAll` takes a `UnaryOperator` type expression.

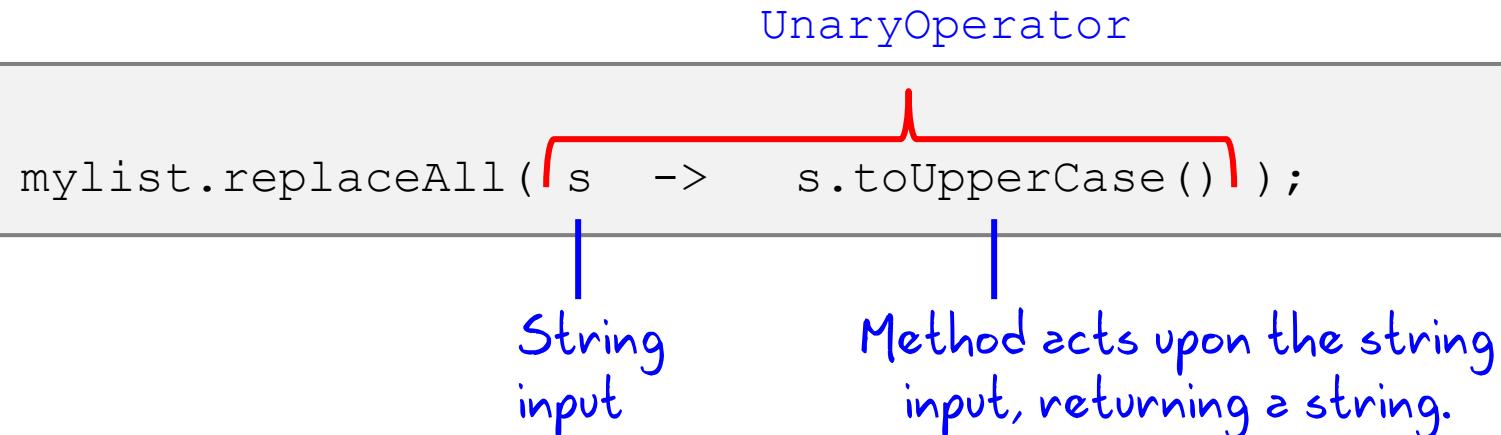
| Method Summary    |                  |                                                                                              |                 |
|-------------------|------------------|----------------------------------------------------------------------------------------------|-----------------|
| All Methods       | Instance Methods | Abstract Methods                                                                             | Default Methods |
| Modifier and Type |                  | Method and Description                                                                       |                 |
| default void      |                  | <code>replaceAll(UnaryOperator&lt;E&gt; operator)</code>                                     |                 |
|                   |                  | Replaces each element of this list with the result of applying the operator to that element. |                 |

- All of the types do similar things, but have different inputs, statements, and outputs.

# The UnaryOperator Lambda Type

A `UnaryOperator` has a single input and returns a value of the same type as the input.

- Example: String *in* – String *out*
- The method body acts upon the input in some way, returning a value of the same type as the input value.
- `replaceAll` example:



# The Predicate Lambda Type

A Predicate type takes a single input argument and returns a boolean.

- Example: String *in* – boolean *out*
- removeIf takes a Predicate type expression.
  - Removes all elements of the ArrayList that satisfy the Predicate expression

```
removeIf
```

```
public boolean removeIf(Predicate<? super E> filter)
```

- Examples:

```
mylist.removeIf (s -> s.equals("Rick"));
mylist.removeIf (s -> s.length() < 5);
```

## Exercise 13-2: Using a Predicate Lambda Expression

In this exercise, you use the `removeIf()` method to remove all items of the shopping cart whose description matches some value.

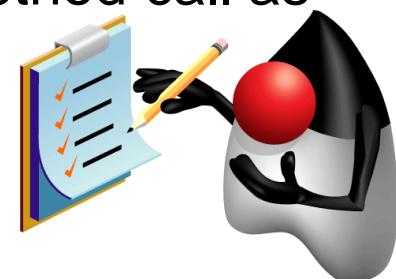
- Code the `removeItemFromCart()` method of `ShoppingCart`.
- Create a `Predicate` `lambda` expression that takes an `Item` object as input to the expression.



# Summary

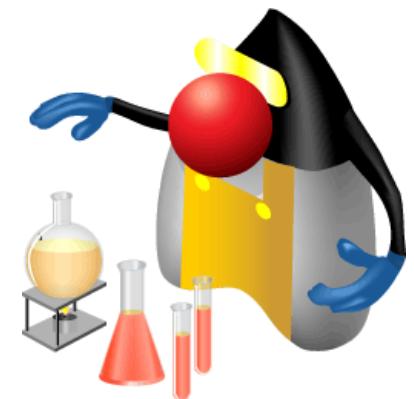
In this lesson, you should have learned the following:

- Polymorphism provides the following benefits:
  - Different classes have the same methods.
  - Method implementations can be unique for each class.
- Interfaces provide the following benefits:
  - You can link classes in different object hierarchies by their common behavior.
  - An object that implements an interface can be assigned to a reference of the interface type.
- Lambda expressions allow you to pass a method call as the argument to another method.



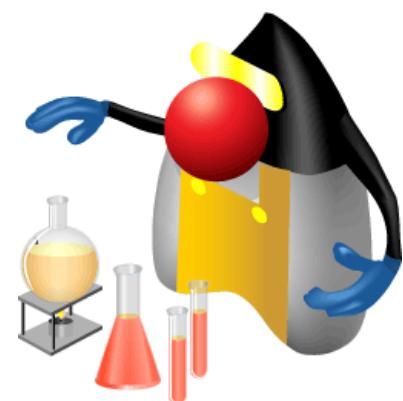
# **Practice 13-1 Overview: Overriding the `toString` Method**

This practice covers overriding the `toString` method in `Goal` and `Possession`.



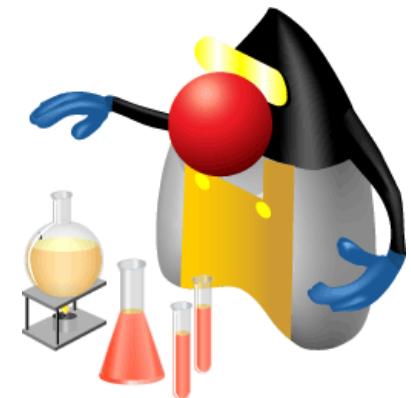
## Practice 13-2 Overview: Implementing an Interface

This practice covers implementing the Comparable interface so that you can order the elements in an array.



## **Practice 13-3 (Optional) Overview: Using a Lambda Expression for Sorting**

This practice covers using a lambda expression to sort the players.



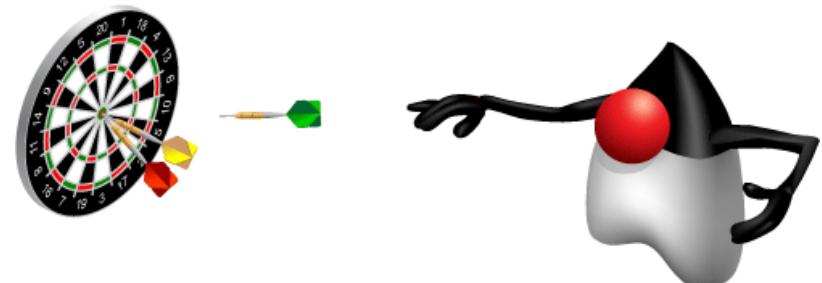
# 14

## Handling Exceptions

# Objectives

After completing this lesson, you should be able to:

- Describe how Java handles unexpected events in a program
- List the three types of `Throwable` classes
- Determine what exceptions are thrown for any foundation class
- Describe what happens in the call stack when an exception is thrown and not caught
- Write code to handle an exception thrown by the method of a foundation class



# Topics

- Handling exceptions: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors

# What Are Exceptions?

Java handles unexpected situations using exceptions.

- Something unexpected happens in the program.
- Java doesn't know what to do, so it:
  - Creates an exception object containing useful information and
  - Throws the exception to the code that invoked the problematic method
- There are several different types of exceptions.

# Examples of Exceptions

- `java.lang.ArrayIndexOutOfBoundsException`
  - Attempt to access a nonexistent array index
- `java.lang.ClassCastException`
  - Attempt to cast an object to an illegal type
- `java.lang.NullPointerException`
  - Attempt to use an object reference that has not been instantiated
- You can create exceptions, too!
  - An exception is just a class.

```
public class MyException extends Exception { }
```

# Code Example

Coding mistake:

```
01 int[] intArray = new int[5];
02 intArray[5] = 27;
```

Output:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
 at TestErrors.main(TestErrors.java:17)
```

# Another Example

Calling code in main:

```
19 TestArray myTestArray = new TestArray(5);
20 myTestArray.addElement(5, 23);
```

TestArray class:

```
13 public class TestArray {
14 int[] intArray;
15 public TestArray (int size) {
16 intArray = new int[size];
17 }
18 public void addElement(int index, int value) {
19 intArray[index] = value;
20 }
```

Stack trace:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
 at TestArray.addElement (TestArray.java:19)
 at TestException.main (TestException.java:20)
Java Result: 1
```

# Types of Throwable classes

Exceptions are subclasses of Throwable. There are three main types of Throwable:

- Error
  - Typically an unrecoverable external error
  - Unchecked
- RuntimeException
  - Typically caused by a programming mistake
  - Unchecked
- Exception
  - Recoverable error
  - Checked (*Must be caught or thrown*)

# Error Example: OutOfMemoryError

Programming error:

```
01 ArrayList theList = new ArrayList();
02 while (true) {
03 String theString = "A test String";
04 theList.add(theString);
05 long size = theList.size();
06 if (size % 1000000 == 0) {
07 System.out.println("List has "+size/1000000
08 +" million elements!");
09 }
10 }
```

Output in console:

```
List now has 156 million elements!
List now has 157 million elements!
Exception in thread "main" java.lang.OutOfMemoryError: Java
heap space
```

# Quiz

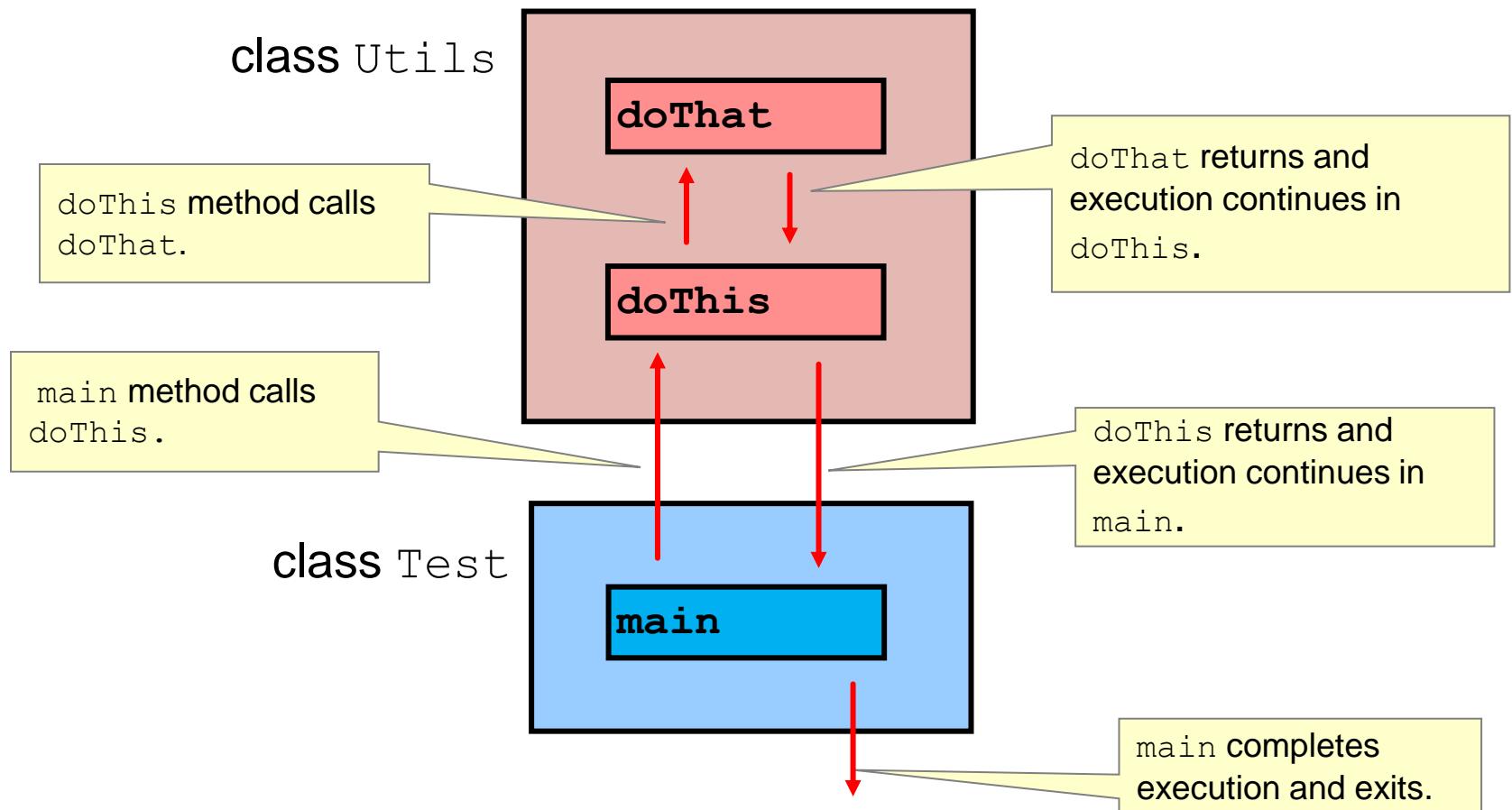
Which of the following objects are checked exceptions?

