Object-Oriented Programming in Java: More Capabilities

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

Customized Java EE Training: http://courses.coreservlets.com/
Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

For customized Java-related training at your organization, email hall@coreservlets.com

Marty is also available for consulting and development support

Taught by lead author of *Core Servlets & JSP*, co-author of *Core JSF* (4th Ed), & this tutorial. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - JSF 2.2, PrimeFaces, servlets/JSP, Ajax, JavaScript, jQuery, Android, Java 7 or 8 programming, GWT, custom mix of topics
  - Courses available in any state or country. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring MVC, Core Spring, Hibernate/JPA, Hadoop, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Overloading**
- **Best practices for "real" classes**
  - Encapsulation and accessor methods
  - JavaDoc
- **Inheritance**
- **Packages**
- **The toString method**
- **More iterations of the Person class**

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 example**

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


Motivation

- **Overloading methods**
  - Lets you have similar names for similar operations
    - MathUtils.arraySum(arrayOfInts)
    - MathUtils.arraySum(arrayOfDoubles)
    - MathUtils.log(number) // Assumes $\log_e$(number)
    - MathUtils.log(number, base) // $\log_{base}$(number)
- **Overloading constructors**
  - Lets you build instances in different ways
    - new Ship(someName) // Default x, y, speed, direction
    - new Ship(someX, someY, someSpeed, someDirection, someName)

Ship Example: Overloading (ship4/Ship.java)

```
package ship4;

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

  public Ship(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 Ship(String name) {
    this.name = name;
  }
  ...
```


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);
  }

  // degreesToRadians and printLocation as before
}
```

Ship Tester

```
package ship4;

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();
  }
}
```


Overloading: Results

- **Compiling and running in Eclipse (common)**
  - Save Ship.java and ShipTest.java
  - R-click inside ShipTest.java, Run As → Java Application
- **Compiling and running manually (rare)**

```
> javac ship4\ShipTest.java
> java ship4.ShipTest
```

- **Output:**

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

OOP Design: Best Practices

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."
– John F. Woods

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 example**

```
/** 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

- **Supports secondary goal of OOP**
  - Limits ripple effect, where changes to one class requires changes to the classes that use it, that require changes to the classes that use that, and so forth.
    - Lets you make changes to internal representation of classes without changing its public interface.
    - Makes code more maintainable.

OOP Principles

- **Basic OOP principles**
  - Primary goal: avoid needing to repeat identical or almost-identical code
    - DRY: Don't Repeat Yourself
    - Code reuse
  - Secondary goal: limit ripple effect
- **Advanced OOP principles**
  - SOLID
    - http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29
    - http://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/

Ship Example: OOP Design and Usage (ship5/Ship.java)

```
/** 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;
  }
  ...
```


Ship Tester

```
package ship5;

/** Small example to test the Ship class.
 *  <p>
 *  From <a href="http://www.coreservlets.com/">the
 *  coreservlets.com Java tutorials</a>.
 */
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();
 }
}
```

Results

- **Compiling and running in Eclipse (common)**
  - Save Ship.java and ShipTest.java
  - R-click inside ShipTest.java, Run As → Java Application

- **Compiling and running manually (rare)**

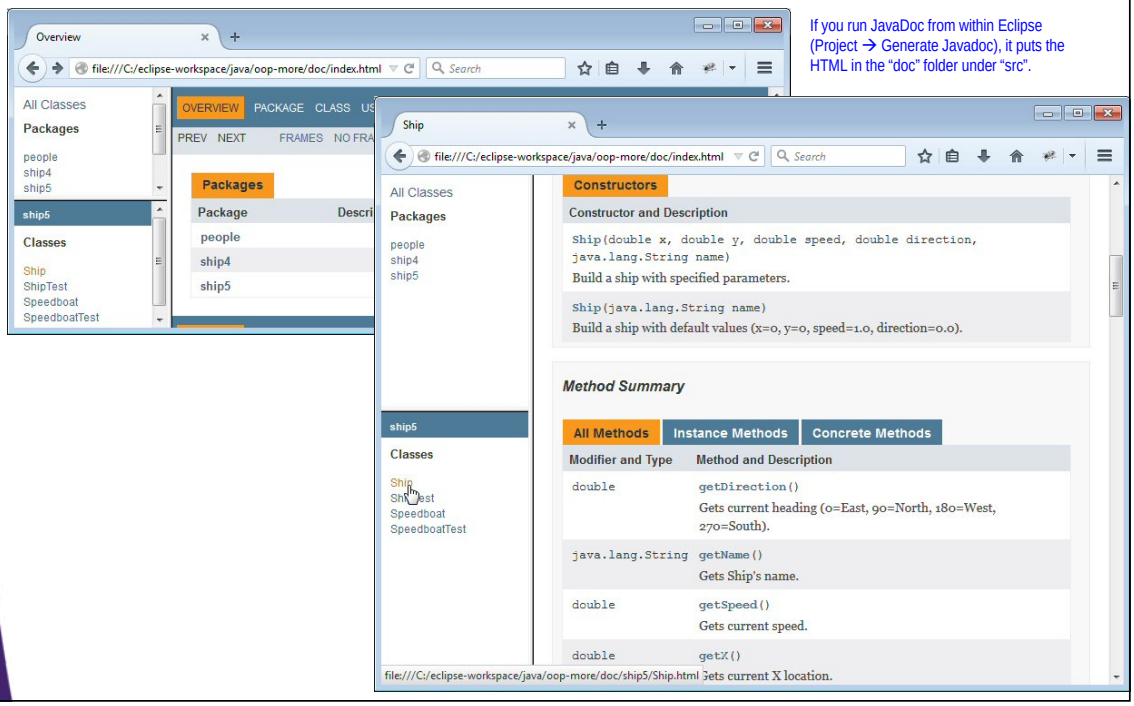
```
> javac ship5\ShipTest.java
> java ship5.ShipTest
> javadoc *.java
```

