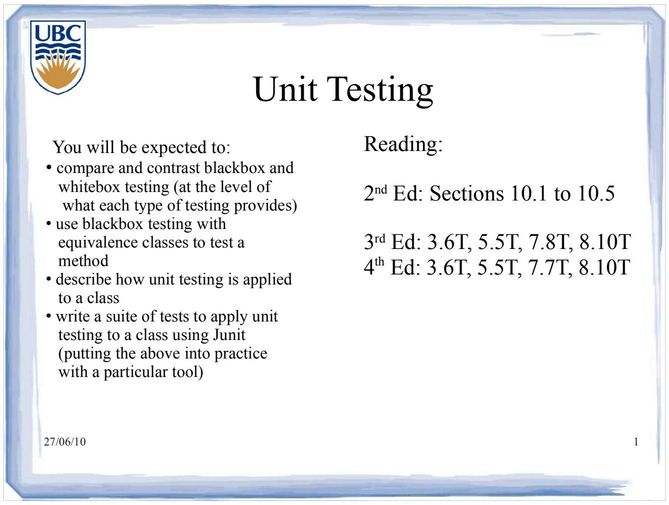


Unit Testing

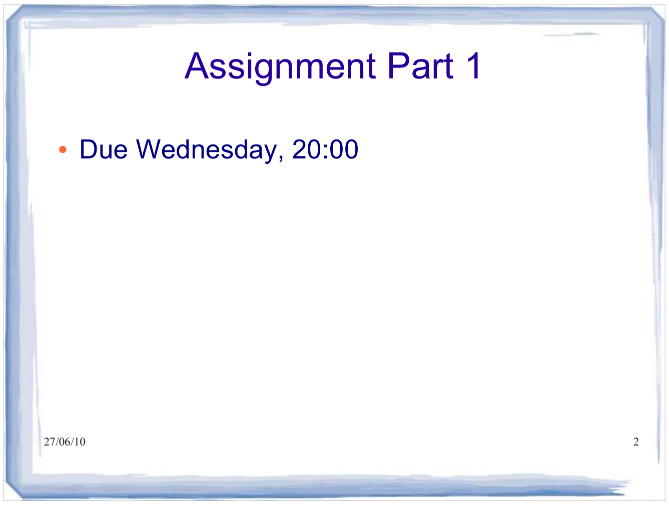
You will be expected to:

- compare and contrast blackbox and whitebox testing (at the level of what each type of testing provides)
- use blackbox testing with equivalence classes to test a method
- describe how unit testing is applied to a class
- write a suite of tests to apply unit testing to a class using Junit (putting the above into practice with a particular tool)

Reading:

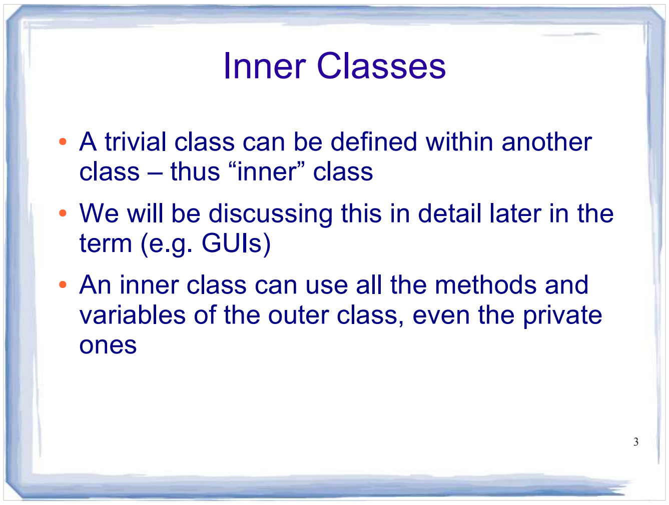
2nd Ed: Sections 10.1 to 10.5

3rd Ed: 3.6T, 5.5T, 7.8T, 8.10T
4th Ed: 3.6T, 5.5T, 7.7T, 8.10T



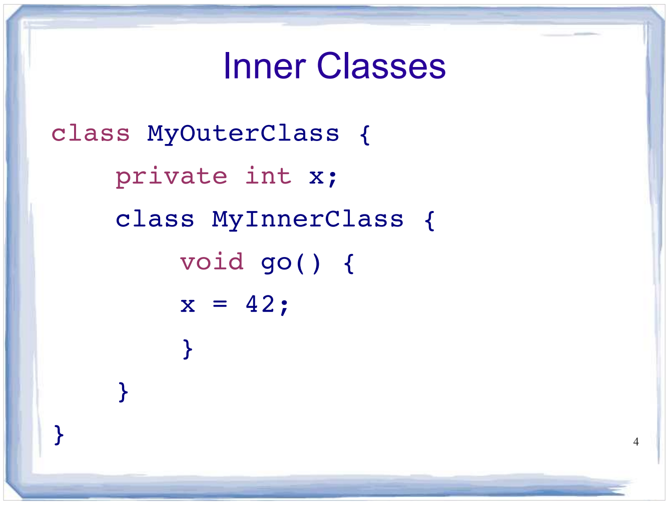
Assignment Part 1

- Due Wednesday, 20:00



Inner Classes

- A trivial class can be defined within another class – thus “inner” class
- We will be discussing this in detail later in the term (e.g. GUIs)
- An inner class can use all the methods and variables of the outer class, even the private ones



Inner Classes

```
class MyOuterClass {
    private int x;
    class MyInnerClass {
        void go() {
        x = 42;
        }
    }
}
```

Inner Classes

```
class MyOuter {
    private int x;
    class MyInner {
        void go() {
        x = 42;
        }
    }
}
```