- a. All objects of type `Throwable`
- b. All objects of type `Exception`
- c. All objects of type `Exception` that are not of type `RuntimeException`
- d. All objects of type `Error`
- e. All objects of type `RuntimeException`

# Topics

- Handling errors: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors

# Normal Program Execution: The Call Stack



# How Exceptions Are Thrown

Normal program execution:

1. Caller method calls worker method.
2. Worker method does work.
3. Worker method completes work and then execution returns to caller method.

When an exception occurs, this sequence changes. An exception object is thrown and either:

- Passed to a `catch` block in the current method  
*or*
- Thrown back to the caller method

# Topics

- Handling errors: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors

# Working with Exceptions in NetBeans

```
10 public class Utils {
11
12 public void doThis() {
13
14 System.out.println("Arrived in doThis()");
15 doThat();
16 System.out.println("Back in doThis()");
17 }
18
19 public void doThat() {
20 System.out.println("In doThat()");
21 }
22 }
23
24 }
```

No exceptions thrown;  
nothing needs be done to  
deal with them.

When you throw an  
exception, NetBeans  
gives you two options.

```
12 public void doThis() {
13
14 System.out.println("Arrived in doThis()");
15 doThat();
16 System.out.println("Back in doThis()");
17 }
18
19 public void doThat() {
20 System.out.println("In doThat()");
21 throw new Exception();
22 }
23
24 }
```

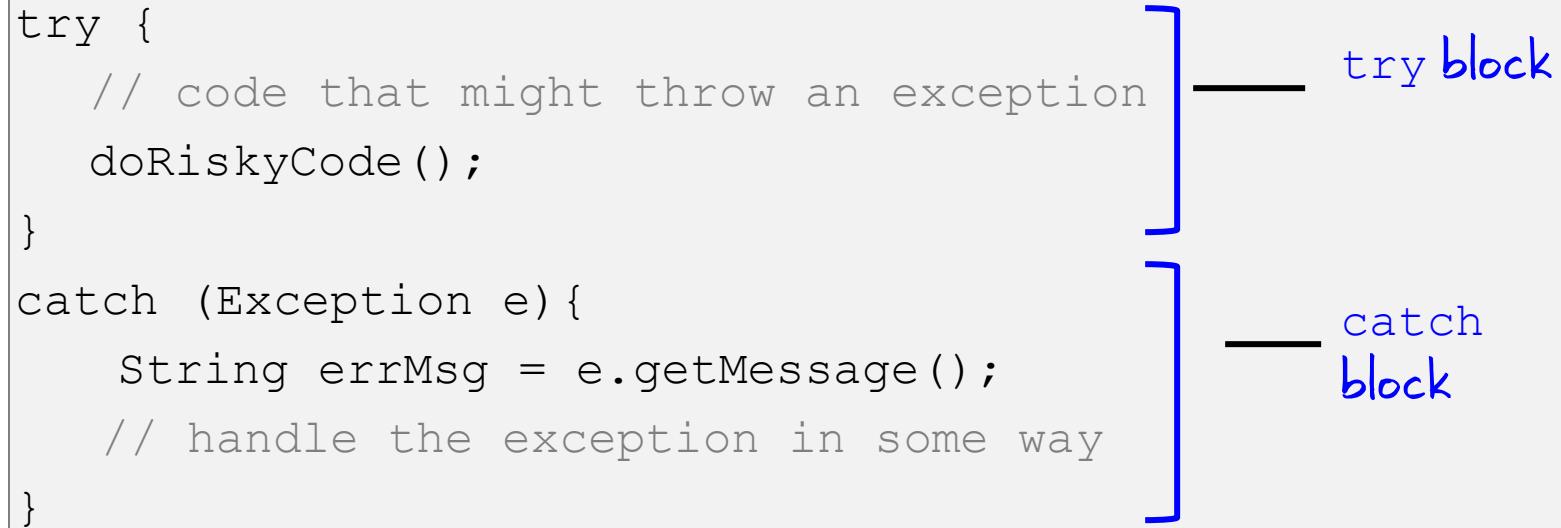
unreported exception java.lang.Exception;  
must be caught or declared to be thrown  
--  
(Alt-Enter shows hints)

# The **try/catch** Block

Option 1: Catch the exception.

```
try {
 // code that might throw an exception
 doRiskyCode();
}

catch (Exception e) {
 String errMsg = e.getMessage();
 // handle the exception in some way
}
```



A diagram illustrating the structure of a try/catch block. It shows two curly braces enclosing the code. A blue line points from the text "try block" to the outer brace. Another blue line points from the text "catch block" to the inner brace.

Option 2: Throw the exception.

```
public void doThat() throws Exception{
 // code that might throw an exception
 doRiskyCode();
}
```

# Program Flow When an Exception Is Caught

main method:

```
01 Utils theUtils = new Utils();
02 theUtils.doThis();
03 System.out.println("Back to main method");
```

Output

3

Utils class methods:

```
04 public void doThis() {
05 try{
06 doThat();
07 }catch(Exception e){
08 System.out.println("doThis - "
09 +" Exception caught: "+e.getMessage());
10 }
11 }
12 public void doThat() throws Exception{
13 System.out.println("doThat: Throwing exception");
14 throw new Exception("Ouch!");
15 }
```

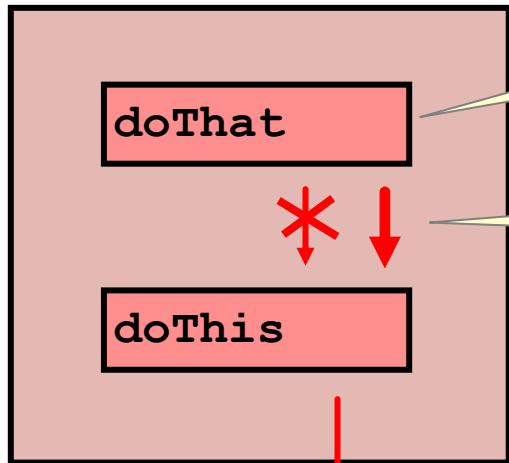
```
run:
doThat: throwing Exception
doThis - Exception caught: Ouch!
Back to main method
BUILD SUCCESSFUL (total time: 0 seconds)
```

2

1

# When an Exception Is Thrown

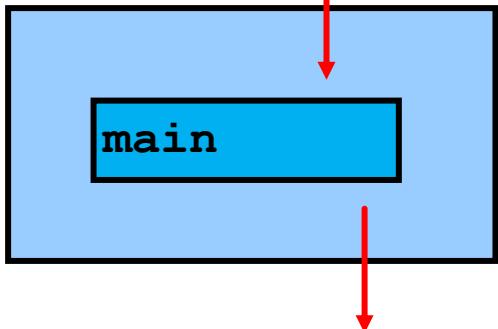
class Utils



Exception thrown in doThat

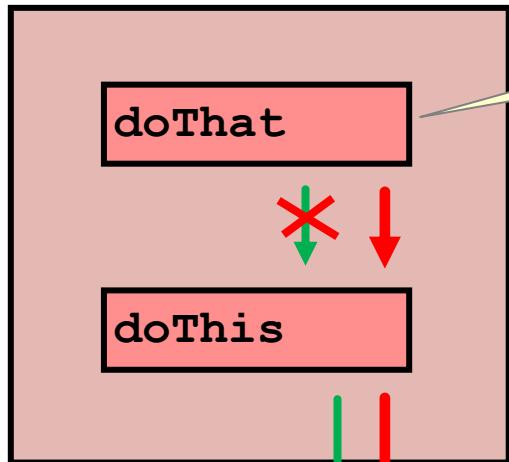
Execution returns to doThis  
and must be caught or thrown.

class Test



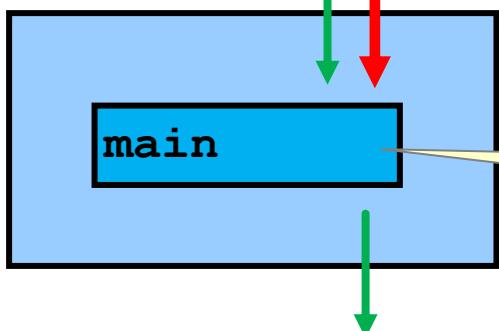
# Throwing Throwable Objects

class Utils



Exception thrown in `doThat`

class Test

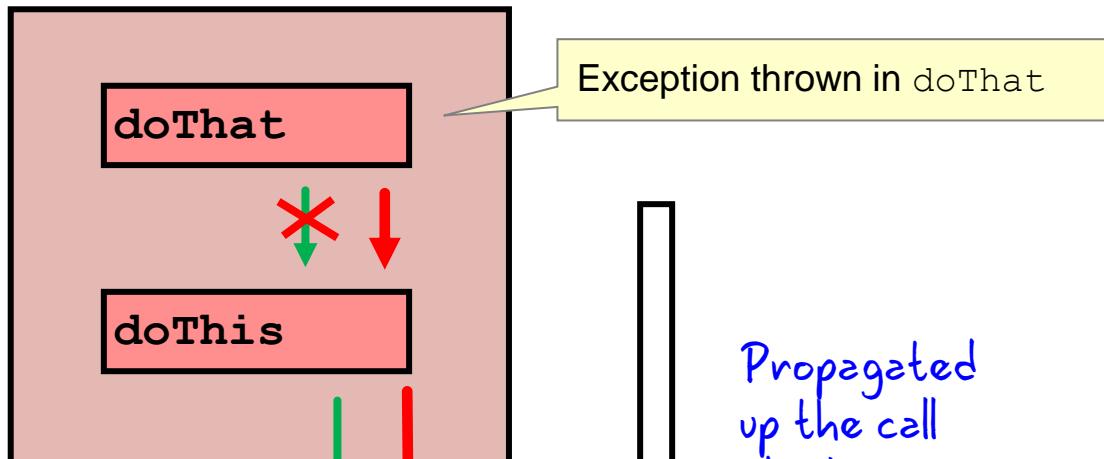


If `doThis` does NOT catch the exception, then ...

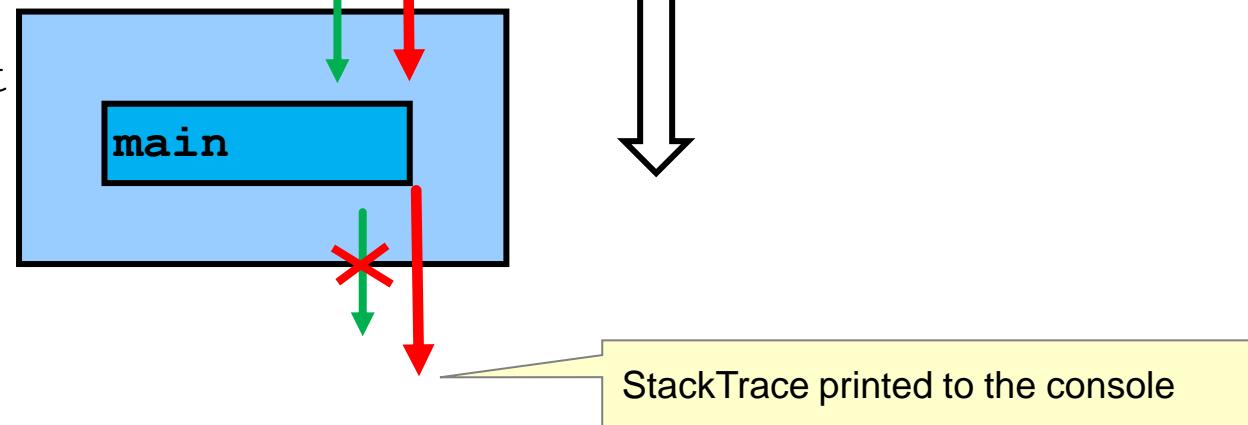
... `main` must catch it OR throw it.

