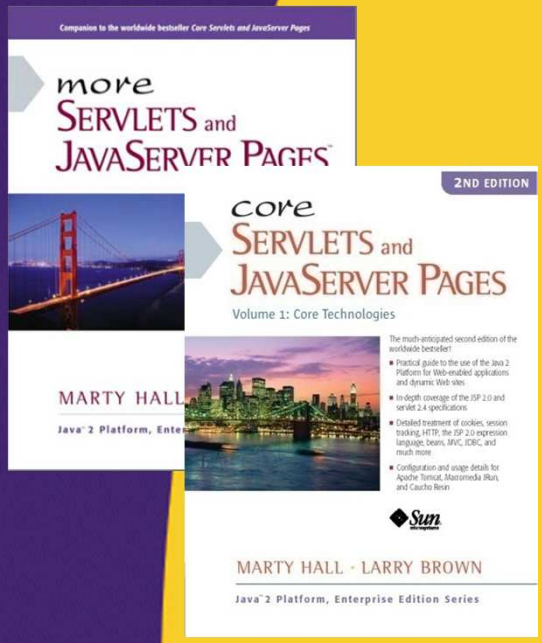
Course Material Usage Rules

- **PowerPoint slides for use only in full-semester, for-credit courses at degree-granting institutions**
  - Slides *not* permitted for use in commercial training courses except when taught by coreservlets.com (see http://courses.coreservlets.com).
- **Slides can be modified by instructor**
  - Please retain this notice and attribution to coreservlets.com
- **Instructor can give PDF or hardcopy to students, but should protect PowerPoint files**
  - *This slide is suppressed in Slide Show mode*

Object-Oriented Programming in Java: More Capabilities

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/java5.html



For live Java EE training, please see training courses at http://courses.coreservlets.com/.

Servlets, JSP, Struts, JSF 1.*x*, JSF 2.0, Ajax (with jQuery, Dojo, Prototype, Ext-JS, Google Closure, etc.), GWT 2.0 (with GXT), Java 5, Java 6, SOAP-based and RESTful Web Services, Spring, Hibernate/JPA, and customized combinations of topics.

Taught by the author of *Core Servlets and JSP*, *More Servlets and JSP*, and this tutorial. Available at public venues, or customized versions can be held on-site at your organization. Contact hall@coreservlets.com for details.

Topics in This Section

- **Overloading**
- **Best practices for "real" classes**
  - Encapsulation and accessor methods
  - JavaDoc
- **Inheritance**
- **Advanced topics**
  - Abstract classes
  - Interfaces
  - CLASSPATH
  - Packages
  - Visibility modifiers
  - JavaDoc options

Overloading

Overview

- **Idea**
  - Classes can have more than one method with the same name, or more than one constructor.
  - The methods (or constructors) have to differ from each other by having different number or types of arguments
- **Syntax**

```
public class MyClass {
    public double getRandomNum() { …};   // Range 1-10
    public double getRandomNum(double range) { … }
}
```

- **Motivation**
  - Methods: lets you have similar names for similar ops
  - Constructors: let you build instances in different ways

Ship Example: Overloading

(In Ship4.java)

```
public class Ship4 {
  public double x=0.0, y=0.0, speed=1.0, direction=0.0;
  public String name;

  public Ship4(double x, double y,
               double speed, double direction,
               String name) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.direction = direction;
    this.name = name;
  }

  public Ship4(String name) {
    this.name = name;
  }

  private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
  }
  ...
```

Overloading (Continued)

```
  ...

  public void move() {
    move(1);
  }

  public void move(int steps) {
    double angle = degreesToRadians(direction);
    x = x + steps * speed * Math.cos(angle);
    y = y + steps * speed * Math.sin(angle);
  }

  public void printLocation() {
    System.out.println(name + " is at ("
                       + x + "," + y + ").");
  }
}
```