We can use x just as if it were a variable of the inner class

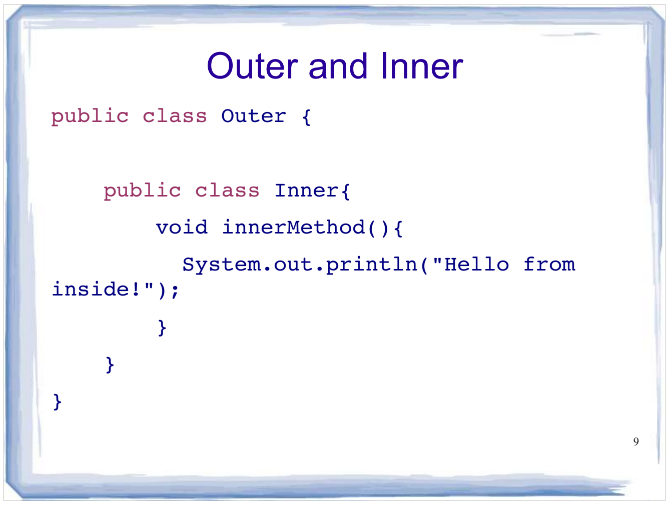
Inner Classes

- An instance of the *inner* class is tied to an instance of the *outer* class

```
class MyOuter {
    private int x;
    MyInner inner = new MyInner();
    public void doStuff(){
      inner.go();}

    class MyInner {
        void go() {
        x = 42;}
       }  // end of inner class
    }     // end of outer class
```

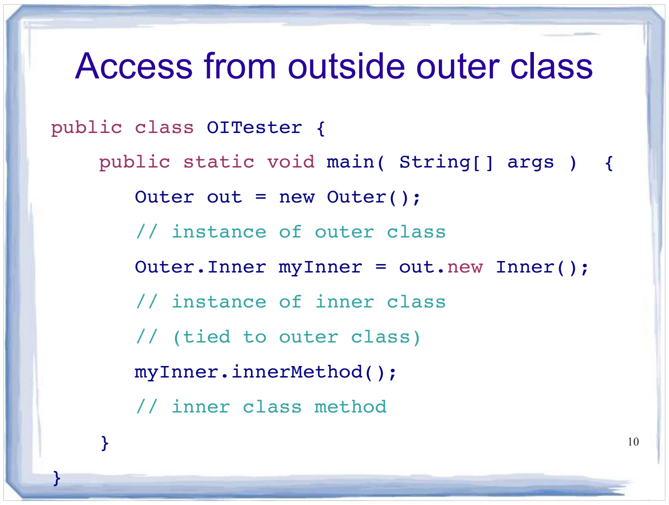
Inner Class Visibility

- Unlike a regular (outer) class, an inner class can be private
    - Q: Why can't a regular class be private?
- In that case, the inner class and its methods cannot be accessed outside of the outer class
- But if the inner class is public, there's nothing stopping us from doing this...



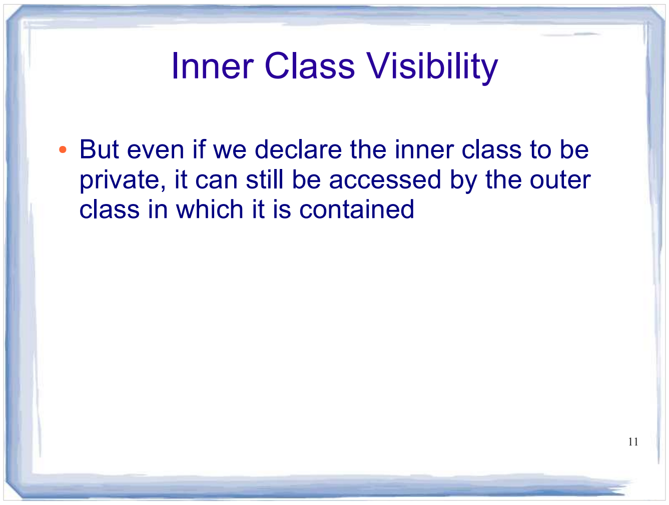
Outer and Inner

```
public class Outer {


    public class Inner{
        void innerMethod(){
          System.out.println("Hello from
inside!");
        }
    }
}
```



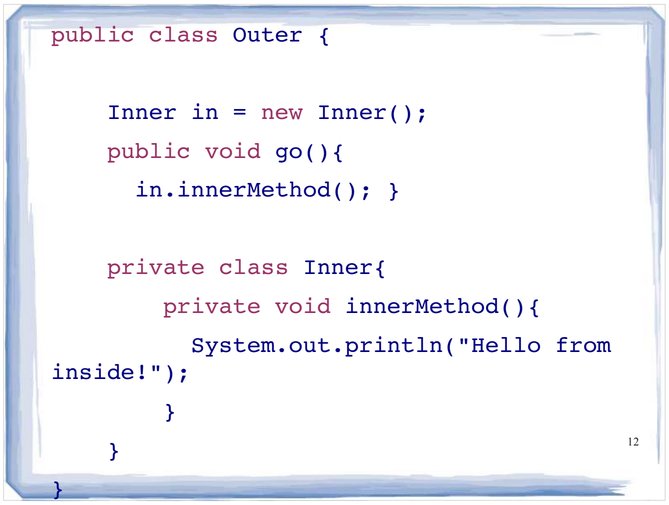
Access from outside outer class

```
public class OITester {
    public static void main( String[] args )  {
       Outer out = new Outer();
       // instance of outer class
       Outer.Inner myInner = out.new Inner();
       // instance of inner class
       // (tied to outer class)
       myInner.innerMethod();
       // inner class method
    }
}
```