# Uncaught Exception

class Utils

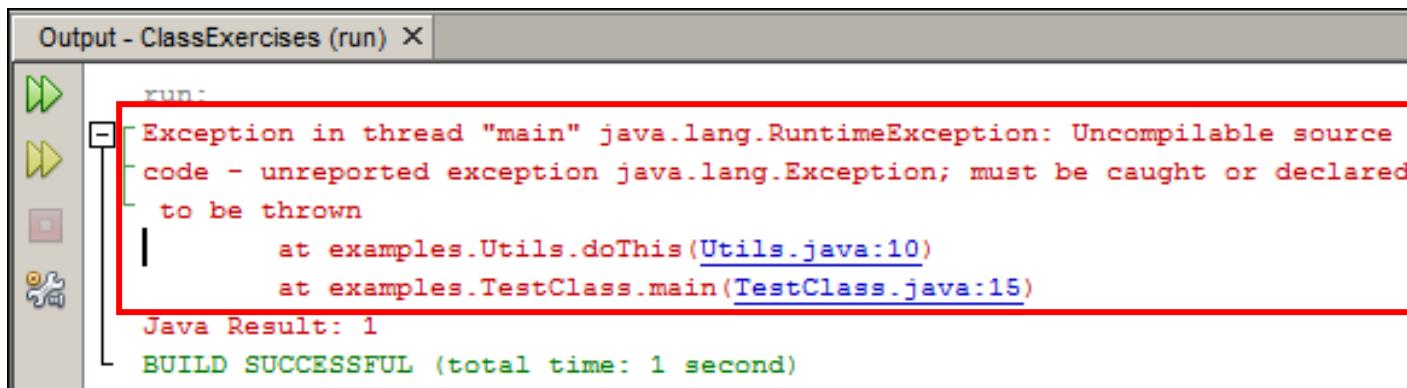


class Test



# Exception Printed to Console

When the exception is thrown up the call stack without being caught, it will eventually reach the JVM. The JVM will print the exception's output to the console and exit.



The screenshot shows an IDE's output window titled "Output - ClassExercises (run)". The window contains the following text:

```
RUN:
Exception in thread "main" java.lang.RuntimeException: Uncompilable source
code - unreported exception java.lang.Exception; must be caught or declared
to be thrown
 at examples.Utils.doThis(Utils.java:10)
 at examples.TestClass.main(TestClass.java:15)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

A red box highlights the exception message and the call stack trace.

# Summary of Exception Types

A `Throwable` is a special type of Java object.

- It is the only object type that:
  - Is used as the argument in a catch clause
  - Can be “thrown” to the calling method
- It has two direct subclasses:
  - `Error`
    - Automatically propagated up the call stack to the calling method
  - `Exception`
    - Must be explicitly handled and requires either:
      - A `try/catch` block to handle the error
      - A `throws` in the method signature to propagate up the call stack
    - Has a subclass `RuntimeException`
      - Automatically propagated up the call stack to the calling method

# Exercise 14-1: Catching an Exception

In this exercise, you work with the `ShoppingCart` class and a `Calculator` class to implement exception handling.

- Change a method signature to indicate that it throws an exception.
- Catch the exception in the class that calls the method.



# Quiz

Which one of the following statements is true?

- a. A RuntimeException must be caught.
- b. A RuntimeException must be thrown.
- c. A RuntimeException must be caught or thrown.
- d. A RuntimeException is thrown automatically.

# Exceptions in the Java API Documentation

Method Summary

Methods

| Modifier and Type | Method and Description                                                                                                                                   |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean           | <b>canExecute ()</b><br>Tests whether the application can execute the file denoted by this abstract pathname.                                            |
| boolean           | <b>canRead ()</b><br>Tests whether the application can read the file denoted by this abstract pathname.                                                  |
| boolean           | <b>canWrite ()</b><br>Tests whether the application can modify the file denoted by this abstract pathname.                                               |
| int               | <b>compareTo (File pathname)</b><br>Compares two abstract pathnames lexicographically.                                                                   |
| boolean           | <b>createNewFile ()</b><br>Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. |

These are methods of the File Class.

**createNewFile**

```
public boolean createNewFile()
 throws IOException
```

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The `FileLock` facility should be used instead.

**Returns:**

true if the named file does not exist and was successfully created; false if the named file already exists

**Throws:**

`IOException` - If an I/O error occurred  
`SecurityException` - If a security manager exists and its `SecurityManager.checkWrite (java.lang.String)` method denies write access to the file

**Since:**

1.2

Click to get the detail of `createNewFile`.

Note the exceptions that can be thrown.

14 - 506

# Calling a Method That Throws an Exception

```
53 public void testCheckedException() {
54 File testFile = new File("//testFile.txt");
55
56 System.out.println("testFile exists: " + testFile.exists());
57 testFile.delete();
58 System.out.println("testFile exists: " + testFile.exists());
59 }
```

Constructor causes no compilation problems.

```
53 public void testCheckedException() {
54 File testFile = new File("//testFile.txt");
55
56 testFile.createNewFile(); // unreported exception IOException; must be caught or declared to be thrown
57
58 System.out.println("testFile exists: " + testFile.exists());
59 testFile.delete();
60 System.out.println("testFile exists: " + testFile.exists());
61 }
```

createNewFile can throw a checked exception, so the method must throw or catch.

# Working with a Checked Exception

Catching IOException:

```
01 public static void main(String[] args) {
02 TestClass testClass = new TestClass();
03
04 try {
05 testClass.testCheckedException();
06 } catch (IOException e) {
07 System.out.println(e);
08 }
09 }
10
11 public void testCheckedException() throws IOException {
12 File testFile = new File("//testFile.txt");
13 testFile.createNewFile();
14 System.out.println("testFile exists:"
15 + testFile.exists());
16 }
```

# Best Practices

- Catch the actual exception thrown, not the superclass type.
- Examine the exception to find out the exact problem so you can recover cleanly.
- You do not need to catch every exception.
  - A programming mistake should not be handled. It must be fixed.
  - Ask yourself, “Does this exception represent behavior I want the program to recover from?”

# Bad Practices

```
01 public static void main(String[] args) {
02 try {
03 createFile("c:/testFile.txt");
04 } catch (Exception e) { ————— Catching superclass?
05 System.out.println("Error creating file.");
06 }
07 }
08 public static void createFile(String name) [No processing of
09 throws IOException;
10 File f = new File(name);
11 f.createNewFile();
12
13 int[] intArray = new int[5];
14 intArray[5] = 27;
15 }
```

# Somewhat Better Practice

```
01 public static void main(String[] args) {
02 try {
03 createFile("c:/testFile.txt");
04 } catch (Exception e) {
05 System.out.println(e);
06 //<other actions>
07 }
08 }
09 public static void createFile(String fname)
10 throws IOException{
11 File f = new File(name);
12 System.out.println(name+" exists? "+f.exists());
13 f.createNewFile();
14 System.out.println(name+" exists? "+f.exists());
15 int[] intArray = new int[5];
16 intArray[5] = 27;
17 }
```

What is the object type?  
toString() is called on this object.

# Topics

- Handling errors: an overview
- Propagation of exceptions
- Catching and throwing exceptions
- Multiple exceptions and errors

# Multiple Exceptions

```
01 public static void createFile() throws IOException {
02 File testF = new File("c:/notWriteableDir");
03
04 File tempF = testF.createTempFile("te", null, testF);
05
06 System.out.println
07 ("Temp filename: "+tempF.getPath());
08 int myInt[] = new int[5];
09 myInt[5] = 25;
11 }
```

Directory must be writeable:  
IOException

Arg must be greater than  
3 characters:  
IllegalArgumentException

Array index must be valid:  
ArrayIndexOutOfBoundsException

# Catching IOException

```
01 public static void main(String[] args) {
02 try {
03 createFile();
04 } catch (IOException ioe) {
05 System.out.println(ioe);
06 }
07 }
08
09 public static void createFile() throws IOException {
10 File testF = new File("c:/notWriteableDir");
11 File tempF = testF.createTempFile("te", null, testF);
12 System.out.println("Temp filename: "+tempF.getPath());
13 int myInt[] = new int[5];
14 myInt[5] = 25;
15 }
```

# Catching `IllegalArgumentException`

```
01 public static void main(String[] args) {
02 try {
03 createFile();
04 } catch (IOException ioe) {
05 System.out.println(ioe);
06 } catch (IllegalArgumentException iae) {
07 System.out.println(iae);
08 }
09 }
10
11 public static void createFile() throws IOException {
12 File testF = new File("c:/writeableDir");
13 File tempF = testF.createTempFile("te", null, testF);
14 System.out.println("Temp filename: "+tempF.getPath());
15 int myInt[] = new int[5];
16 myInt[5] = 25;
17 }
```

# Catching Remaining Exceptions