Overloading: Testing and Results

(In Test4.java)

```
public class Test4 {
  public static void main(String[] args) {
    Ship4 s1 = new Ship4("Ship1");
    Ship4 s2 = new Ship4(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move(3);
    s1.printLocation();
    s2.printLocation();
  }
}
```

- **Compiling and Running**

```
> javac Test4.java
> java Test4
```

- **Output:**

```
Ship1 is at (1.0,0.0).
Ship2 is at (-4.24264...,4.24264...).
```

Overloading: Major Points

- **Idea**
  - Allows you to define more than one function or constructor with the same name
    - Overloaded functions or constructors must differ in the number or types of their arguments (or both), so that Java can always tell which one you mean
- **Simple examples:**
  - Here are two `square` methods that differ only in the type of the argument; they would both be permitted inside the same class definition.

```
// square(4) is 16
public int square(int x) { return(x*x); }

// square("four") is "four four"
public String square(String s) {
  return(s + " " + s);
}
```

OOP Design: Best Practices

Overview

- **Ideas**
  - Instance variables should always be private
    - And hooked to outside world with getBlah and/or setBlah
  - From very beginning, put in JavaDoc-style comments
- **Syntax**

```
/** Short summary. More detail. Can use HTML. */
public class MyClass {
    private String firstName;
    public String getFirstName() { return(firstName); }
    public void setFirstName(String s) { firstName = s; }
}
```

- **Motivation**
  - Limits ripple effect. Makes code more maintainable.

Ship Example: OOP Design and Usage

```
/** Ship example to demonstrate OOP in Java. */

public class Ship {
  private double x=0.0, y=0.0, speed=1.0, direction=0.0;
  private String name;
  …
  /** Get current X location. */

  public double getX() {
    return(x);
  }

  /** Set current X location. */

  public void setX(double x) {
    this.x = x;
  }
  ...
```

OOP Design: Testing and Results

(In ShipTest.java)

```
public class ShipTest {
  public static void main(String[] args) {
    Ship s1 = new Ship("Ship1");
    Ship s2 = new Ship(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move(3);
    s1.printLocation();
    s2.printLocation();
  }
}
```