Inner Class Visibility

- But even if we declare the inner class to be private, it can still be accessed by the outer class in which it is contained



```
public class Outer {


    Inner in = new Inner();
    public void go(){
      in.innerMethod(); }


    private class Inner{
        private void innerMethod(){
          System.out.println("Hello from
inside!");
        }
    }
}
```

Inner Class

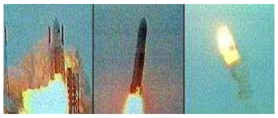
```
public class OITest {

public static void main(String[] args) {
Outer out = new Outer();
out.go();
```

"Hello from inside!"

```
}
}
```

Software Testing

Some "Famous" Software Problems



Ariane 5, June 4, 1996

Vancouver Stock Exchange
Rounding Problem, 1983

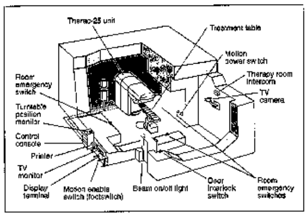


Figure 1. Typical Therac-25 facility.

Therac-25, mid-80s

See comp.risks for more each day.

Testing

- Terminology:
  - **test case**: a set of inputs and expected outputs that test a single use of a piece of the system (e.g., a method, a class, a subsystem)
  - **test:** a set of test cases
  - **test driver**: code that sets up any context needed to run a test, calls the test case(s), and displays the results
  - **test stub**: code that simulates the behaviour of the actual code that is still to be written

Testing Activities

- **Unit Testing** (for individual classes or small groups of classes)
  - find differences between what an object does, and what it is supposed to do
  - testing one (or a few) class(es) is easier than testing the whole system
  - Enables incremental and parallel testing
- There are other kinds of testing (e.g., …)
  - Integration Testing (for a group of classes or subsystems)
  - System Testing (check if system does what is intended)

Unit Testing Types

- There are two major types of unit testing

Blackbox testing
- focuses on input/outputs only
- cases are derived from class specification

- is good for testing interfaces
- does not effectively test all cases

Whitebox testing
- focuses on the component's internal structure
- attempts to test all states and interactions

- also known as structural testing
- complementary to black-box testing

Blackbox Testing : Input Partition

- In general, we can't fully test an application.
  - applications often accept many different inputs
  - testing every different combination of inputs is practically impossible.
- To test a method, divide its inputs into equivalence classes *(here we use the term class as category, not a Java class!)*
  - all values within an equivalence class behave similarly with respect to specification
  - equivalence classes are disjoint
  - they should cover the entire input space

Blackbox Testing : Input Partition (cont'd)

- Use preconditions, postconditions and class invariants to determine the equivalence classes for the input partition
- The method preconditions will divide the input into
  - *Valid* space that satisfies the preconditions and
  - *Invalid* space that violates the preconditions

Blackbox Testing : Selecting Test Cases

- First, identify the valid input space and divide it into equivalence classes
- From each equivalence class, select:
  - at least one typical value - equivalence partition testing (sometimes called equivalence class testing)
  - some boundary values – boundary testing

Boundary test cases

- These test cases are at the boundary of acceptable inputs

Example 1

```
class Account {
   …
   /**
  * @pre amount >= 0
  * …
  */
  public void deposit(double amount ) { … }
}
```

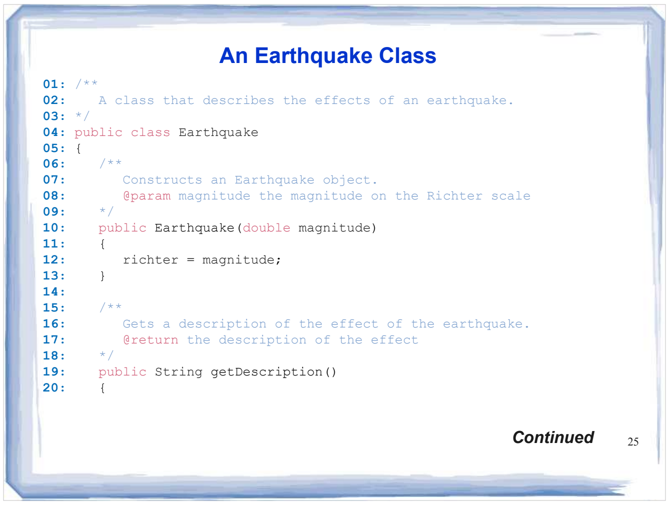
- One equivalence class that satisfies the precondition: *amount >= 0*
  - Select at least one typical member of the class, *amount = 200*
  - Select values at boundaries, only one boundary, *amount = 0*
- Test cases are then: {amount = 200, amount = 0 }

Example 2

```
class Account {
   …
   /**
  * @pre true
  * …
  * @throws IllegalValueException when amount < 0
  */
  public void withdraw(double amount) { … }
}
```

- Two equivalence classes. What are they?

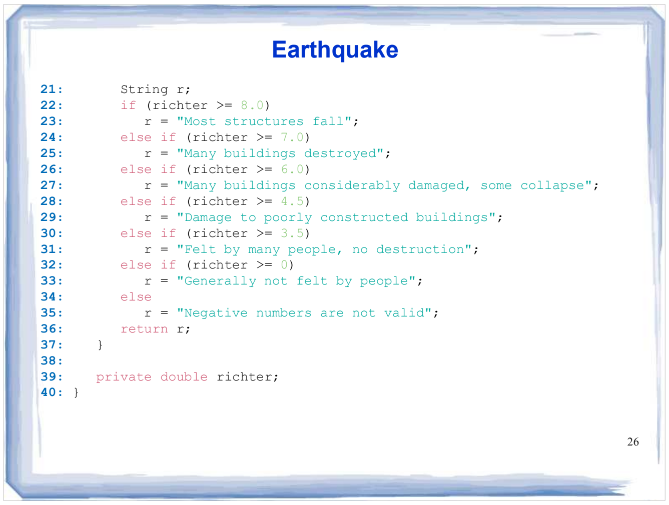
- What test cases would you specify?