```
01 public static void main(String[] args) {
02 try {
03 createFile();
04 } catch (IOException ioe) {
05 System.out.println(ioe);
06 } catch (IllegalArgumentException iae) {
07 System.out.println(iae);
08 } catch (Exception e) {
09 System.out.println(e);
10 }
11 }
12 public static void createFile() throws IOException {
13 File testF = new File("c:/writeableDir");
14 File tempF = testF.createTempFile("te", null, testF);
15 System.out.println("Temp filename: "+tempF.getPath());
16 int myInt[] = new int[5];
17 myInt[5] = 25;
18 }
```

# Summary

In this lesson, you should have learned how to:

- Describe the different kinds of errors that can occur and how they are handled in Java
- Describe what exceptions are used for in Java
- Determine what exceptions are thrown for any foundation class
- Write code to handle an exception thrown by the method of a foundation class



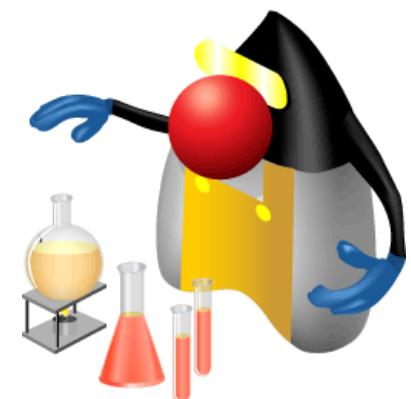
# Interactive Quizzes



# Practice 14-1 Overview: Adding Exception Handling

This practice covers the following topics:

- Investigating how the Soccer application can break under certain circumstances
- Modifying your code to handle the exceptions gracefully

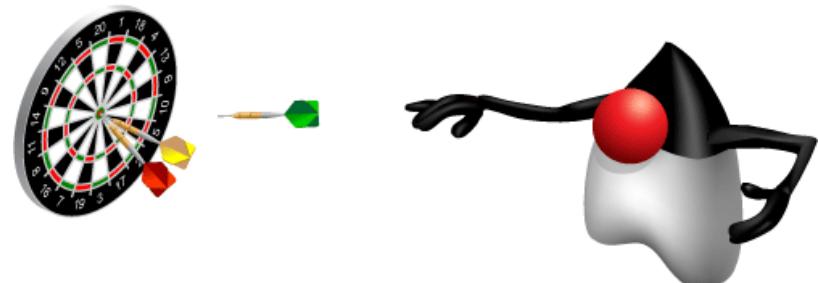


# **Deploying and Maintaining the Soccer Application**

# Objectives

After completing this lesson, you should be able to:

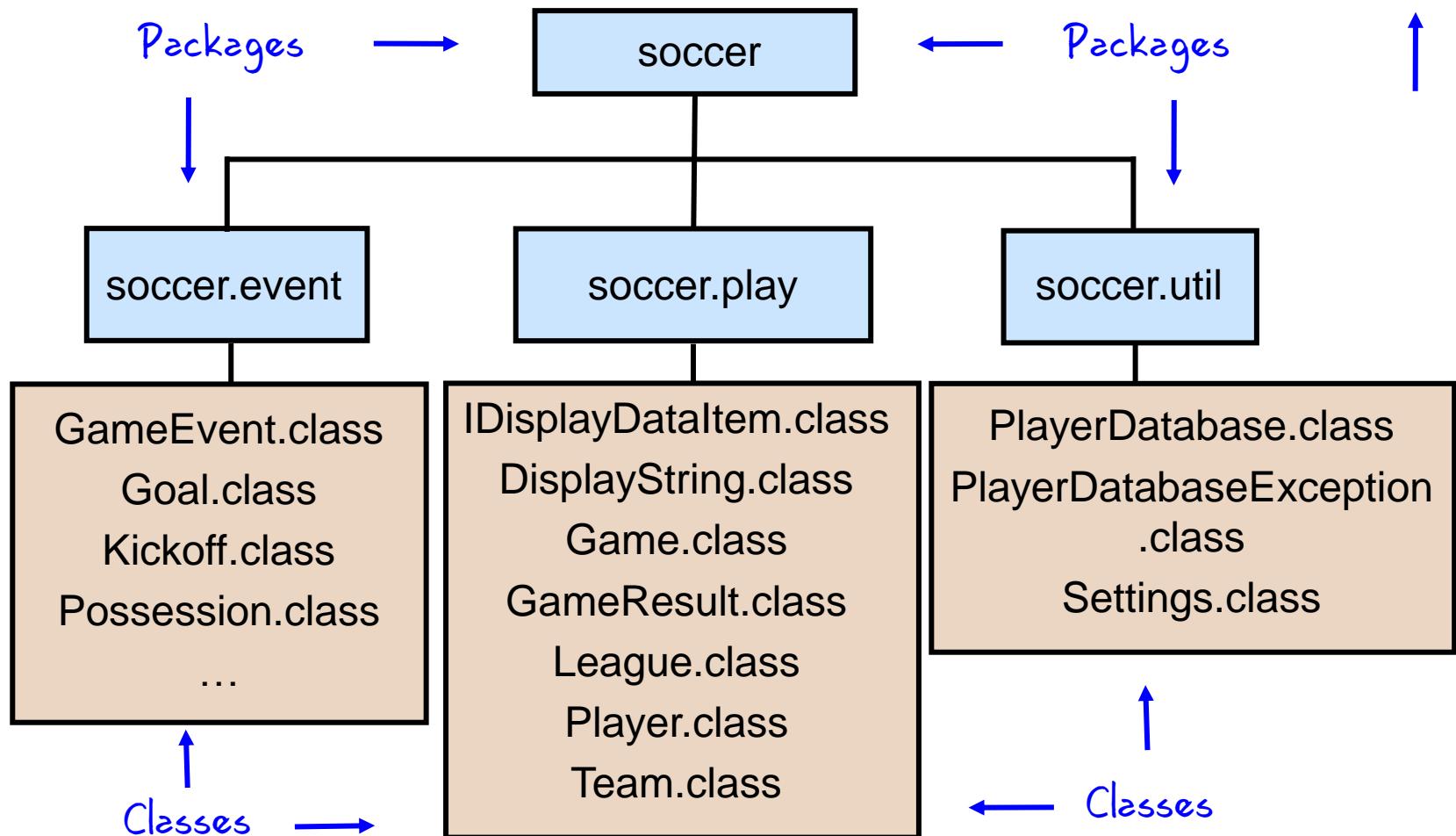
- Deploy a simple application as a JAR file
- Describe the parts of a Java application, including the user interface and the back end
- Describe how classes can be extended to implement new capabilities in the application



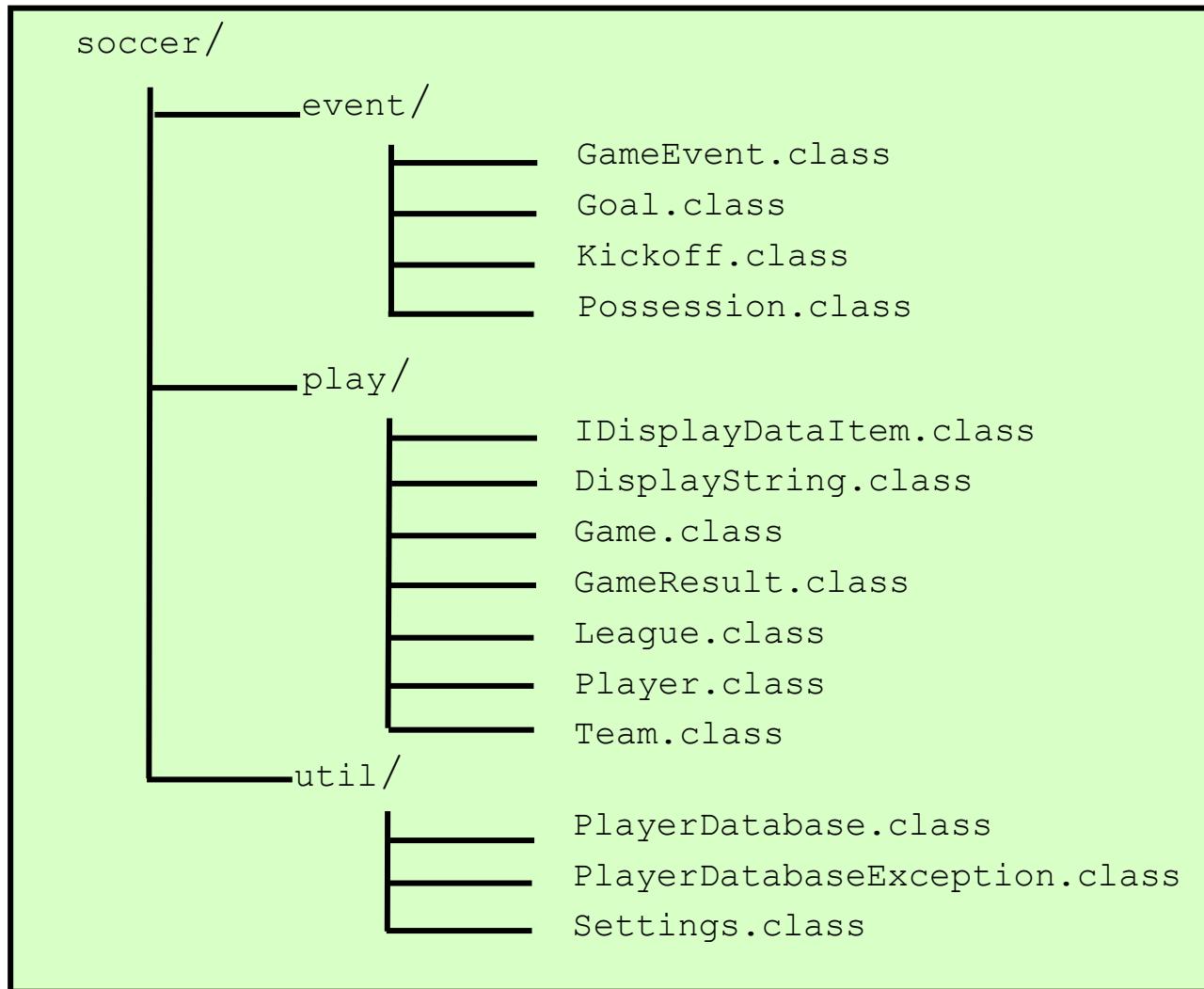
# Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements

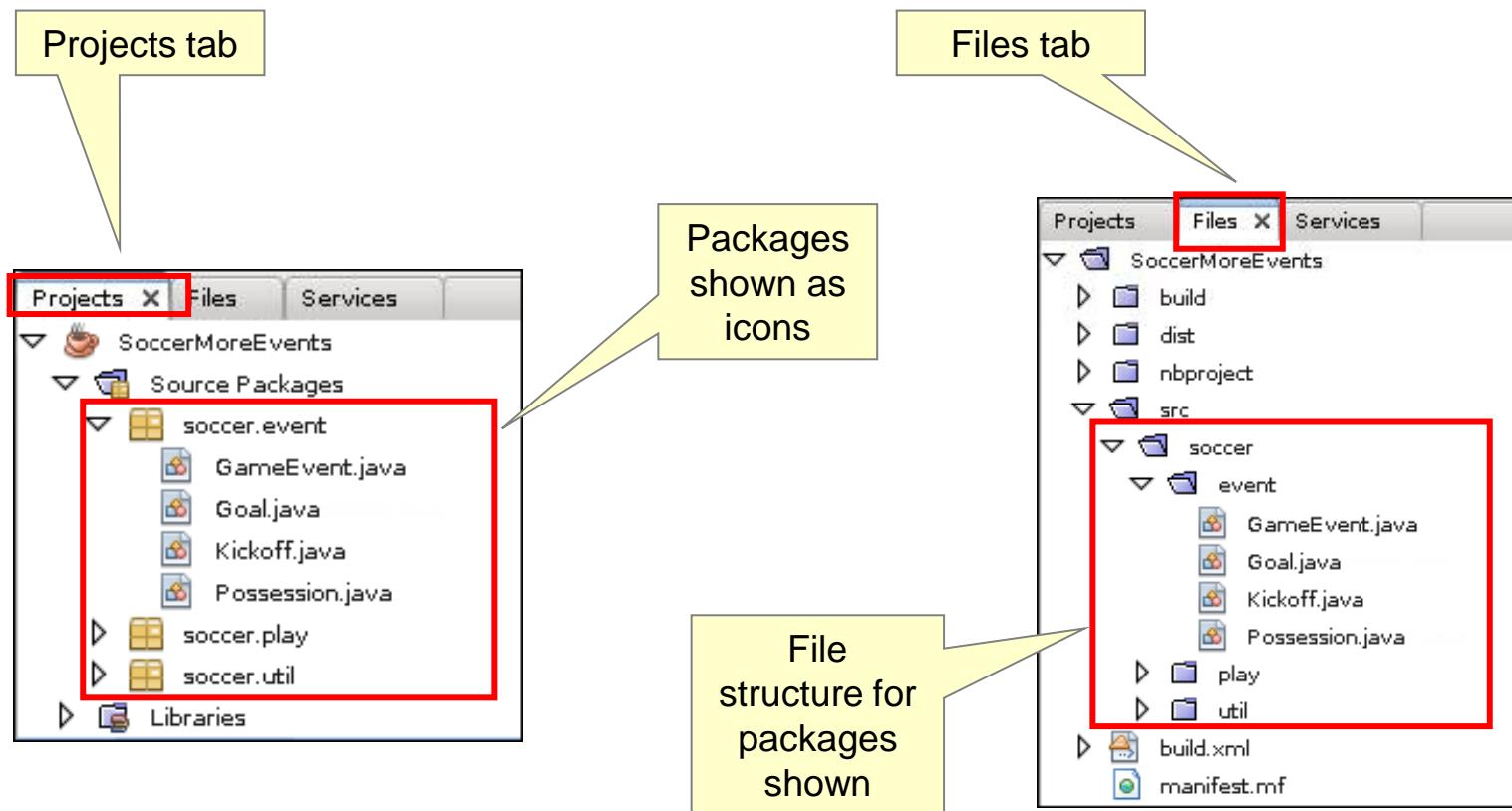
# Packages



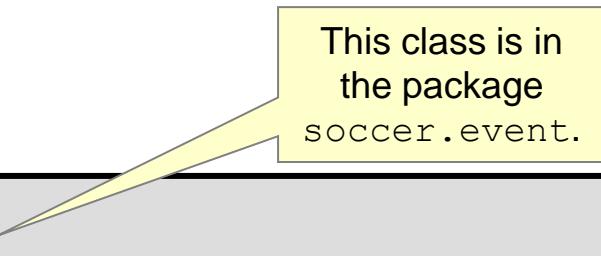
# Packages Directory Structure



# Packages in NetBeans



# Packages in Source Code



This class is in  
the package  
soccer.event.

```
package soccer.event;