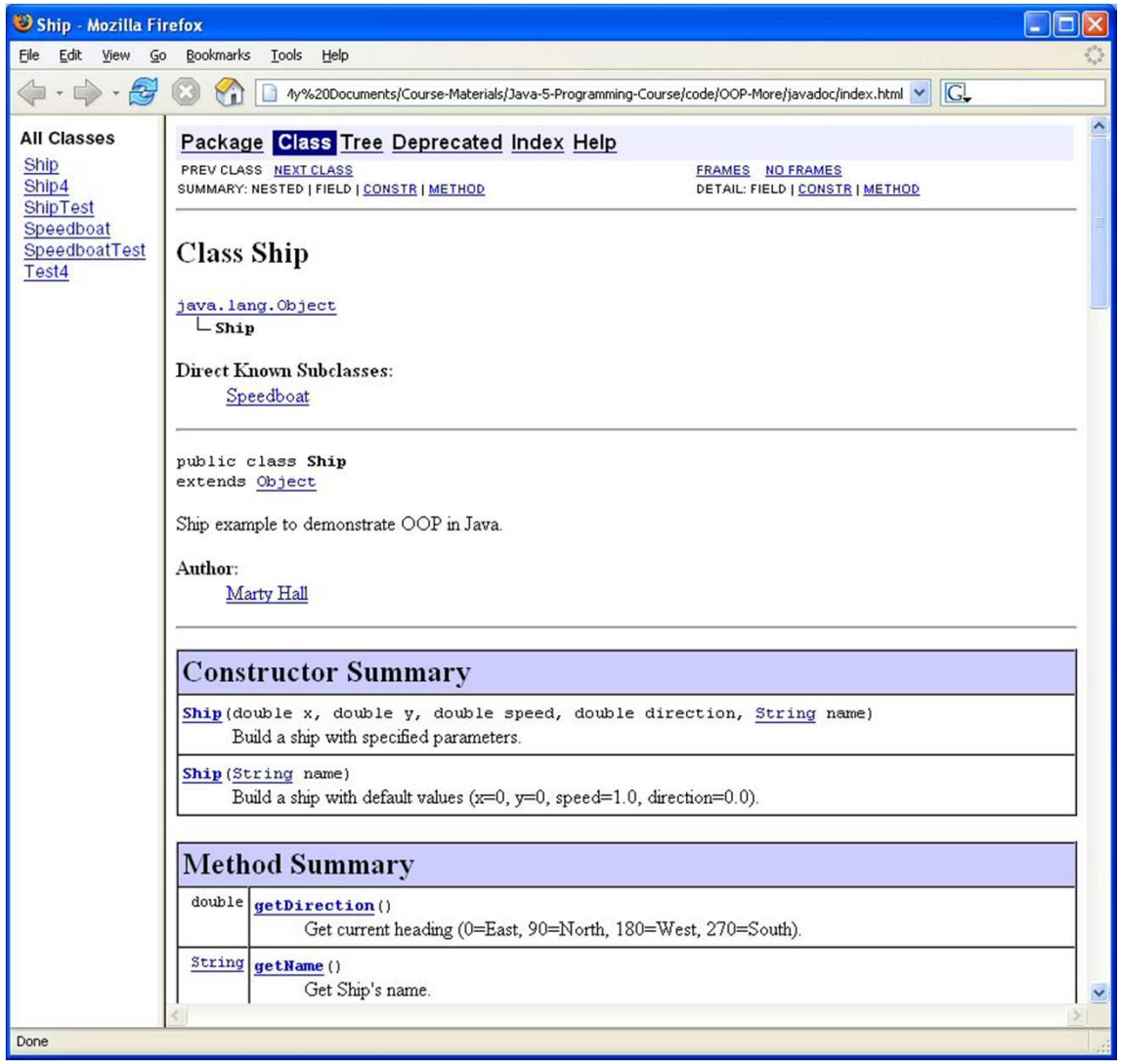
- **Compiling and Running**

```
>javac ShipTest.java
>java ShipTest
>javadoc *.java
```

- **Output:**

```
Ship1 is at (1.0,0.0).
Ship2 is at (-4.24264...,4.24264...).
```

OOP Design: Testing and Results (Continued)



Major Points

- **Encapsulation**
  - Lets you change internal representation and data structures *without users of your class changing their code*
  - Lets you put constraints on values *without users of your class changing their code*
  - Lets you perform arbitrary side effects *without users of your class changing their code*
- **Comments and JavaDoc**
  - Comments marked with /** ... */ will be part of the online documentation
  - Call "javadoc *.java" to build online documentation.
  - See later slides for details

More Details on Getters and Setters

- **There need not be both getters and setters**
  - It is common to have fields that can be set at instantiation, but never changed again (immutable field). It is even quite common to have classes containing only immutable fields (immutable classes)

```
public class Ship {
    private final String shipName;

    public Ship(…) {  shipName = …; … }

    public String getName() { return(shipName); }

    // No setName method
}
```

More Details on Getters and Setters

- **Getter/setter names need not correspond to instance variable names**
  - Common to do so if there is a simple correspondence, but this is not required
    - Notice on previous page that instance var was "shipName", but methods were "getName" and "setName"
  - In fact, there doesn't even have to *be* a corresponding instance variable

```
public class Customer {
    …
    public String getFirstName() { getFromDatabase(…); }
    public void setFirstName(…) { storeInDatabase(…); }
    public double getBonus() { return(Math.random()); }
}
```

Inheritance

Overview

- **Ideas**
  - You can make a class that "inherits" characteristics of another class
    - The original class is called "parent class", "super class", or "base class". The new class is called "child class", "subclass", or "extended class".
  - The child class has access to all non-private methods of the parent class.
    - No special syntax need to call inherited methods
- **Syntax**
  - public class ChildClass extends ParentClass { … }
- **Motivation**
  - Supports the key OOP idea of code reuse (i.e., don't write the same code twice). Design class hierarchies so that shared behavior is in inherited to all classes that need it.

