CSE 331

Introduction;
Review of Java and OOP

slides created by Marty Stepp
http://www.cs.washington.edu/331/
What is this course about?

- specification and documentation
- object-oriented design
  - taking a problem and turning it into a set of well-designed classes
- testing, debugging, and correctness
- learning to use existing software libraries and APIs
- using software tools and development environments (IDEs)
- working in small groups to solve programming projects

- things that are "sort of" course topics:
  - Java language features
  - graphical user interfaces (GUIs)
Building Good Software is Hard

- large software systems are enormously complex
  - millions of "moving parts"

- people expect software to be malleable
  - software mitigates the deficiencies of other components

- we are always trying to do new things with software
  - relevant experience is often missing

- software engineering is about:
  - managing complexity and change
  - coping with potential defects
    - customers, developers, environment, software
Managing Complexity

• abstraction and specification
  ▪ procedural, data, control flow
  ▪ why they are useful and how to use them

• writing, understanding, and reasoning about code
  ▪ the examples are in Java, but the issues are more general

• program design and documentation
  ▪ the process of design; design tools

• pragmatic considerations
  ▪ testing
  ▪ debugging and defensive programming
Prerequisite knowledge

To do well in this course, you should know (or quickly review):

- basic **Java syntax** (loops, if/else, variables, arrays, parameters/return)
- **primitive vs. object** types; value vs. **reference semantics**
- creating **classes** of objects (syntax and semantics)
  - fields, encapsulation, public/private, instance methods, constructors
  - client (external) vs. implementation (internal) views of an object
  - **static** vs. non-static
- **inheritance** and **interfaces** (basic syntax and semantics)
- Java **Collections** Framework (List, Set, Map, Stack, Queue, PriorityQueue)
  - using **generics**; primitive "wrapper" classes
- **exceptions** (throwing and catching)
- **recursion**

*see Review slides on course web site, or Core Java Ch. 1-6, for review material*
OOP and OOD

• **object-oriented programming**: A programming paradigm where a software system is represented as a collection of objects that interact with each other to solve the overall task.

  ▪ most CSE 142 assignments are not object-oriented (why not?)
  
  ▪ many CSE 143 assignments are object-oriented
    • but not all are well-*designed* (seen later)

  ▪ most software you will write after CSE 143 is object-oriented
    • exceptions: functional code; systems programming; web programming
Major OO concepts

- Object-oriented programming is founded on these ideas:

  - **object/class**: An object is an entity that combines data with behavior that acts on that data. A class is a type or category of objects.
  
  - **information hiding (encapsulation)**: The ability to protect some components of the object from external entities ("private").
  
  - **inheritance**: The ability for a class ("subclass") to extend or override functionality of another class ("superclass").
  
  - **polymorphism**: The ability to replace an object with its sub-objects to achieve different behavior from the same piece of code.
  
  - **interface**: A specification of method signatures without supplying implementations, as a mechanism for enabling polymorphism.
Object-oriented design

• **object-oriented design**: The process of planning a system of interacting objects and classes to solve a software problem.
  ▪ (looking at a problem and deducing what classes will help to solve it)
  ▪ one of several styles of software design

• What are the benefits of OO design?
  ▪ How do classes and objects help improve the style of a program?
  ▪ What benefits have you received by using objects created by others?
Inputs to OO design

• OO design is not the start of the software development process. First the dev team may create some or all of the following:

  ▪ **requirements specification**: Documents that describe the desired implementation-independent functionality of the system as a whole.
  ▪ **conceptual model**: Implementation-independent diagram that captures concepts in the problem domain.
  ▪ **use cases**: Descriptions of sequences of events that, taken together, lead to a system doing something useful to achieve a specific goal.
  ▪ **user interface prototype**: Shows and describes the look and feel of the product's user interface.
  ▪ **data model**: An abstract description of how data is represented and used in the system (databases, files, network connections, etc.).
A classic type of object-oriented design question is as follows:

- Look at a description of a particular problem domain or software system and its necessary features in high-level general terms.
- From the description, try to identify items that might be good to represent as classes if the system were to be implemented.

Hints:

- Classes and objects often correspond to nouns in the problem description.
  - Some nouns are too trivial to represent as entire classes; maybe they are simply data (fields) within other classes or objects.
- Behaviors of objects are often verbs in the problem description.
- Look for related classes that might make candidates for inheritance.
OO design exercise

What classes are in this Texas Hold 'Em poker system?

- 2 to 8 human or computer players
- Computer players with skill setting: easy, medium, hard
- Each player has a name and stack of chips
- Summary of each hand:
  - Dealer collects ante from appropriate players, shuffles the deck, and deals each player a hand of 2 cards from the deck.
  - A betting round occurs, followed by dealing 3 shared cards from the deck.
  - As shared cards are dealt, more betting rounds occur, where each player can fold, check, or raise.
  - At the end of a round, if more than one player is remaining, players' hands are compared, and the best hand wins the pot of all chips bet.
What classes are in this video store kiosk system?