public class Goal extends GameEvent {

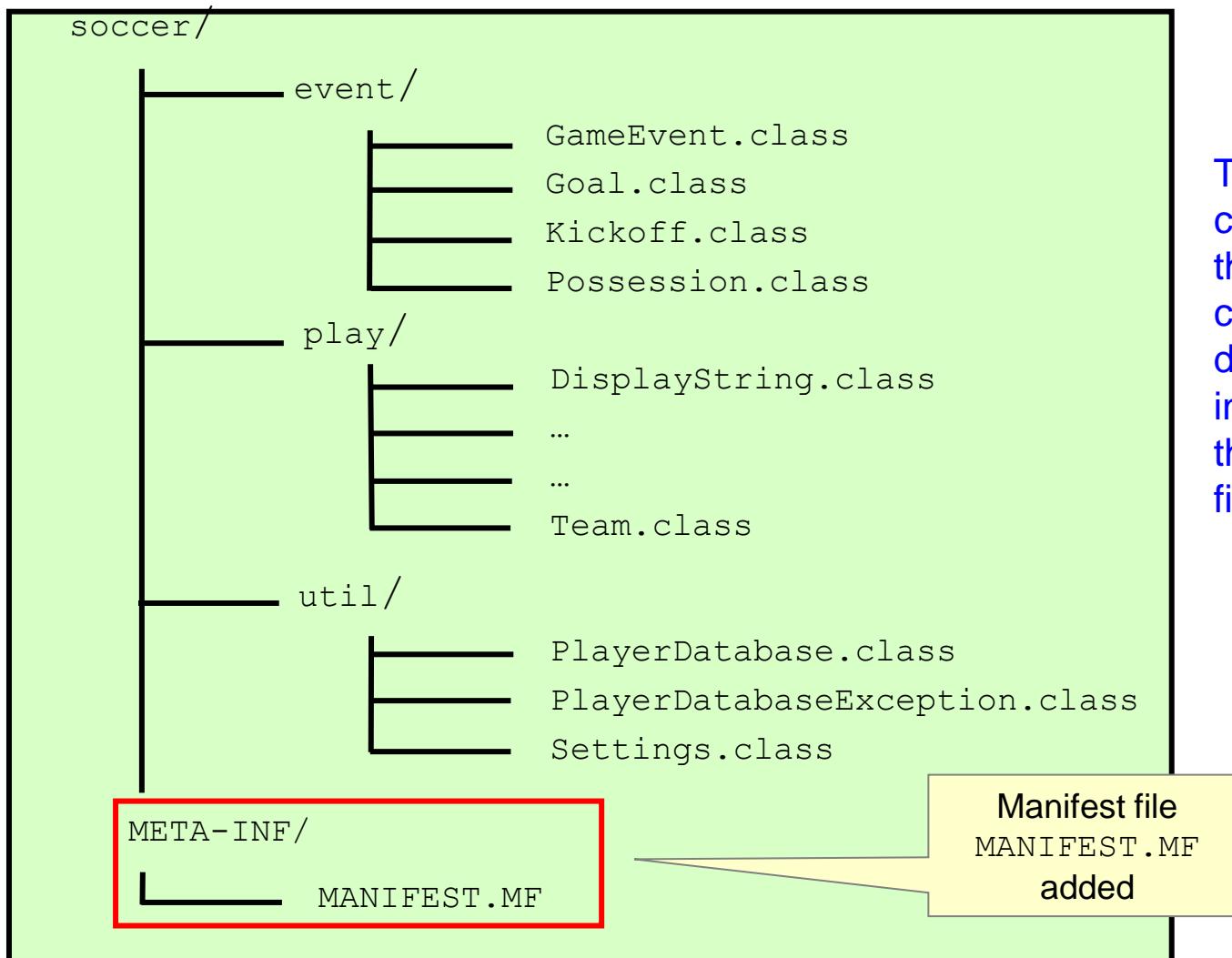
 public String toString() {
 return "GOAL! ";
 }
 ... < remaining code omitted > ...
}
```

The package that a class belongs to is defined in the source code.

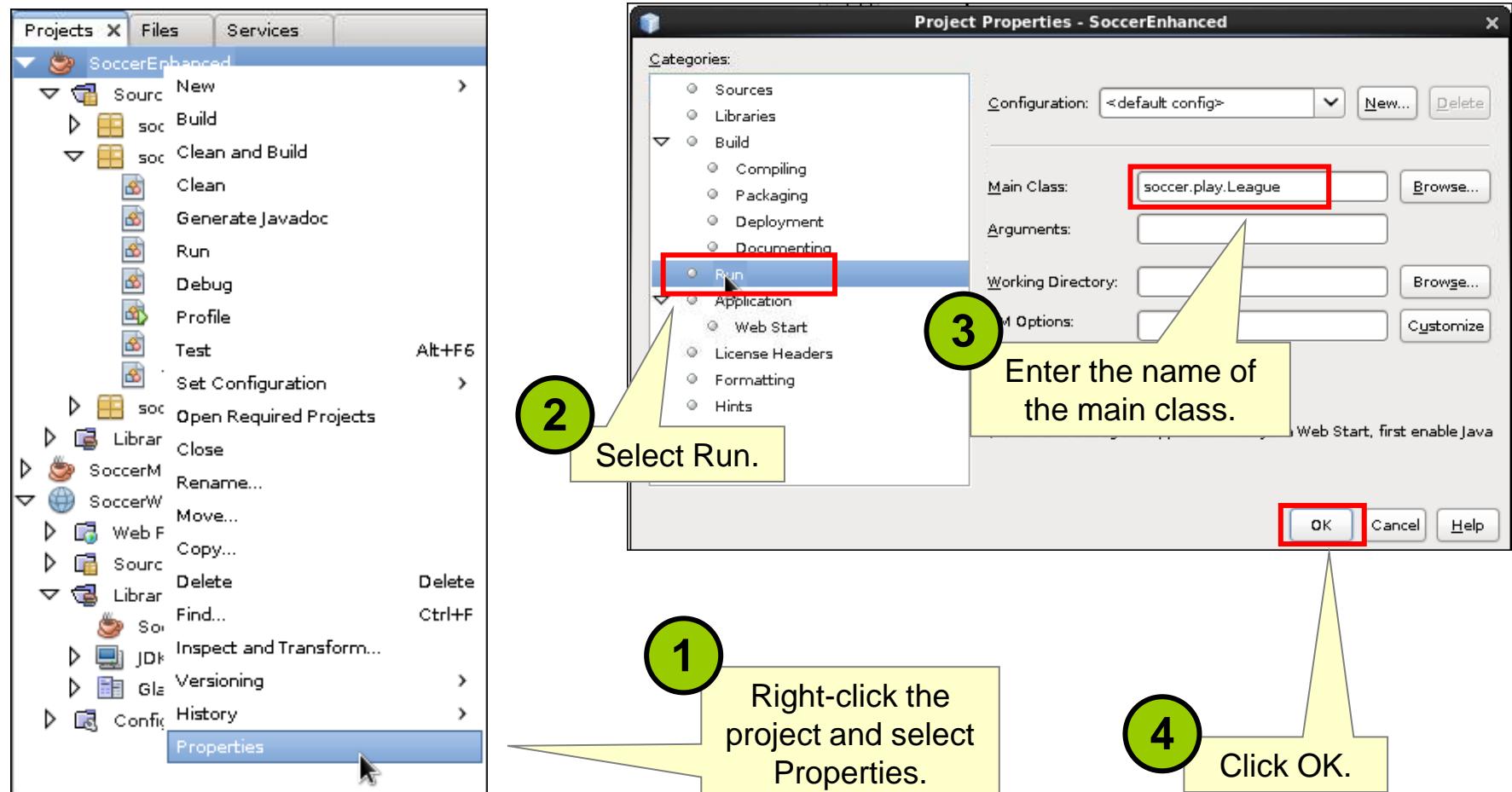
# Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements

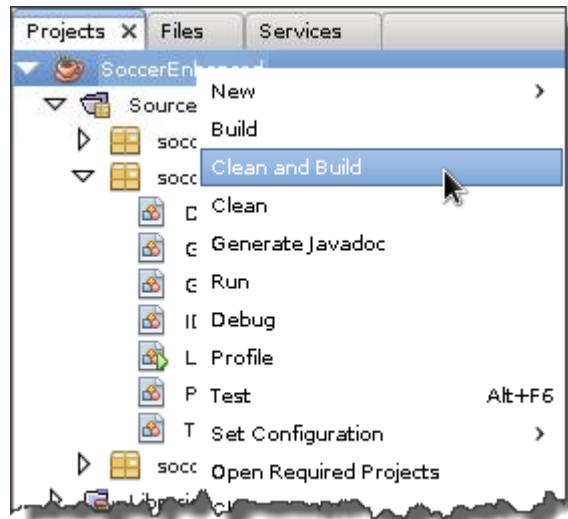
# SoccerEnhanced.jar



# Set Main Class of Project



# Creating the JAR File with NetBeans



1

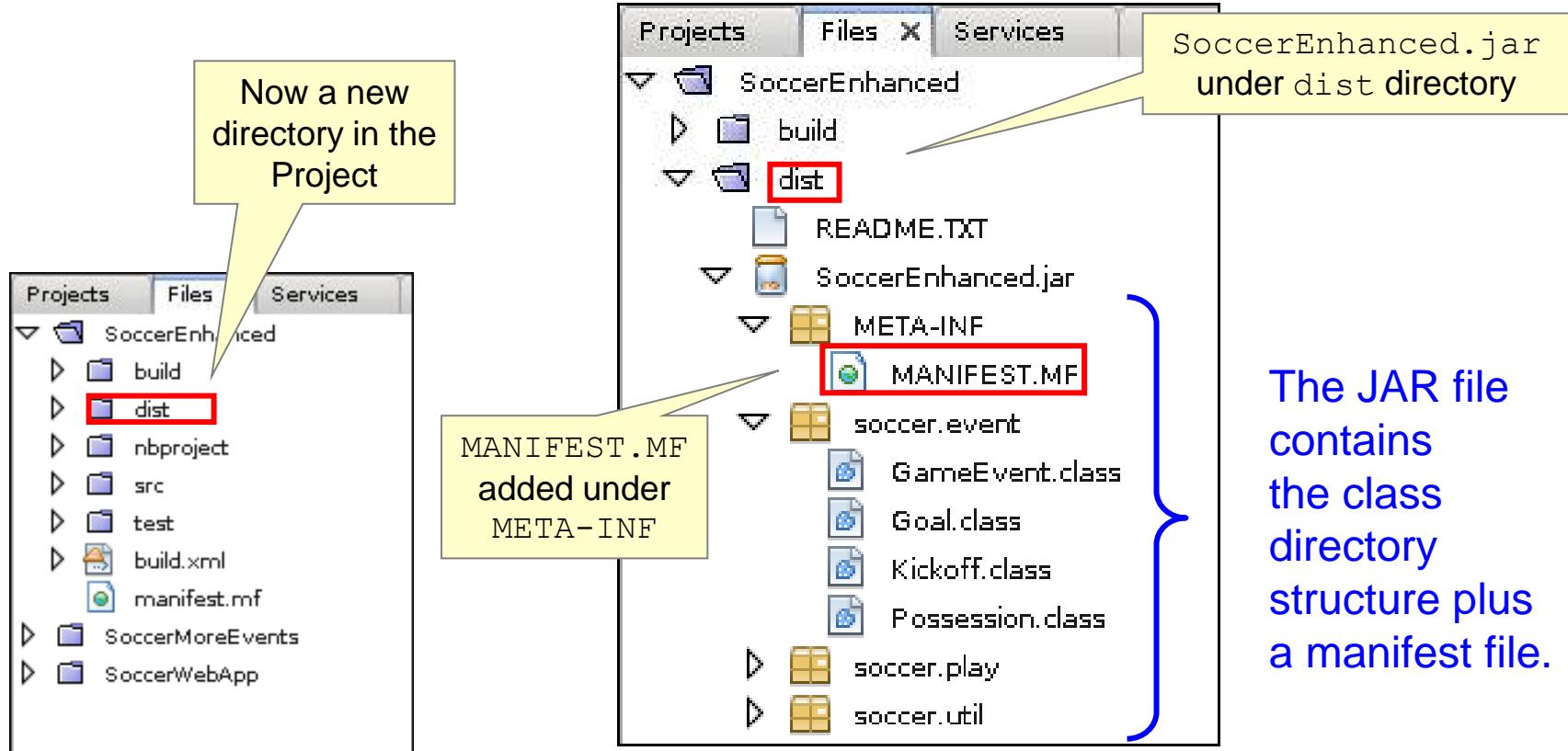
Right-click the project and select "Clean and Build."

2

Check the output to ensure that the build is successful.

```
Output - SoccerEnhanced (clean,jar) X
compile.
Created dir: /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist
Copying 1 file to /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/build
Nothing to copy.
Building jar: /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar
To run this application from the command line without Ant, try:
java -jar "/home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar"
jar:
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Creating the JAR File with NetBeans



# Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements

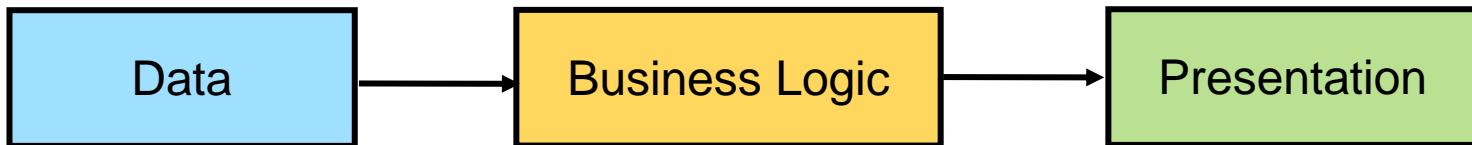
# **Client/Server Two-Tier Architecture**

Client/server computing involves two or more computers sharing tasks:

- Each computer performs logic appropriate to its design and stated function.
- The front-end client communicates with the back-end database.
- The client requests data from the back end.
- The server returns the appropriate results.
- The client handles and displays data.