Simple Example

- **Person**

```
public class Person {
  public String getFirstName() { … }
  public String getLastName() { … }
}
```

- **Employee**

```
public class Employee extends Person {
  public double getSalary() { … }

  public String getEmployeeInfo() {
    return(getFirstName() + " " + getLastName() +
           " earns " + getSalary());
  }
}
```

Ship Example: Inheritance

```
public class Speedboat extends Ship {
  private String color = "red";

  public Speedboat(String name) {
    super(name);
    setSpeed(20);
  }

  public Speedboat(double x, double y,
                   double speed, double direction,
                   String name, String color) {
    super(x, y, speed, direction, name);
    setColor(color);
  }

  @Override // Optional -- discussed later
  public void printLocation() {
    System.out.print(getColor().toUpperCase() + " ");
    super.printLocation();
  }
  ...
}
```

Inheritance Example: Testing

```
public class SpeedboatTest {
  public static void main(String[] args) {
    Speedboat s1 = new Speedboat("Speedboat1");
    Speedboat s2 = new Speedboat(0.0, 0.0, 2.0, 135.0,
                                 "Speedboat2", "blue");
    Ship s3 = new Ship(0.0, 0.0, 2.0, 135.0, "Ship1");
    s1.move();
    s2.move();
    s3.move();
    s1.printLocation();
    s2.printLocation();
    s3.printLocation();
  }
}
```

Inheritance Example: Result

- **Compiling and running manually**

```
> javac SpeedboatTest.java
```

  - The above calls javac on **`Speedboat.java`** and **`Ship.java`** automatically

```
> java SpeedboatTest
```

- **Output**

```
RED Speedboat1 is at (20,0).
BLUE Speedboat2 is at (-1.41421,1.41421).
Ship1 is at (-1.41421,1.41421).
```

Ship Inheritance Example: Major Points

- **Format for defining subclasses**
- **Using inherited methods**
- **Using super(…) for inherited constructors**
  - *Only* when the zero-arg constructor is not OK
- **Using super.someMethod(…) for inherited methods**
  - *Only* when there is a name conflict

Inheritance

- **Syntax for defining subclasses**

```
public class NewClass extends OldClass {
  ...
}
```

- **Nomenclature:**
  - The old class is called the superclass, base class or parent class
  - The new class is called the subclass, derived class or child class
- **Effect of inheritance**
  - Subclasses automatically have all public fields and methods of the parent class
  - You don't need any special syntax to access the inherited fields and methods; you use the exact same syntax as with locally defined fields or methods.
  - You can also add in fields or methods not available in the superclass
- **Java doesn't support multiple inheritance**
  - A class can only have one *direct* parent. But grandparent and great-grandparent (etc.) are legal and common.

Inherited constructors and super(...)

- **When you instantiate an object of a subclass, the system will automatically call the superclass constructor first**
  - By default, the zero-argument superclass constructor is called
  - If you want to specify that a different parent constructor is called, invoke the parent class constructor with `super(`*`args`*`)`
  - If `super(…)` is used in a subclass constructor, then `super(…)` must be the first statement in the constructor
- **Constructor life-cycle**
  - Each constructor has three phases:
    1. Invoke the constructor of the superclass
       - The zero-argument constructor is called automatically. No special syntax is needed unless you want a *different* parent constructor.
    2. Initialize all instance variables based on their initialization statements
    3. Execute the body of the constructor

Overridden methods and super.method(...)