An Earthquake Class

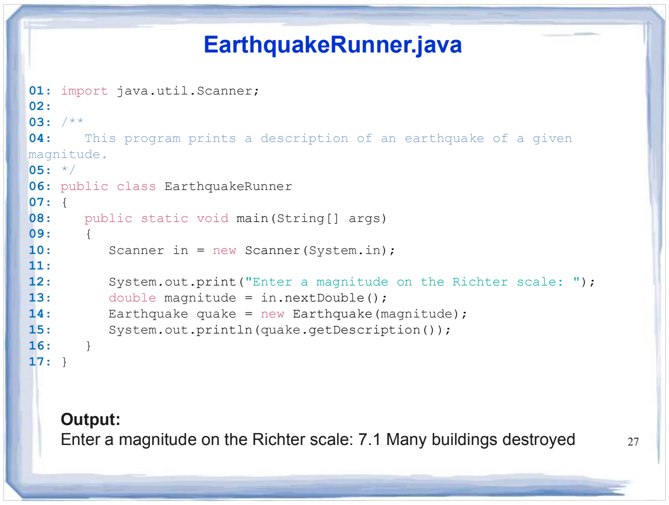
```
/**
   A class that describes the effects of an earthquake.
*/
public class Earthquake
{
   /**
      Constructs an Earthquake object.
      @param magnitude the magnitude on the Richter scale
   */
   public Earthquake(double magnitude)
   {
      richter = magnitude;
   }

   /**
      Gets a description of the effect of the earthquake.
      @return the description of the effect
   */
   public String getDescription()
   {
```

Continued



Earthquake

```
      String r;
      if (richter >= 8.0)
         r = "Most structures fall";
      else if (richter >= 7.0)
         r = "Many buildings destroyed";
      else if (richter >= 6.0)
         r = "Many buildings considerably damaged, some collapse";
      else if (richter >= 4.5)
         r = "Damage to poorly constructed buildings";
      else if (richter >= 3.5)
         r = "Felt by many people, no destruction";
      else if (richter >= 0)
         r = "Generally not felt by people";
      else
         r = "Negative numbers are not valid";
      return r;
   }

   private double richter;
}
```

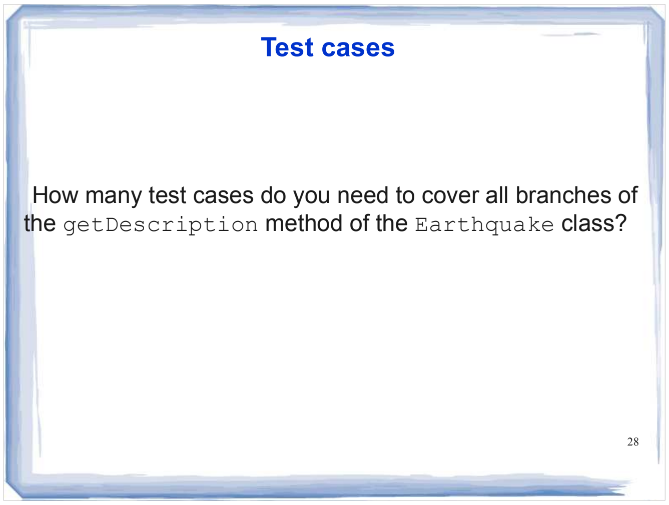


EarthquakeRunner.java

```
import java.util.Scanner;

/**
   This program prints a description of an earthquake of a given magnitude.
*/
public class EarthquakeRunner
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("Enter a magnitude on the Richter scale: ");
      double magnitude = in.nextDouble();
      Earthquake quake = new Earthquake(magnitude);
      System.out.println(quake.getDescription());
   }
}
```

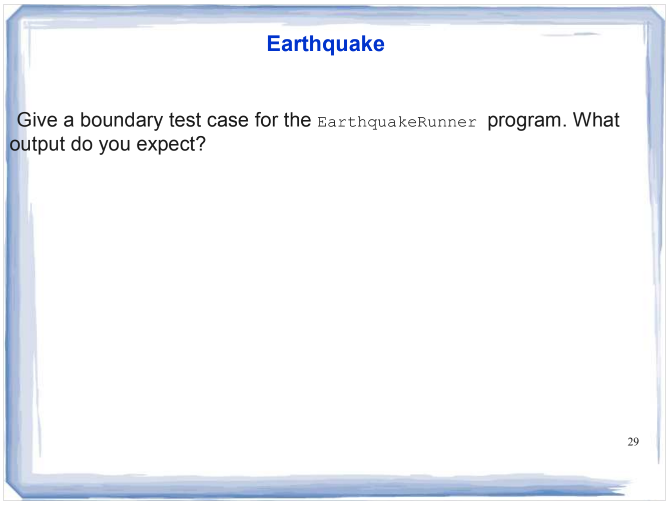
Output:

Enter a magnitude on the Richter scale: 7.1 Many buildings destroyed



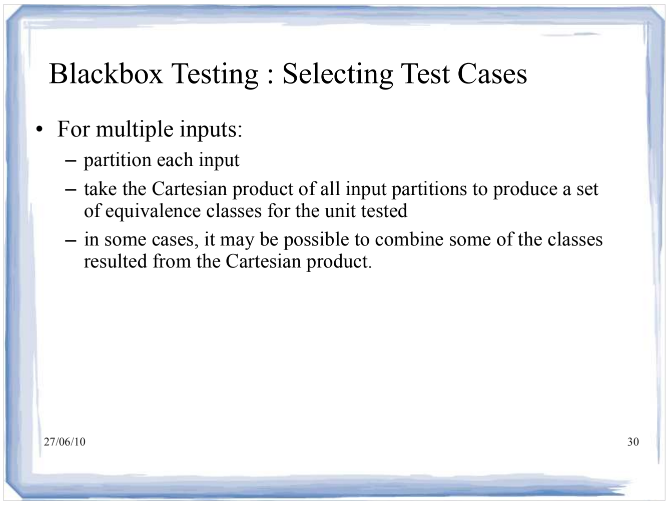
Test cases

How many test cases do you need to cover all branches of the `getDescription` method of the `Earthquake` class?



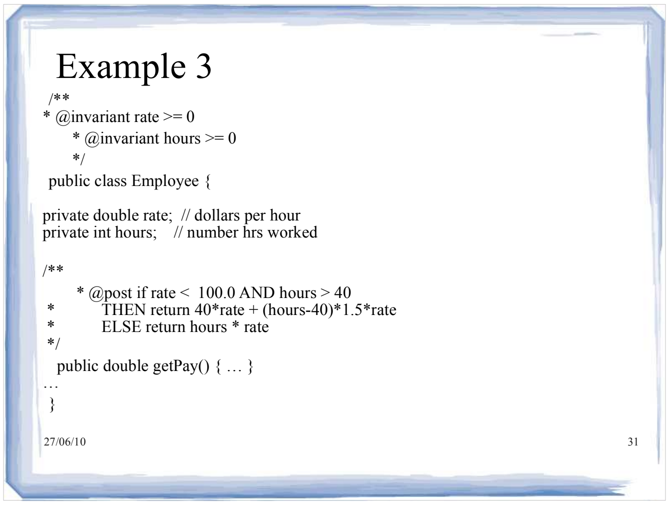
Earthquake

Give a boundary test case for the `EarthquakeRunner` program. What output do you expect?



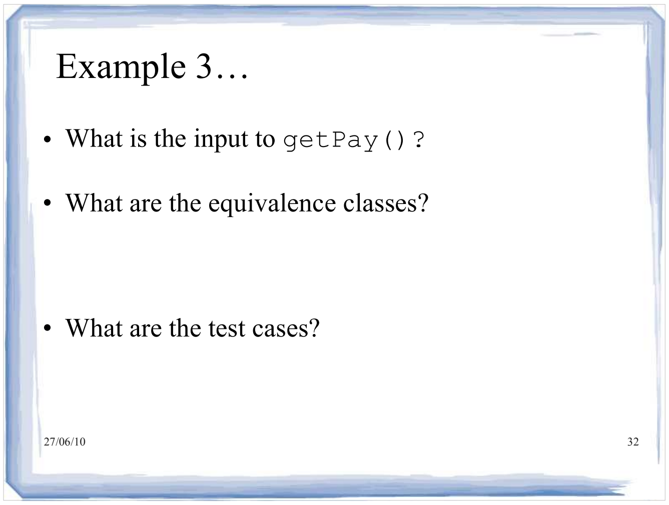
Blackbox Testing : Selecting Test Cases

- For multiple inputs:
  - partition each input
  - take the Cartesian product of all input partitions to produce a set of equivalence classes for the unit tested
  - in some cases, it may be possible to combine some of the classes resulted from the Cartesian product.



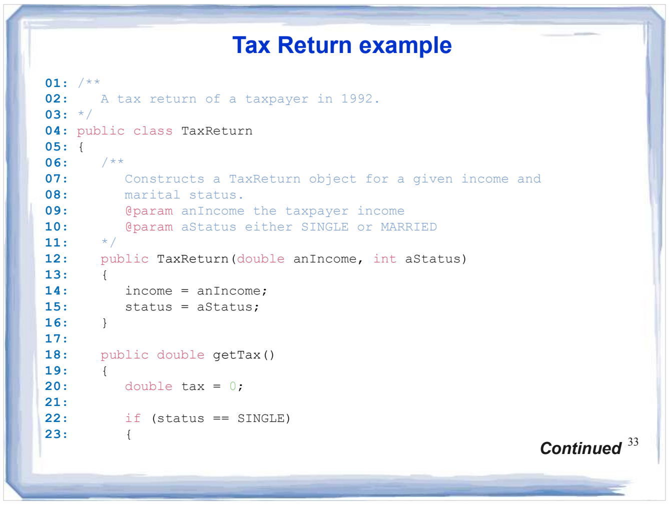
Example 3

```
 /**
* @invariant rate >= 0
     * @invariant hours >= 0
       */
 public class Employee {

private double rate;  // dollars per hour
private int hours;    // number hrs worked

/**
      * @post if rate <  100.0 AND hours > 40
 *          THEN return 40*rate + (hours-40)*1.5*rate
 *          ELSE return hours * rate
 */
   public double getPay() { … }
…
 }
```