# Client/Server Three-Tier Architecture

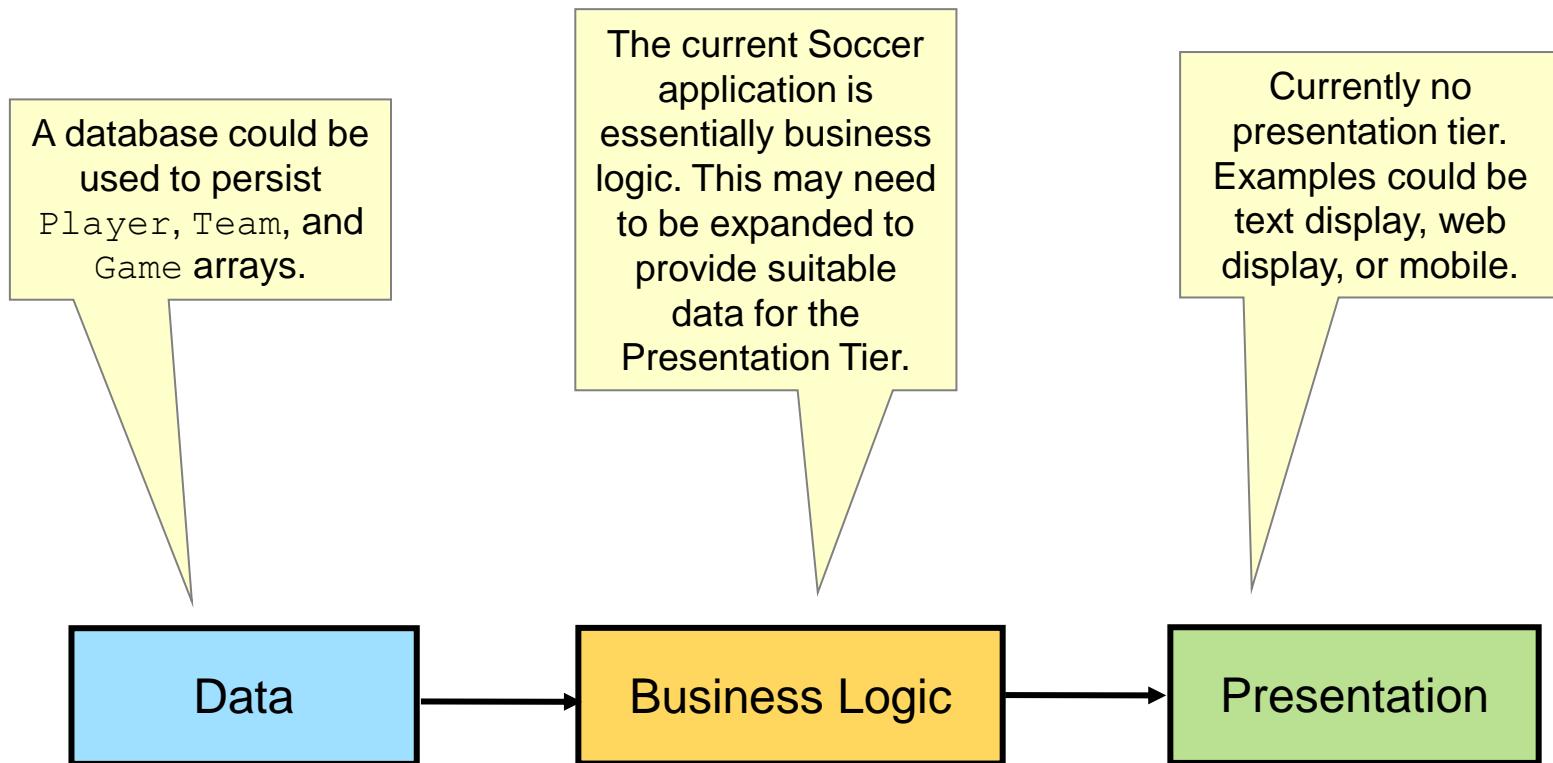
- Three-tier client/server is a more complex, flexible approach.
- Each tier can be replaced by a different implementation:
  - The data tier is an encapsulation of all existing data sources.
  - Business logic defines business rules.
  - Presentation can be GUI, web, smartphone, or even console.



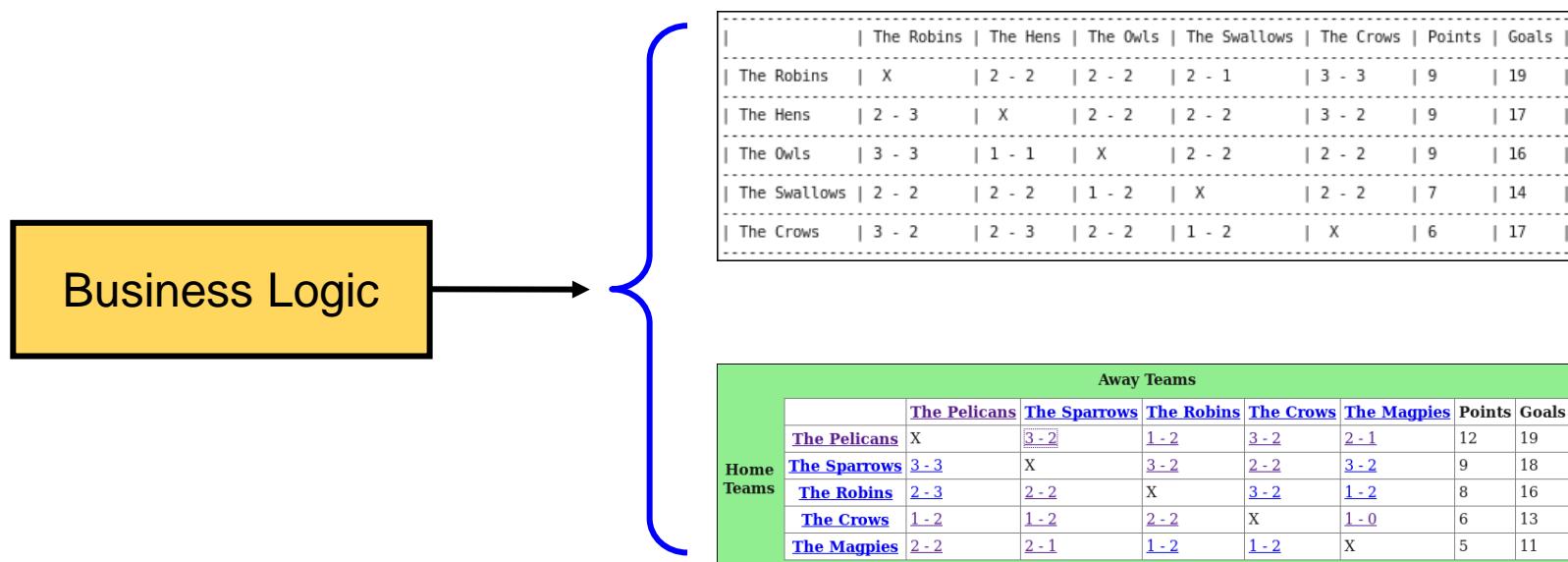
# Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- **The Soccer application**
- Application modifications and enhancements

# Client/Server Three-Tier Architecture



# Client/Server Three-Tier Architecture



# Different Outputs

A two-dimensional String array could provide the String output for each element of the grid, but this is inflexible:

- The presentation can only display the String provided.
- The presentation cannot access other useful information—for example, the data required to allow users to click on the score for more details.

|              | The Robins | The Hens | The Owls | The Swallows | The Crows | Points | Goals |
|--------------|------------|----------|----------|--------------|-----------|--------|-------|
| The Robins   | X          | 2 - 2    | 2 - 2    | 2 - 1        | 3 - 3     | 9      | 19    |
| The Hens     | 2 - 3      | X        | 2 - 2    | 2 - 2        | 3 - 2     | 9      | 17    |
| The Owls     | 3 - 3      | 1 - 1    | X        | 2 - 2        | 2 - 2     | 9      | 16    |
| The Swallows | 2 - 2      | 2 - 2    | 1 - 2    | X            | 2 - 2     | 7      | 14    |
| The Crows    | 3 - 2      | 2 - 3    | 2 - 2    | 1 - 2        | X         | 6      | 17    |

| Home Teams   | Away Teams   |              |            |           |             | Points | Goals |
|--------------|--------------|--------------|------------|-----------|-------------|--------|-------|
|              | The Pelicans | The Sparrows | The Robins | The Crows | The Magpies |        |       |
| The Pelicans | X            | 3 - 2        | 1 - 2      | 3 - 2     | 2 - 1       | 12     | 19    |
| The Sparrows | 3 - 3        | X            | 3 - 2      | 2 - 2     | 3 - 2       | 9      | 18    |
| The Robins   | 2 - 3        | 2 - 2        | X          | 3 - 2     | 1 - 2       | 8      | 16    |
| The Crows    | 1 - 2        | 1 - 2        | 2 - 2      | X         | 1 - 0       | 6      | 13    |
| The Magpies  | 2 - 2        | 2 - 1        | 1 - 2      | 1 - 2     | X           | 5      | 11    |

# The Soccer Application

- Abstract classes
  - GameEvent
    - Extended by Goal and other GameEvent classes
- Interfaces
  - Comparable
    - Implemented by Team and Player so that they can be ranked
  - IDisplayDataItem
    - Implemented by Team, Game, and DisplayString

# **IDisplayDataItem Interface**

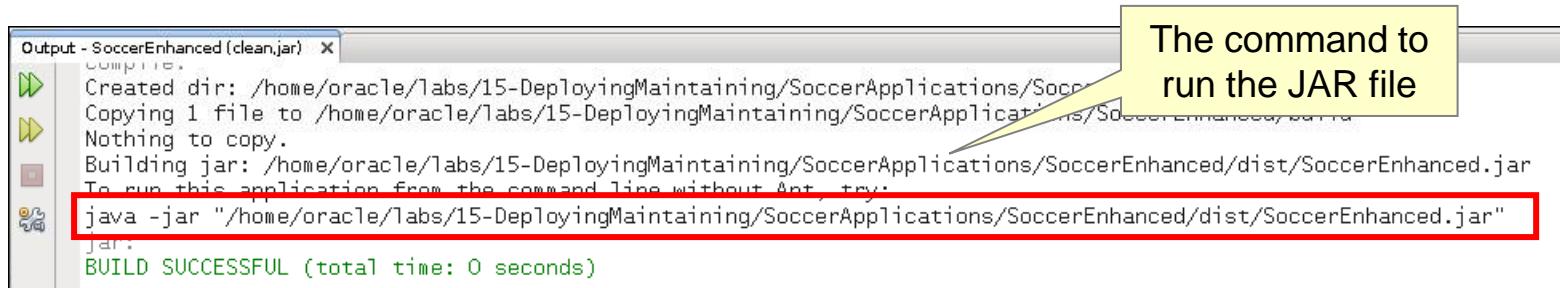
```
package soccer.play;

public interface IDisplayDataItem {

 public boolean isDetailAvailable ();
 public String getDisplayDetail();
 public int getID();
 public String getDetailType();

}
```

# Running the JAR File from the Command Line



The command to run the JAR file

```
Output - SoccerEnhanced (clean,jar) X
[INFO]
[INFO] Created dir: /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/target
[INFO] Copying 1 file to /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/target
[INFO] Nothing to copy.
[INFO] Building jar: /home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar
[INFO] To run this application from the command line without Ant, try:
[INFO] java -jar "/home/oracle/labs/15-DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar"
[INFO]
[INFO] BUILD SUCCESSFUL (total time: 0 seconds)
```

```
[oracle@EDBSR2P14 ~]$ java -jar "/home/oracle/labs/15-
DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar"

| The Robins | The Hens | The Owls | The Swallows | The Crows | Points | Goals |

| The Robins | X | 2 - 2 | 2 - 2 | 2 - 1 | 3 - 3 | 9 | 19 |

| The Hens | 2 - 3 | X | 2 - 2 | 2 - 2 | 3 - 2 | 9 | 17 |

| The Owls | 3 - 3 | 1 - 1 | X | 2 - 2 | 2 - 2 | 9 | 16 |

| The Swallows | 2 - 2 | 2 - 2 | 1 - 2 | X | 2 - 2 | 7 | 14 |

| The Crows | 3 - 2 | 2 - 3 | 2 - 2 | 1 - 2 | X | 6 | 17 |
```

# Text Presentation of the League

```
[oracle@EDBSR2P14 ~]$ java -jar "/home/oracle/labs/15-
DeployingMaintaining/SoccerApplications/SoccerEnhanced/dist/SoccerEnhanced.jar"
```

|              | The Robins | The Hens | The Owls | The Swallows | The Crows | Points | Goals |
|--------------|------------|----------|----------|--------------|-----------|--------|-------|
| The Robins   | X          | 2 - 2    | 2 - 2    | 2 - 1        | 3 - 3     | 9      | 19    |
| The Hens     | 2 - 3      | X        | 2 - 2    | 2 - 2        | 3 - 2     | 9      | 17    |
| The Owls     | 3 - 3      | 1 - 1    | X        | 2 - 2        | 2 - 2     | 9      | 16    |
| The Swallows | 2 - 2      | 2 - 2    | 1 - 2    | X            | 2 - 2     | 7      | 14    |
| The Crows    | 3 - 2      | 2 - 3    | 2 - 2    | 1 - 2        | X         | 6      | 17    |

The object type behind these data elements is Team.

The object type behind these data elements (except for the output Xs) is Game.

The object type behind these data elements is DisplayString.

# Web Presentation of the League

|            |              | Away Teams   |              |            |           |             |        |       |
|------------|--------------|--------------|--------------|------------|-----------|-------------|--------|-------|
| Home Teams |              | The Pelicans | The Sparrows | The Robins | The Crows | The Magpies | Points | Goals |
|            | The Pelicans | X            | 3 - 2        | 1 - 2      | 3 - 2     | 2 - 1       | 12     | 19    |
|            | The Sparrows | 3 - 3        | X            | 3 - 2      | 2 - 2     | 3 - 2       | 9      | 18    |
|            | The Robins   | 2 - 3        | 2 - 2        | X          | 3 - 2     | 1 - 2       | 8      | 16    |
|            | The Crows    | 1 - 2        | 1 - 2        | 2 - 2      | X         | 1 - 0       | 6      | 13    |
|            | The Magpies  | 2 - 2        | 2 - 1        | 1 - 2      | 1 - 2     | X           | 5      | 11    |

The object type behind these data elements is Team.