- **When a class defines a method using the same name, return type, and arguments as a method in the superclass, then the class *overrides* the method in the superclass**
  - Only non-static methods can be overridden

- **If there is a locally defined method and an inherited method that have the same name and take the same arguments, you can use the following to refer to the inherited method**

  `super.methodName(...)`

  - Successive use of `super` (super.super.methodName) is not legal.

Example: Person Class

Iterations of Person

- **Last lecture: four iterations of Person**
  - Instance variables
  - Methods
  - Constructors
  - Constructors with "this" variable
- **This lecture**
  - Person class
    - Change instance vars to private, add accessor methods
    - Add JavaDoc comments
  - Employee class
    - Make a class based on Person that has all of the information of a Person, plus new data

Person Class (Part 1)

```
/** A class that represents a person's given name
 *  and family name.
 */
public class Person {
  private String firstName, lastName;

  public Person(String firstName,
                String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
```

Person Class (Part 2)

```
/** The person's given (first) name. */

public String getFirstName() {
  return (firstName);
}

public void setFirstName(String firstName) {
  this.firstName = firstName;
}
```

Person Class (Part 3)

```
  /** The person's family name (i.e.,
   *  last name or surname).
   */
  public String getLastName() {
    return (lastName);
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  /** The person's given name and family name, printed
   *  in American style, with given name first and
   *  a space in between.
   */
  public String getFullName() {
    return(firstName + " " + lastName);
  }
}
```

Employee Class (Part 1)

```
/** Represents people that work at a company. */

public class Employee extends Person {
  private int employeeId;
  private String companyName;

  public Employee(String firstName, String lastName,
                  int employeeId, String companyName) {
    super(firstName, lastName);
    this.employeeId = employeeId;
    this.companyName = companyName;
  }
```

Employee Class (Part 2)

```
/** The ID of the employee, with the assumption that
 *  lower numbers are people that started working at
 *  the company earlier than those with higher ids.
 */
 public int getEmployeeId() {
   return (employeeId);
 }

 public void setEmployeeId(int employeeId) {
   this.employeeId = employeeId;
 }
```

Employee Class (Part 3)

```
  /** The name of the company that the person
   *  works for.
   */
  public String getCompanyName() {
    return (companyName);
  }

  public void setCompanyName(String companyName) {
    this.companyName = companyName;
  }
}
```

Advanced Topics

Advanced OOP Topics

- Abstract classes
- Interfaces
- Using @Override
- CLASSPATH
- Packages
- Visibility other than public or private
- JavaDoc details

Abstract Classes

- **Idea**
  - A class that you cannot directly instantiate (i.e., on which you cannot use "new"). But you can subclass it and instantiate the subclasses.
- **Syntax**

```
public abstract class SomeClass {
  public abstract SomeType method1(…);  // No body
  public SomeType method2(…) { … } // Not abstract
}
```

- **Motivation**
  - Guarantees that all subclasses will have certain methods
  - Lets you make collections of mixed types

Abstract Classes: Problem

- **You have**
  - Circle and Rectangle classes, each with getArea methods
- **Goal**
  - Get sum of areas of an array of Circles and Rectangles

- **Why does this fail?**

```
Object[] shapes =
  { new Circle(…), new Rectangle(…) … };
double sum = 0;
for(Object shape: shapes) {
  sum = sum + shape.getArea();
}
```

Abstract Classes: Solution

- **Shape**

```
public abstract class Shape {
  public abstract double getArea();

  public void printInfo() {
    System.out.println(getClass().getSimpleName() +
                       " has area " + getArea());
  }
}
```

- **Circle (and similar for Rectangle)**

```
public class Circle extends Shape {
  public double getArea() { … }
}
```

Interfaces

- **Idea**
  - A model for a class. More or less an abstract class but without any concrete methods.
- **Syntax**

```
public interface SomeInterface {
   public SomeType method1(…);  // No body
   public SomeType method2(…);  // No body
}
public class SomeClass implements SomeInterface {
  // Real definitions of method1 and method 2
}
```

- **Motivation**
  - Like abstract classes, guarantees classes have certain methods. But classes can implement multiple interfaces.