Example 3…

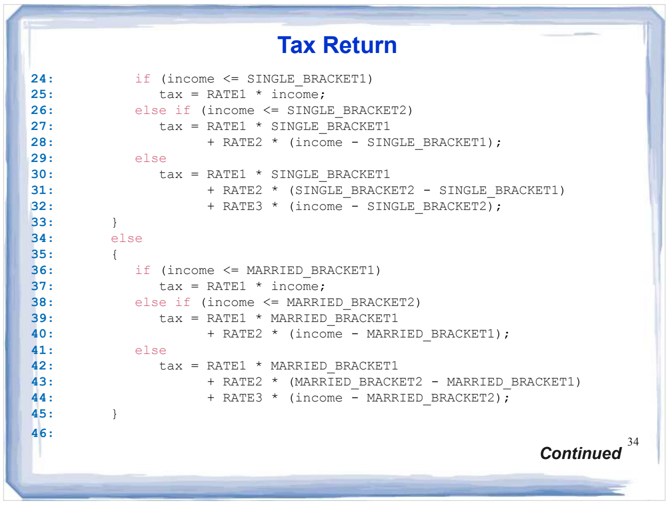
- What is the input to `getPay()`?
- What are the equivalence classes?
- What are the test cases?



Tax Return example

```
/**
   A tax return of a taxpayer in 1992.
*/
public class TaxReturn
{
   /**
      Constructs a TaxReturn object for a given income and
      marital status.
      @param anIncome the taxpayer income
      @param aStatus either SINGLE or MARRIED
   */
   public TaxReturn(double anIncome, int aStatus)
   {
      income = anIncome;
      status = aStatus;
   }

   public double getTax()
   {
      double tax = 0;

      if (status == SINGLE)
      {
```

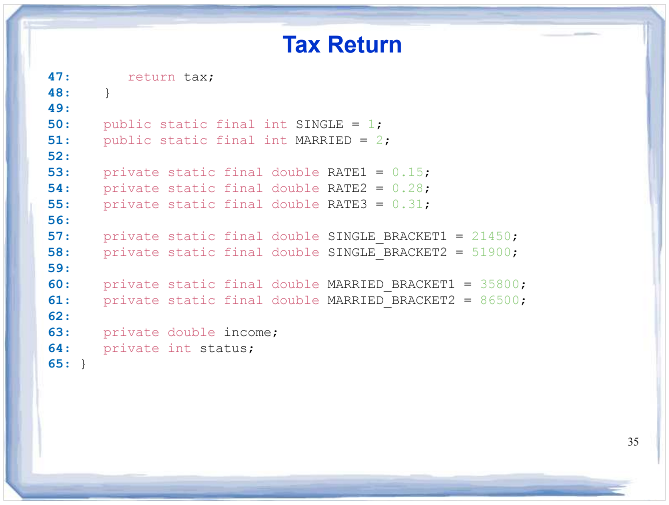
Continued



Tax Return

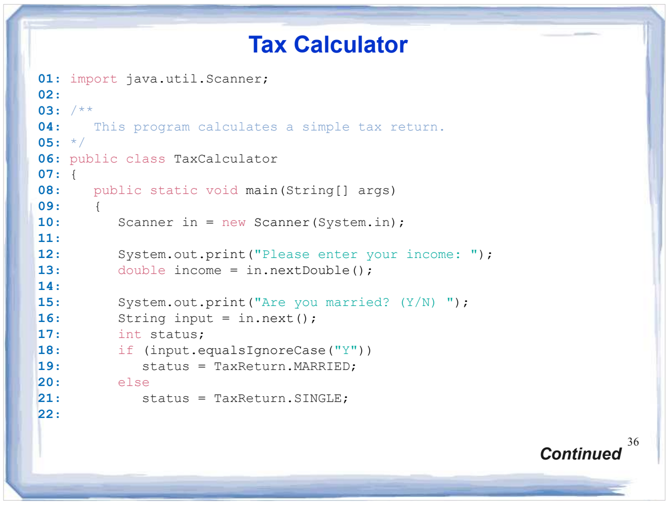
```
         if (income <= SINGLE_BRACKET1)
            tax = RATE1 * income;
         else if (income <= SINGLE_BRACKET2)
            tax = RATE1 * SINGLE_BRACKET1
                  + RATE2 * (income - SINGLE_BRACKET1);
         else
            tax = RATE1 * SINGLE_BRACKET1
                  + RATE2 * (SINGLE_BRACKET2 - SINGLE_BRACKET1)
                  + RATE3 * (income - SINGLE_BRACKET2);
      }
      else
      {
         if (income <= MARRIED_BRACKET1)
            tax = RATE1 * income;
         else if (income <= MARRIED_BRACKET2)
            tax = RATE1 * MARRIED_BRACKET1
                  + RATE2 * (income - MARRIED_BRACKET1);
         else
            tax = RATE1 * MARRIED_BRACKET1
                  + RATE2 * (MARRIED_BRACKET2 - MARRIED_BRACKET1)
                  + RATE3 * (income - MARRIED_BRACKET2);
      }