The object type behind these data elements (except for the output Xs) is Game.

The object type behind these data elements is DisplayString.

# Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Soccer application
- Application modifications and enhancements

# Enhancing the Application

- Well-designed Java software minimizes the time required for:
  - Maintenance
  - Enhancements
  - Upgrades
- For the Soccer application, it should be easy to:
  - Add new GameEvent subclasses (business logic)
  - Develop new clients (presentation)
    - Take the application to a smartphone (for example)
  - Change the storage system (data)

# **Adding a New GameEvent Kickoff**

It is possible to add a new GameEvent to record kickoffs by:

- Creating a new Kickoff class that extends the GameEvent class
- Adding any new unique features for the item
- Modifying any other classes that need to know about this new class

# Game Record Including Kickoff

| The Magpies vs. The Sparrows (2 - 3) |              |                    |      |
|--------------------------------------|--------------|--------------------|------|
| Event                                | Team         | Player             | Time |
| Kickoff                              | The Sparrows | Dorothy Parker     | 0    |
| Possession                           | The Sparrows | Jane Austin        | 15   |
| Possession                           | The Sparrows | J. M. Synge        | 19   |
| Possession                           | The Sparrows | Brendan Behan      | 20   |
| Possession                           | The Sparrows | Dorothy Parker     | 26   |
| GOAL!                                | The Sparrows | Dorothy Parker     | 32   |
| Kickoff                              | The Magpies  | G. K. Chesterton   | 34   |
| Possession                           | The Magpies  | Oscar Wilde        | 35   |
| Possession                           | The Magpies  | G. K. Chesterton   | 41   |
| GOAL!                                | The Magpies  | G. K. Chesterton   | 43   |
| Kickoff                              | The Sparrows | Dorothy Parker     | 50   |
| Possession                           | The Sparrows | J. M. Synge        | 54   |
| GOAL!                                | The Sparrows | J. M. Synge        | 55   |
| Kickoff                              | The Magpies  | Wilkie Collins     | 59   |
| Possession                           | The Magpies  | G. K. Chesterton   | 62   |
| Possession                           | The Magpies  | Arthur Conan Doyle | 63   |
| Possession                           | The Magpies  | Oscar Wilde        | 64   |
| GOAL!                                | The Magpies  | Oscar Wilde        | 74   |
| Kickoff                              | The Sparrows | Frank O'Connor     | 75   |
| Possession                           | The Sparrows | Frank O'Connor     | 81   |
| GOAL!                                | The Sparrows | Frank O'Connor     | 83   |

The new event,  
Kickoff, has  
been added.

# Summary

In this lesson, you should have learned how to:

- Deploy a simple application as a JAR file
- Describe the parts of a Java application, including the user interface and the back end
- Describe how classes can be extended to implement new capabilities in the application



# Course Summary

In this course, you should have learned how to:

- List and describe several key features of the Java technology: object-oriented, multithreaded, distributed, simple, and secure
- Identify different Java technology groups
- Describe examples of how Java is used in applications as well as in consumer products
- Describe the benefits of using an integrated development environment (IDE)
- Develop classes and describe how to declare a class
- Analyze a business problem to recognize objects and operations that form the building blocks of the Java program design

# Course Summary

- Define the term *object* and its relationship to a class
- Demonstrate Java programming syntax
- Write a simple Java program that compiles and runs successfully
- Declare and initialize variables
- List several primitive data types
- Instantiate an object and effectively use object reference variables
- Use operators, loops, and decision constructs
- Declare and instantiate arrays and ArrayLists and be able to iterate through them

# Course Summary

- Use Javadocs to look up Java foundation classes
- Declare a method with arguments and return values
- Use inheritance to declare and define a subclass of an existing superclass
- Describe how errors are handled in a Java program
- Describe how to deploy a simple Java application by using the NetBeans IDE

# Building Microservices with Spring Boot

- Setting up the latest Spring development environment
- Developing RESTful services using the Spring framework
- Using Spring Boot to build fully qualified microservices
- Useful Spring Boot features to build production-ready microservices

# Setting up a Development Environment

- **JDK 1.8:** <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- **Spring Tool Suite 4 (STS):** <https://spring.io/tools/sts/all>
- **Maven 3.3.1:** <https://maven.apache.org/download.cgi>

We are doing this class based on the following versions of Spring libraries:

- Spring Framework `4.2.6.RELEASE`
- Spring Boot `2.1.6.RELEASE`

# Build a Legacy Rest Application with Spring (optional)

Please complete LAB 1 : <https://jmp.sh/2xiRgOF>

# Legacy to Microservices

- Carefully examining the preceding RESTful service will reveal whether this really constitutes a microservice.
- At first glance, the preceding RESTful service is a fully qualified interoperable REST/JSON service.
- However, it is not fully autonomous in nature.
  - This is primarily because the service relies on an underlying application server or web container.
- This is a traditional approach to developing RESTful services as a web application. However, from the microservices point of view, one needs a mechanism to develop services as executables, self-contained JAR files with an embedded HTTP listener.
  - Spring Boot is a tool that allows easy development of such kinds of services. Dropwizard and WildFly Swarm are alternate server-less RESTful stacks.

# Use Spring Boot to Build Microservices

- Spring Boot is a utility framework from the Spring team to bootstrap Spring-based applications and microservices quickly and easily.
  - The framework uses an opinionated approach over configurations for decision making, thereby reducing the effort required in writing a lot of boilerplate code and configurations.
- Using the 80-20 principle, developers should be able to kickstart a variety of Spring applications with many default values.
  - Spring Boot further presents opportunities for the developers to customize applications by overriding the autoconfigured values.

# Use Spring Boot to Build Microservices

```
<dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

 <groupId>org.hsqldb</groupId>

 <artifactId>hsqldb</artifactId>

 <scope>runtime</scope>

</dependency>
```

# Let's Get Started with Spring Boot

- Using the Spring Boot CLI as a command-line tool
- Using IDEs such as STS to provide Spring Boot, which are supported out of the box
- Using the Spring Initializr project at <http://start.spring.io>

# CLI

- The easiest way to develop and demonstrate Spring Boot's capabilities is using the Spring Boot CLI, a command-line tool.
- Complete Lab 2: <https://jmp.sh/YW2fGnV>

# Lab 3 - Create a Spring Boot Java Microservice with STS

Lab 3 : <https://jmp.sh/PHm3TQX>

# POM File

```
<parent>
```

```
 <groupId>org.springframework.boot</groupId>
```

```
 <artifactId>spring-boot-starter-parent</artifactId>
```

```
 <version>1.3.4.RELEASE</version>
```

```
</parent>
```

# POM File Properties

```
<spring-boot.version>2.1.6.BUILD-SNAPSHOT</spring-boot.version>

<hibernate.version>4.3.11.Final</hibernate.version>

<jackson.version>2.6.6</jackson.version>

<jersey.version>2.22.2</jersey.version>

<logback.version>1.1.7</logback.version>

<spring.version>4.2.6.RELEASE</spring.version>

<spring-data-releasetrain.version>Gosling-SR4</spring-data-releasetrain.version>

<tomcat.version>8.0.33</tomcat.version>
```

# More POM File Review

```
<dependencies>

 <dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-web</artifactId>

 </dependency>

 <dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-test</artifactId>

 <scope>test</scope>

 </dependency>

</dependencies>
```

# Java Version in the POM File

```
<java.version>1.8</java.version>
```

# Application.java

Spring Boot, by default, generated a `org.rvslab.session2.Application.java` class under `src/main/java` to bootstrap, as follows:

```
@SpringBootApplication

public class Application {

 public static void main(String[] args) {

 SpringApplication.run(Application.class, args);

 }

}
```

# More Application.java

@Configuration

@EnableAutoConfiguration

@ComponentScan

```
public class Application {
```

# application.properties

- A default `application.properties` file is placed under `src/main/resources`.
- It is an important file to configure any required properties for the Spring Boot application.

# ApplicationTests.java

- The last file to be examined is

`ApplicationTests.java` under `src/test/java`.

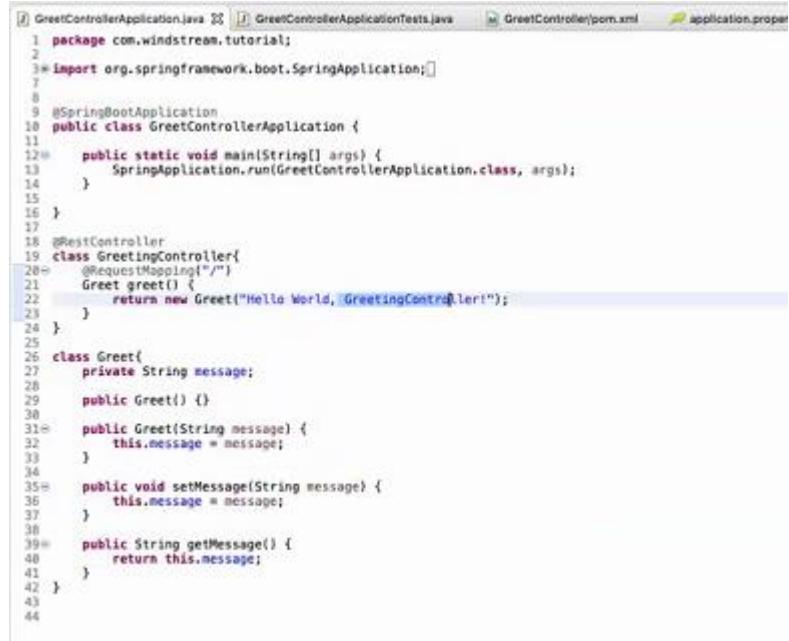
- This is a placeholder to write test cases against the Spring Boot application.

# Implement a RESTful Web Service : Lab

Lab 4 : <https://jmp.sh/kQbFFuR>



```
1 package com.windstream.tutorial;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.test.context.junit4.SpringRunner;
8 import org.springframework.web.client.RestTemplate;
9 import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
10 import org.springframework.boot.test.web.client.TestRestTemplate;
11 import org.springframework.boot.web.server.LocalServerPort;
12
13 import junit.framework.Assert;
14
15 @RunWith(SpringRunner.class)
16 @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
17 public class GreetControllerApplicationTests {
18
19 @Autowired
20 //private TestRestTemplate restTemplate;
21 @LocalServerPort
22 private int port;
23
24 @SuppressWarnings("deprecation")
25 @Test
26 public void contextLoads() {
27 RestTemplate restTemplate = new RestTemplate();
28 Greet greet = restTemplate.getForObject("http://localhost:" + port + "/", Greet.class);
29 Assert.assertEquals("Hello World!", greet.getMessage());
30 }
31
32 }
33
34
```



```
1 package com.windstream.tutorial;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class GreetControllerApplication {
7
8 public static void main(String[] args) {
9 SpringApplication.run(GreetControllerApplication.class, args);
10 }
11
12 }
13
14 @RestController
15 class GreetController{
16 @RequestMapping("/")
17 Greet greet() {
18 return new Greet("Hello World!,GreetingController");
19 }
20 }
21
22 class Greet{
23 private String message;
24
25 public Greet() {}
26
27 public Greet(String message) {
28 this.message = message;
29 }
30
31 public void setMessage(String message) {
32 this.message = message;
33 }
34
35 public String getMessage() {
36 return this.message;
37 }
38 }
39
40 }
```

# HATEOAS

- HATEOAS is a REST service pattern in which navigation links are provided as part of the payload metadata.
- The client application determines the state and follows the transition URLs provided as part of the state.
- This methodology is particularly useful in responsive mobile and web applications in which the client downloads additional data based on user navigation patterns.

## HATEOAS : LAB 5

<https://jmp.sh/n8QVINO>

# Momentum

- A number of basic Spring Boot examples have been reviewed so far.
- The rest of this section will examine some of the Spring Boot features that are important from a microservices development perspective.
- In the upcoming sections, we will take a look at how to work with dynamically configurable properties, change the default embedded web server, add security to the microservices, and implement cross-origin behavior when dealing with microservices.

# Spring Boot Configuration

- In this section, the focus will be on the configuration aspects of Spring Boot.
- The `session2.bootrest` project, already developed, will be modified in this section to showcase configuration capabilities.
- Copy and paste `session2.bootrest` and rename the project as `session2.boot-advanced`.