Interfaces: Problem

- **Sum of areas**
  - You again want to get sum of areas of mixture of Circles and Rectangles.
    - But, this time you do not need Shape “class” to have a concrete printInfo method
- **Why interface instead of abstract class?**
  - Classes can directly extend only one class (abstract or otherwise)
  - Classes can implement many interfaces

```
public class Foo extends Bar implements Baz, Boo {
…
}
```

Interfaces: Solution

- **Shape**

```
public interface Shape {
  public double getArea();
}
```

- **Circle**

```
public class Circle implements Shape {
  public double getArea(…) { … }
}
```

- **Rectangle**

```
public class Rectangle implements Shape {
  public double getArea() { … }
}
```

Using @Override

- **Parent class**

```
public class Ellipse implements Shape {
    public double getArea() { … }
}
```

If Ellipse does not properly define getArea, code won't even compile since then the class does not satisfy the requirements of the interface.

- **Child class (mistake!)**

```
public class Circle extends Ellipse {
    public double getarea() { … }
}
```

This code will compile, but when you call getArea at runtime, you will get version from Ellipse, since there was a typo in this name.

- **Catching mistake at compile time**

```
public class Circle extends Ellipse {
    @Override
    public double getarea() { … }
}
```

This tells the compiler "I think that I am overriding a method from the parent class". If there is no such method in the parent class, code won't compile. If there is such a method in the parent class, then @Override has no effect on the code. Recommended but optional. More on @Override in later sections.

CLASSPATH

- **Idea**
  - The CLASSPATH environment variable defines a list of directories in which to look for classes
    - Default = current directory and system libraries
    - Best practice is to not set this when first learning Java!
- **Setting the CLASSPATH**

```
set CLASSPATH = .;C:\java;D:\cwp\echoserver.jar
setenv CLASSPATH .:~/java:/home/cwp/classes/
```

  - The "." indicates the current working directory
- **Supplying a CLASSPATH**

```
javac -classpath .;D:\cwp  WebClient.java
java -classpath .;D:\cwp WebClient
```

Packages

- **Idea**
  - Organize classes in groups.
- **Syntax**
  - To put your code in package
    - Make folder called “somePackage”
    - put “package somePackage” at top of file
  - To use code from another package
    - put “import somePackage.*” in file below your package statement
- **Motivation**
  - You only have to worry about name conflicts within your package.
    - So, team members can work on different parts of project without worrying about what class names the other teams use.

Visibility Modifiers

- **public**
  - This modifier indicates that the variable or method can be accessed anywhere an instance of the class is accessible
  - A class may also be designated `public`, which means that any other class can use the class definition
  - The name of a public class must match the filename, thus a file can have only one public class
- **private**
  - A `private` variable or method is only accessible from methods within the same class
  - Declare *all* instance variables private
  - Declare methods private if they are not part of class contract and are just internal implementation helpers

Visibility Modifiers (Continued)

- **protected**
  - Protected variables or methods can only be accessed by methods within the class, within classes in the same package, and within subclasses
- **[default]**
  - Default visibility indicates that the variable or method can be accessed by methods within the class, and within classes in the same package
  - A variable or method has default visibility if a modifier is omitted . Rarely used!
    - private: very common. Use this as first choice.
    - public: common for methods and constructors. 2nd choice
    - protected: usually for instance vars only. Moderately rare.
    - default: very rare. Don't omit modifier without good reason.