```

Continued



Tax Return

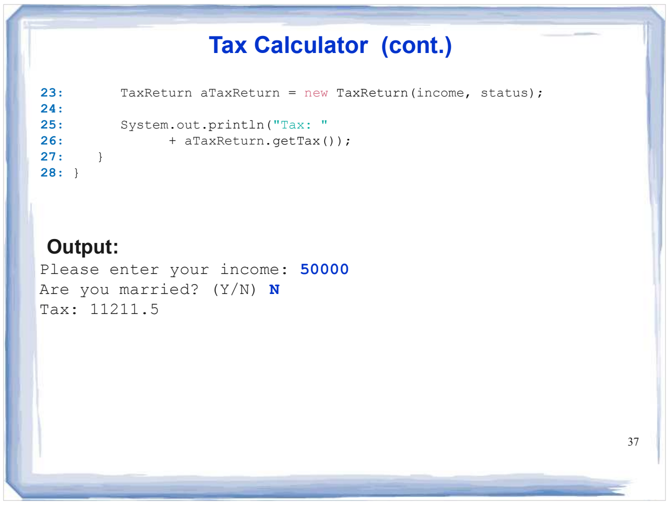
```
      return tax;
   }

   public static final int SINGLE = 1;
   public static final int MARRIED = 2;

   private static final double RATE1 = 0.15;
   private static final double RATE2 = 0.28;
   private static final double RATE3 = 0.31;

   private static final double SINGLE_BRACKET1 = 21450;
   private static final double SINGLE_BRACKET2 = 51900;

   private static final double MARRIED_BRACKET1 = 35800;
   private static final double MARRIED_BRACKET2 = 86500;

   private double income;
   private int status;
}
```



Tax Calculator

```
import java.util.Scanner;

/**
   This program calculates a simple tax return.
*/
public class TaxCalculator
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.print("Please enter your income: ");
      double income = in.nextDouble();

      System.out.print("Are you married? (Y/N) ");
      String input = in.next();
      int status;
      if (input.equalsIgnoreCase("Y"))
         status = TaxReturn.MARRIED;
      else
         status = TaxReturn.SINGLE;

```

Continued

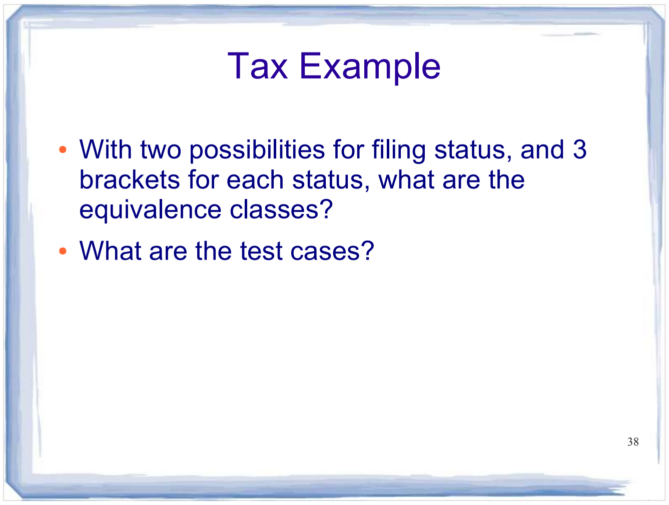


Tax Calculator (cont.)

```
        TaxReturn aTaxReturn = new TaxReturn(income, status);

        System.out.println("Tax: "
              + aTaxReturn.getTax());
    }
}
```

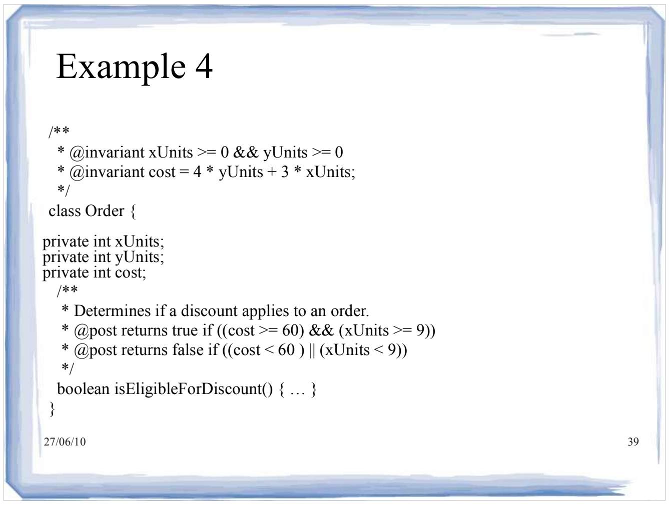
Output:

```
Please enter your income: 50000
Are you married? (Y/N) N
Tax: 11211.5
```



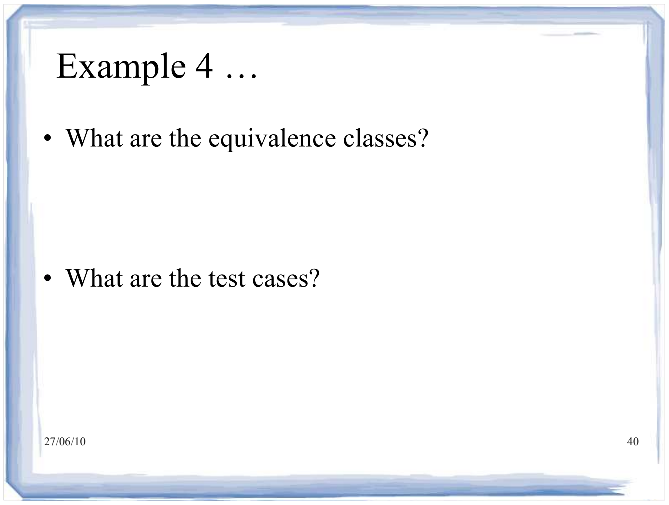
Tax Example

- With two possibilities for filing status, and 3 brackets for each status, what are the equivalence classes?
- What are the test cases?