- The software is for a video kiosk that replaces human clerks.
- A customer with an account can use their membership and credit card at the kiosk to check out a video.
- The software can look up movies and actors by keywords.
- A customer can check out up to 3 movies, for 5 days each.
- Late fees can be paid at the time of return or at next checkout.
Java's object-oriented features (overview)
• **field**: A variable inside an object that is part of its state.
  – Each object has *its own copy* of each field.

• Declaration syntax:

```java
private type name;
```

– Example:

```java
public class Point {
    private int x;
    private int y;
    ...
}
```
Instance methods

- **instance method** (or object method): Exists inside each object of a class and gives behavior to each object.

```java
public type name(parameters) {
    statements;
}
```

- same syntax as static methods, but without `static` keyword

Example:

```java
public void translate(int dx, int dy) {
    x += dx;
    y += dy;
}
```
Categories of methods

- **accessor**: A method that lets clients examine object state.
  - Examples: `distance`, `distanceFromOrigin`
  - often has a non-`void` return type

- **mutator**: A method that modifies an object's state.
  - Examples: `setLocation`, `translate`

- **helper**: Assists some other method in performing its task.
  - often declared as private so outside clients cannot call it
The `toString` method

tells Java how to convert an object into a `String` for printing

```java
public String toString() {
    code that returns a String representing this object;
}
```

- Method name, return, and parameters must match *exactly*.

- Example:

  ```java
  // Returns a String representing this Point.
  public String toString() {
      return "(" + x + ", " + y + ");
  }
  ```
Constructors

- **constructor**: Initializes the state of new objects.

  ```java
  public type(parameters) {
    statements;
  }
  ```

  - runs when the client uses the `new` keyword
  - no return type is specified; implicitly "returns" the new object

```java
public class Point {
  private int x;
  private int y;

  public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
  }
}
```
The keyword **this**

- **this**: Refers to the implicit parameter inside your class. *(a variable that stores the object on which a method is called)*

  - Refer to a field: `this.field`
  - Call a method: `this.method(parameters);`
  - One constructor `this(parameters);`
    can call another:
public class Point {
    private int x;
    private int y;

    public Point() {
        this(0, 0);
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...}

• Avoids redundancy between constructors
• Only a constructor (not a method) can call another constructor
Inheritance

- **inheritance**: Forming new classes based on existing ones.
  - a way to share/reuse code between two or more classes
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
    - gets a copy of every field and method from superclass
  - **is-a relationship**: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.
Inheritance syntax

public class name extends superclass {

• Example:

public class Lawyer extends Employee {
    ...
}

• By extending Employee, each Lawyer object now:
  • receives a copy of each method from Employee automatically
  • can be treated as an Employee by client code

• Lawyer can also replace ("override") behavior from Employee.
The super keyword

- A subclass can call its parent's method/constructor:

```java
super . method (parameters)  // method
super (parameters);         // constructor
```

```java
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super (name);
    }

    // give Lawyers a $5K raise (better)
    public double getSalary() {
        double baseSalary = super . getSalary();
        return baseSalary + 5000.00;
    }
}
```
Shapes example

• Consider the task of writing classes to represent 2D shapes such as Circle, Rectangle, and Triangle.

• Certain attributes or operations are common to all shapes:
  ▪ perimeter: distance around the outside of the shape
  ▪ area: amount of 2D space occupied by the shape

  ▪ Every shape has these, but each computes them differently.
Interfaces

• **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A *Lawyer* can be treated as an *Employee* and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A *Rectangle* object can be treated as a *Shape* but inherits no code.

• Analogous to non-programming idea of roles or certifications:
  - "I'm certified as a CPA accountant. This assures you I know how to do taxes, audits, and consulting."
  - "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know how to compute my area and perimeter."
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}

Example:
    public interface Shape {
        public double area();
        public double perimeter();
    }

```
    class Circle {
        radius
        Circle(radius)
        area()
        perimeter()
    }
    class Rectangle {
        width, height
        Rectangle(w,h)
        area()
        perimeter()
    }
    class Triangle {
        a, b, c
        Triangle(a, b, c)
        area()
        perimeter()
    }
```
Implementing an interface

```java
public class name implements interface {
    ...
}
```

- A class can declare that it "implements" an interface.
  - The class promises to contain each method in that interface.
    (Otherwise it will fail to compile.)

- Example:
  ```java
  public class Rectangle implements Shape {
      ...
      public double area() { ... }
      public double perimeter() { ... }
  }
  ```
Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.
  - they allow **polymorphism**
    (the same code can work with different types of objects)

```java
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
    System.out.println();
}

... Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```