Visibility Summary

Data Fields and Methods	Modifiers			
	public	**protected**	**default**	**private**
Accessible from same class?	yes	yes	yes	yes
Accessible to classes (**nonsubclass**) from the **same package**?	yes	yes	yes	no
Accessible to **subclass** from the **same package**?	yes	yes	yes	no
Accessible to classes (**nonsubclass**) from **different package**?	yes	no	no	no
Accessible to **subclasses** from **different package**?	yes	no	no	no
Inherited by subclass in the same package?	yes	yes	yes	no
Inherited by subclass in different package?	yes	yes	no	no

Other Modifiers

- **final**
  - For a variable: cannot be changed after instantiation
    - Widely used to make “immutable” classes
  - For a class: cannot be subclassed
  - For a method: cannot be overridden in subclasses
- **synchronized**
  - Sets a lock on a section of code or method
    - Only one thread can access the code at any given time
- **volatile**
  - Guarantees other threads see changes to variable
- **transient**
  - Variables are not stored in serialized objects
- **native**
  - Indicates that the method is implement using C or C++

Comments and JavaDoc

- **Java supports 3 types of comments**
  - // Comment to end of line.
  - /* Block comment containing multiple lines.
    Nesting of comments in not permitted. */
  - /** A JavaDoc comment placed before class definition and nonprivate methods.
    Text may contain (most) HTML tags, hyperlinks, and JavaDoc tags. */
- **JavaDoc**
  - Used to generate on-line documentation

```
javadoc Foo.java Bar.java (or *.java)
```

  - JavaDoc Home Page
    - http://java.sun.com/javase/6/docs/technotes/tools/windows/javadoc.html

Useful JavaDoc Tags

- **@author**
  - Specifies the author of the document
  - Must use **`javadoc -author ...`** to generate in output

```
/** Description of some class ...
 *
 * @author <A HREF="mailto:hall@coreservlets.com">
 *         Marty Hall</A>
 */
```

- **@version**
  - Version number of the document
  - Must use **`javadoc -version ...`** to generate in output
- **@param**
  - Documents a method argument
- **@return**
  - Documents the return type of a method

Useful JavaDoc Command-line Arguments

- **-author**
  - Includes author information (omitted by default)
- **-version**
  - Includes version number (omitted by default)
- **-noindex**
  - Tells `javadoc` not to generate a complete index
- **-notree**
  - Tells `javadoc` not to generate the tree.html class hierarchy
- **-link, -linkoffline**
  - Tells `javadoc` where to look to resolve links to other packages