# Spring Boot autoconfiguration

- Spring Boot uses convention over configuration by scanning the dependent libraries available in the class path.
- For each `spring-boot-starter-*` dependency in the POM file, Spring Boot executes a default `AutoConfiguration` class. `AutoConfiguration` classes use the `*AutoConfiguration` lexical pattern, where `*` represents the library.
  - For example, the autoconfiguration of JPA repositories is done through `JpaRepositoriesAutoConfiguration`.
- Run the application with `--debug` to see the autoconfiguration report. The following command shows the autoconfiguration report for the `session2.boot-advanced` project:

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --debug
```

# Autoconfiguration Classes

- `ServerPropertiesAutoConfiguration`
- `RepositoryRestMvcAutoConfiguration`
- `JpaRepositoriesAutoConfiguration`
- `JmsAutoConfiguration`

You can exclude the autoconfiguration of a library - here is an example:

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```

# Overriding default config values

It is also possible to override default

configuration values using the

application.properties file.

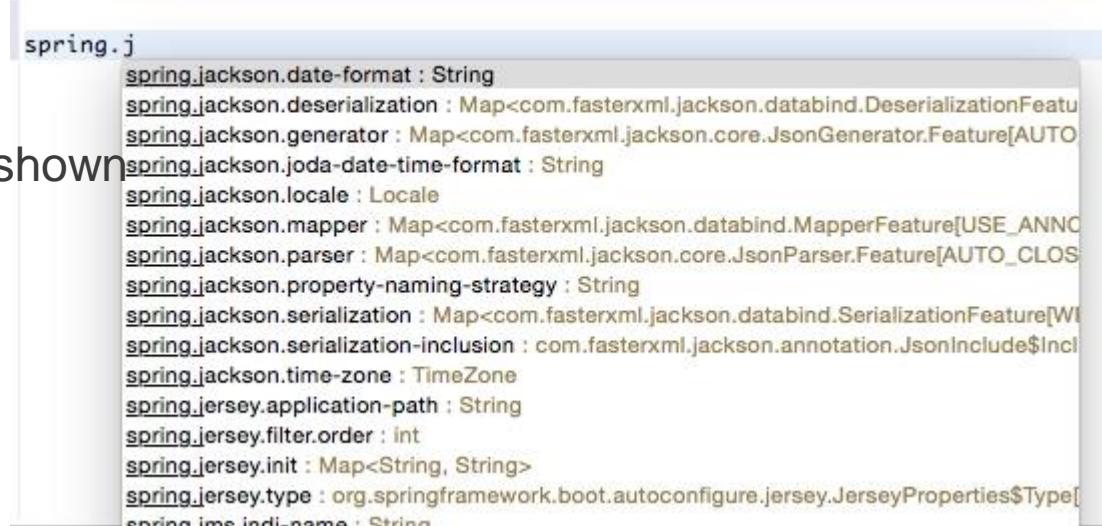
STS provides an easy-to-autocomplete,

server.port 9090

contextual help on

application.properties, as shown

in the following screenshot:



# Where is the config file?

Spring Boot externalizes all configurations into `application.properties`

```
spring.config.name= # config file name
```

```
spring.config.location= # location of config file
```

```
$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar --
spring.config.name=bootrest.properties
```

# Custom Property Files : Lab 6

- At startup, `SpringApplication` loads all the properties and adds them to the Spring `Environment` class.
- Add a custom property to the `application.properties` file.
- In this case, the custom property is named `bootrest.customproperty`.
- Autowire the Spring `Environment` class into the `GreetingController` class.
- Edit the `GreetingController` class to read the custom property from `Environment` and add a log statement to print the custom property to the console.

Lab 6 : <https://jmp.sh/illuGbD>

# Default Web Server

Embedded HTTP listeners can easily be customized as follows. By default, Spring Boot supports Tomcat, Jetty, and Undertow. Replace Tomcat is replaced with Undertow (This is Lab 6.5):

```
<dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-web</artifactId>

 <exclusions>
 <exclusion>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-tomcat</artifactId>
 </exclusion>
 </exclusions>

</dependency>

<dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-undertow</artifactId>
```

# Securing Microservices with basic security

- Adding basic authentication to Spring Boot is pretty simple. Add the following dependency to `pom.xml`. This will include the necessary Spring security library files:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

# Securing Microservices with basic security

Open `Application.java` and add `@EnableGlobalMethodSecurity` to the `Application` class.

This annotation will enable method-level security:

```
@EnableGlobalMethodSecurity
@SpringBootApplication
public class Application {
 public static void main(String[] args) {
 SpringApplication.run(Application.class, args);
 }
}
```

# Securing Microservices with basic security

The default basic authentication assumes the user as being `user`. The default password will be printed in the console at startup. Alternately, the username and password can be added in `application.properties`, as shown here:

```
security.user.name=guest
security.user.password=guest123
```

# Securing Microservices with basic security

```
@Test

public void testSecureService() {

 String plainCreds = "guest:guest123";

 HttpHeaders headers = new HttpHeaders();

 headers.add("Authorization", "Basic " + new
String(Base64.encode(plainCreds.getBytes())));

 HttpEntity<String> request = new HttpEntity<String>(headers);

 RestTemplate restTemplate = new RestTemplate();

 ResponseEntity<Greet> response = restTemplate.exchange("http://localhost:8080",
HttpMethod.GET, request, Greet.class);

 Assert.assertEquals("Hello World!", response.getBody().getMessage());
}
```

# Securing Microservices with basic security

As shown in the code, a new `Authorization` request header with Base64 encoding the username-password string is created.

Rerun the application using Maven. Note that the new test case passed, but the old test case failed with an exception. The earlier test case now runs without credentials, and as a result, the server rejected the request with the following message:

```
org.springframework.web.client.HttpClientErrorException: 401 Unauthorized
```

# Securing a Microservice with OAuth2

- When a client application requires access to a protected resource, the client sends a request to an authorization server.
- The authorization server validates the request and provides an access token.
- This access token is validated for every client-to-server request.
- The request and response sent back and forth depends on the grant type.

LAB 7 : OATH2 LAB : <https://jmp.sh/C212HLk>

# Enabling cross-origin access for Microservices

- Browsers are generally restricted when client-side web applications running from one origin request data from another origin. Enabling cross-origin access is generally termed as **CORS (Cross-Origin Resource Sharing)**.
- With microservices, as each service runs with its own origin, it will easily get into the issue of a client-side web application consuming data from multiple origins. For instance, a scenario where a browser client accessing Customer from the Customer microservice and Order History from the Order microservices is very common in the microservices world.
- Spring Boot provides a simple declarative approach to enabling cross-origin requests.

# Enabling cross-origin access for Microservices

- The following example shows how to enable a microservice to enable cross-origin requests:

```
@RestController
class GreetingController{
 @CrossOrigin
 @RequestMapping("/")
 Greet greet(){
 return new Greet("Hello World!");
 }
}
```

# Enabling cross-origin access for Microservices

- By default, all the origins and headers are accepted. We can further customize the cross-origin annotations by giving access to specific origins, as follows. The `@CrossOrigin` annotation enables a method or class to accept cross-origin requests:

```
@CrossOrigin("http://mytrustedorigin.com")
```

- Global CORS can be enabled using the `WebMvcConfigurer` bean and customizing the `addCorsMappings(CorsRegistry registry)` method.

# Implementing Spring Boot Messaging: Lab 8

Lab 8 : <https://jmp.sh/UGrAhXI>

# Developing a comprehensive microservice example: Lab 9

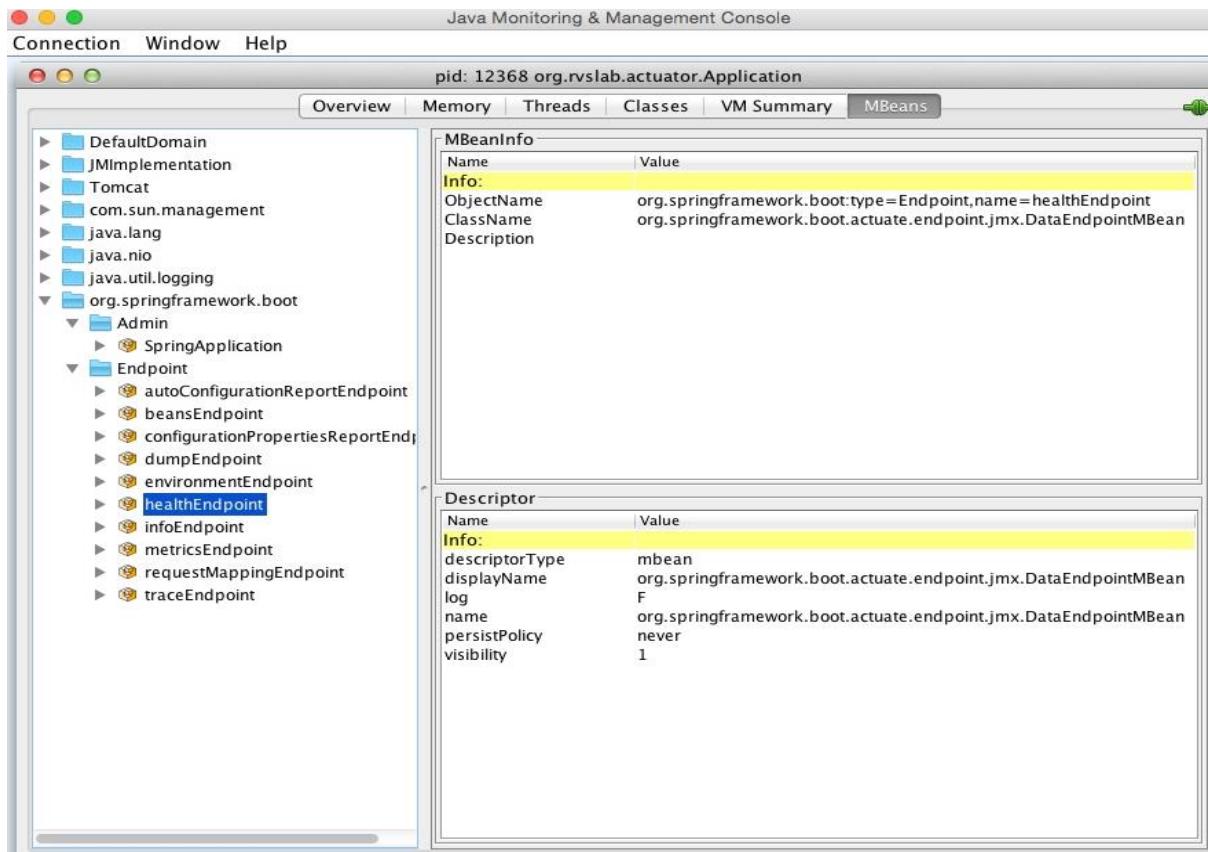
- So far, the examples we have considered are no more than just a simple "Hello world." Putting together what we have learned, this section demonstrates an end-to-end Customer Profile microservice implementation.
- The Customer Profile microservices will demonstrate interaction between different microservices.
- It also demonstrates microservices with business logic and primitive data stores.
- The Customer Profile microservice exposes methods to **create, read, update, and delete(CRUD)** a customer and a registration service to register a customer.
- The registration process applies certain business logic, saves the customer profile, and sends a message to the Customer Notification microservice.
- The Customer Notification microservice accepts the message sent by the registration service and sends an e-mail message to the customer using an SMTP server.
- Asynchronous messaging is used to integrate Customer Profile with the Customer Notification service.

Lab 9 : Complete End to End Microservice : <https://jmp.sh/eQfpQFC>

# Spring Boot Actuators

- Spring Boot actuators provide an excellent out-of-the-box mechanism to monitor and manage Spring Boot applications in production
- Lab 10 - <https://jmp.sh/wUXrzcs>

# Monitoring Using JConsole



# Monitoring Using SSH

Spring Boot provides remote access to the Boot application using SSH. The following command connects to the Spring Boot application from a terminal window:

```
$ ssh -p 2000 user@localhost
```

The password can be customized by adding the `shell.auth.simple.user.password` property in the `application.properties` file. The updated `application.properties` file will look similar to the following:

```
shell.auth.simple.user.password=admin
```

# Configuring Application Information

```
management.endpoints.web.exposure.include=*
info.app.name=Boot Actuator
info.app.description=My Greetings Service
info.app.version=1.0.0
#endpoints.app.name=Boot Actuator
```

```
class TPSCounter {
 LongAdder count;
 int threshold = 2;
 Calendar expiry = null;

 TPSCounter(){
 this.count = new LongAdder();
 this.expiry = Calendar.getInstance();
 this.expiry.add(Calendar.MINUTE, 1);
 }

 boolean isExpired(){
 return Calendar.getInstance().after(expiry);
 }

 boolean isWeak(){
 return (count.intValue() > threshold);
 }

 void increment(){
 count.increment();
 }
}
```

# Documenting Microservices

- The traditional approach of API documentation is either by writing service specification documents or using static service registries.
- With a large number of microservices, it would be hard to keep the documentation of APIs in sync.

```
<dependency>
 <groupId>io.springfox<...

```

# Summary