You can also run JavaDoc from within Eclipse by starting at Project menu and choosing "Generate Javadoc". If it asks you where javadoc.exe is located, you can find it in the bin folder of your Java installation (e.g., C:\Program Files\Java\jdk1.8.0_45\bin)

- **Output:**

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


OOP Design: Testing and Results (Continued)



If you run JavaDoc from within Eclipse (Project → Generate Javadoc), it puts the HTML in the "doc" folder under "src".

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
    - These should go before every public class, every public method, and every public constructor
  - To build online documentation, use "javadoc *.java" from command line or, within Eclipse, do Project → Generate Javadoc

More Details on Getters and Setters

- **Eclipse will automatically build getters/setters from instance variables**
  - R-click anywhere in code
  - Choose Source → Generate Getters and Setters
  - However, if you later click on instance variable and do Refactor → Rename, Eclipse will not automatically rename the accessor methods

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

Q: What is the object-oriented way of getting rich?
A: 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 example**
  - public class ChildClass extends ParentClass { … }

Motivation

- **Supports primary goal of OOP**
  - Supports the key OOP principle of code reuse (i.e., don't write the same code twice)
  - You can design class hierarchies so that shared behavior is inherited by 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 in Eclipse**
  - Save SpeedBoatTest.java
  - R-click, Run As → Java Application
- **Compiling and running manually**

```
> javac ship5\SpeedboatTest.java
```

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

```
> java ship5.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**
  - And nomenclature (parent/child, super/sub, base/extended)
- **Using inherited methods**
- **Using super(…) for inherited constructors**
  - *Only* when the zero-arg constructor is not OK
    - The most common case is to omit super and use zero-arg constructor of parent, but super is used moderately often
- **Using super.someMethod(…) for inherited methods**
  - *Only* when there is a name conflict
    - Used very rarely

Review of Packages

Overview

- **Idea**
  - Organize classes in groups.
- **Syntax**
  - Make folder called somepackage
    - In Eclipse, R-click on “src” and do New → Package
  - Put “package somepackage” at top of file
    - Automatic in Eclipse
  - To use code from another package
    - put “import somepackage.*” below your package statement

Motivation

- **Avoiding name conflicts**
  - Team members can work on different parts of project without worrying about what class names other teams use
- **Different versions for testing**
  - For example, in next section, I have three packages: shapes1, shapes1, shapes3. They have variations on ways to make shapes where you can sum their areas.
    - But I use same core class names (Circle, Rectangle, etc.) in each of the packages

Running Packaged Code that has "main"

- **From Eclipse**
  - Same as always: R-click, Run As → Application
- **From command line**
  - Go to top-level of package hierarchy, i.e., for simple packages, the folder above the one containing the Java code
  - Use the fully-qualified name, i.e., including package

```
> java packagename.Classname ...
```

The toString Method

Overview

- **Idea**
  - If you give a class a toString method, that method is automatically called whenever
    - An object of that class is converted to a String
    - An object of that class is printed
- **Example**

```
public class Person {
  // Main code covered earlier

  @Override
  public String toString() {
    return("PERSON: " + getFullName());
  }
}
```

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
    - Use toString
  - 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;
}
}
```

Wrap-Up

Java OOP References

- **Online**
  - "OOP Concepts" section in Oracle Java Tutorial. See also "Classes and Objects" and "Interfaces and Inheritance".
    - http://docs.oracle.com/javase/tutorial/java/
- **Books**
  - *Murach's Java SE* (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 programming book.
  - *Effective Java, 2nd Edition* (Josh Bloch)
    - In my opinion, by far the best Java best-practices 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
- **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
- **Organization**
  - Put all code in packages
  - Make output more readable by implementing toString

Questions?

More info:
http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial
http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial
http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training
Many additional free tutorials at coreservlets.com (JSF, Android, Ajax, Hadoop, and lots more)

Customized Java EE Training: http://courses.coreservlets.com/
Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.