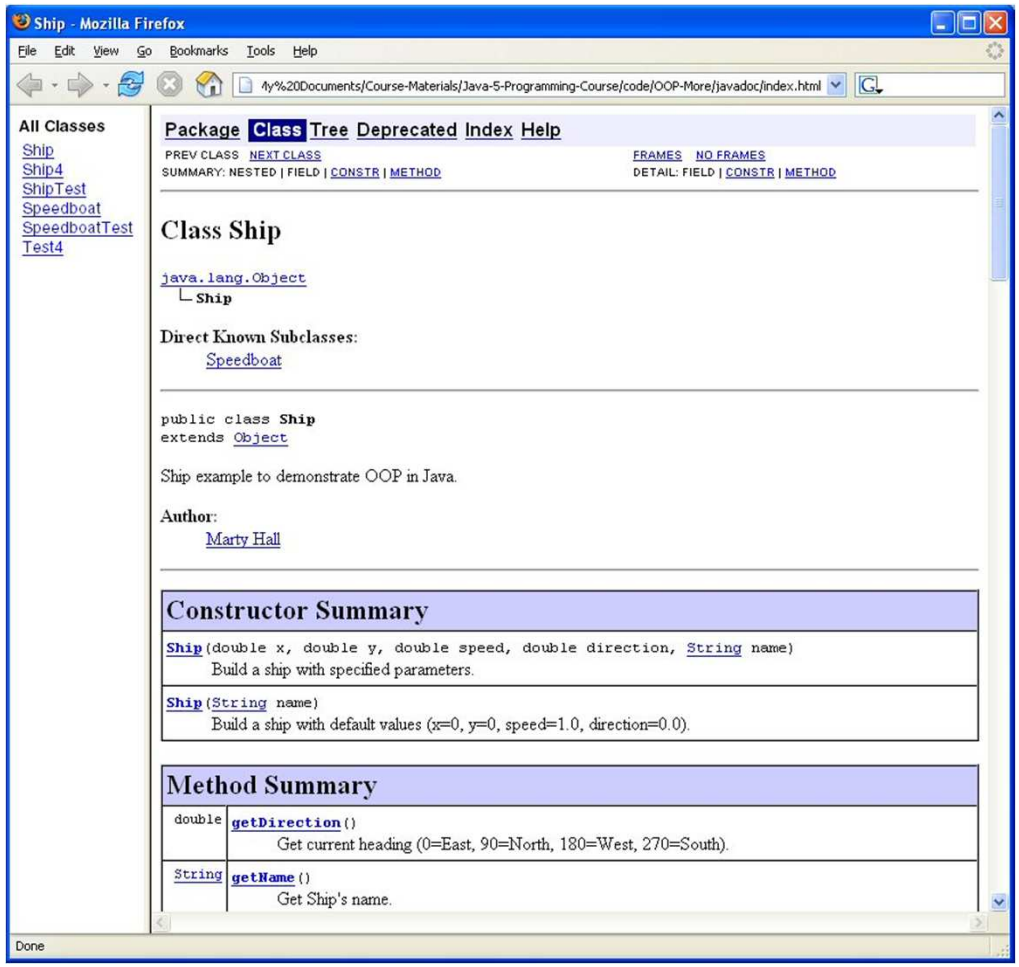
```
-link http://java.sun.com/j2se/1.5.0/docs/api/
-linkoffline http://java.sun.com/j2se/1.5.0/docs/api/
             c:\jdk1.5\docs\api
```

JavaDoc: Example

```
/** Ship example to demonstrate OOP in Java.
 *
 * @author <a href="mailto:hall@coreservlets.com">
 *         Marty Hall</a>
 */

public class Ship {
  private double x=0.0, y=0.0, speed=1.0, direction=0.0;
  private String name;

  /** Build a ship with specified parameters. */

  public Ship(double x, double y, double speed,
              double direction, String name) {
    setX(x);
    setY(y);
    setSpeed(speed);
    setDirection(direction);
    setName(name);
  }
  ...
```

JavaDoc: Example

```
> javadoc -link http://java.sun.com/j2se/1.5.0/docs/api/
          -author *.java
```



Wrap-Up

Java OOP References

- **Online**
  - "Basics" section of Sun Java Tutorial
    - http://java.sun.com/docs/books/tutorial/java/
- **Books**
  - *Murach's Java SE 6* (Murach, Steelman, and Lowe)
    - Excellent Java intro for beginners to Java (but not first-time programmers). Very good OOP section.
  - *Thinking in Java* (Bruce Eckel)
    - Perhaps not quite as good as Murach's book in general, but possibly the best OOP coverage of any Java book.
  - *Effective Java, 2nd Edition* (Josh Bloch)
    - In my opinion, the best Java book ever written. Fantastic coverage of OOP best practices.
      - However, very advanced. Other than the OOP chapter, you won't understand much unless you have been doing Java fulltime for at least a year. Even experts will learn a lot from this book.

Summary

- **Overloading**
  - You can have multiple methods or constructors with the same name. They must differ in argument signatures (number and/or type of arguments).
- **Best practices**
  - Make *all* instance variables private
  - Hook them to the outside with getBlah and/or setBlah
  - Use JavaDoc-style comments from the very beginning
- **Inheritance**
  - public class Subclass extends Superclass { … }
    - Non-private methods available with no special syntax
    - Use super() on first line of constructor if you need non-default parent constructor (moderately common)
    - Use super.method(…) if local method and inherited method have the same name (rare!)