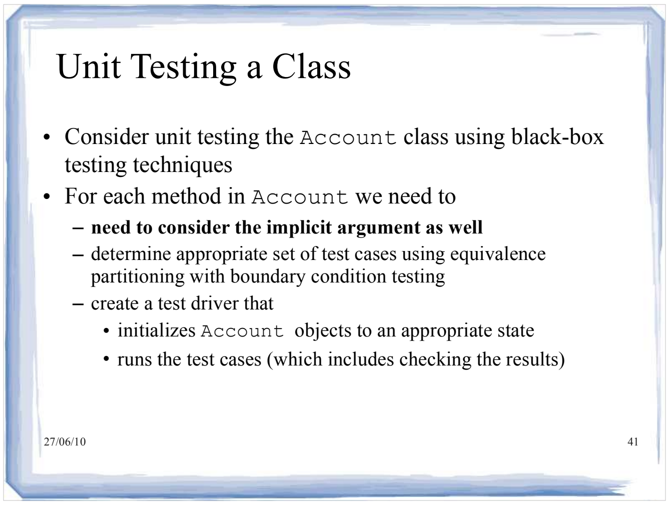
Example 4

```
/**
  * @invariant xUnits >= 0 && yUnits >= 0
  * @invariant cost = 4 * yUnits + 3 * xUnits;
  */
class Order {
private int xUnits;
private int yUnits;
private int cost;
   /**
    * Determines if a discount applies to an order.
    * @post returns true if ((cost >= 60) && (xUnits >= 9))
    * @post returns false if ((cost < 60 ) || (xUnits < 9))
    */
  boolean isEligibleForDiscount() { … }
}
```



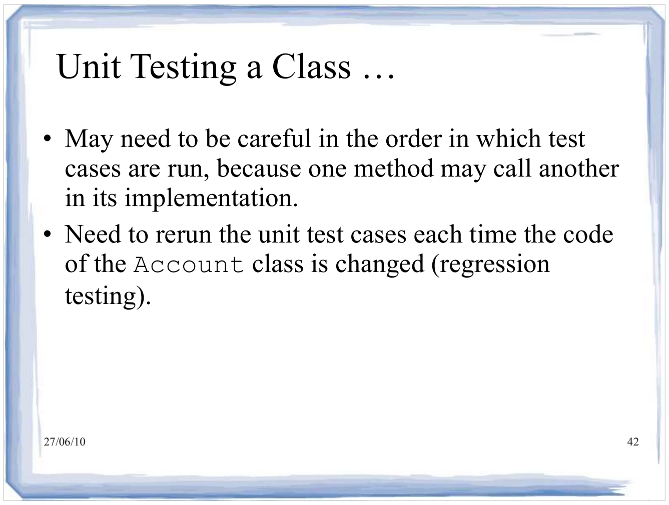
Example 4 …

- What are the equivalence classes?
- What are the test cases?



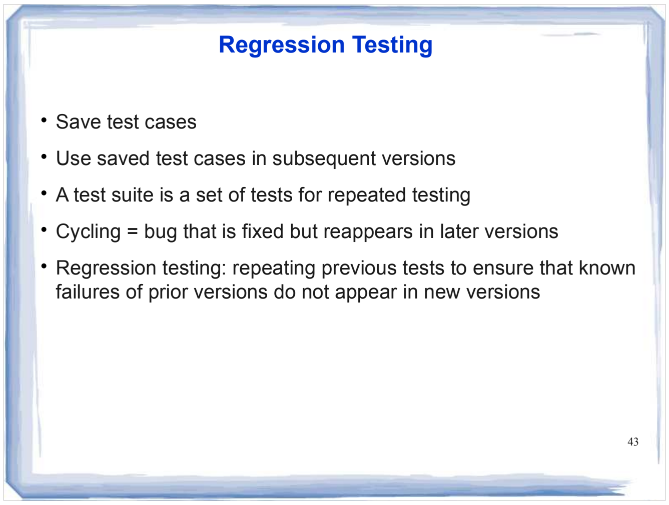
Unit Testing a Class

- Consider unit testing the `Account` class using black-box testing techniques
- For each method in `Account` we need to
  - **need to consider the implicit argument as well**
  - determine appropriate set of test cases using equivalence partitioning with boundary condition testing
  - create a test driver that
    - initializes `Account` objects to an appropriate state
    - runs the test cases (which includes checking the results)



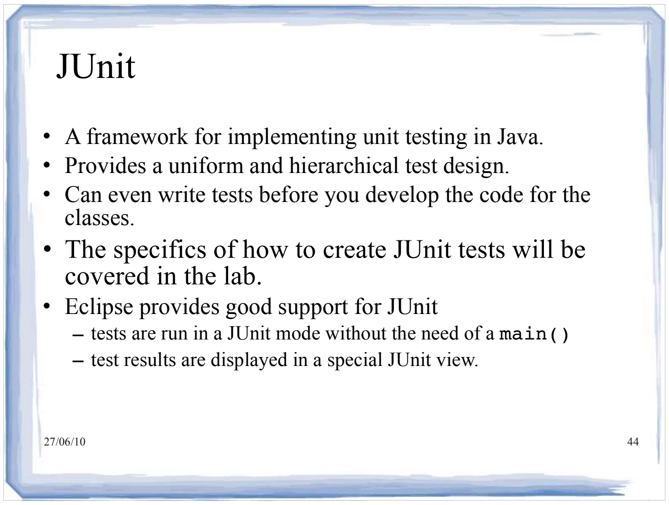
Unit Testing a Class …

- May need to be careful in the order in which test cases are run, because one method may call another in its implementation.
- Need to rerun the unit test cases each time the code of the `Account` class is changed (regression testing).



Regression Testing

- Save test cases
- Use saved test cases in subsequent versions
- A test suite is a set of tests for repeated testing
- Cycling = bug that is fixed but reappears in later versions
- Regression testing: repeating previous tests to ensure that known failures of prior versions do not appear in new versions



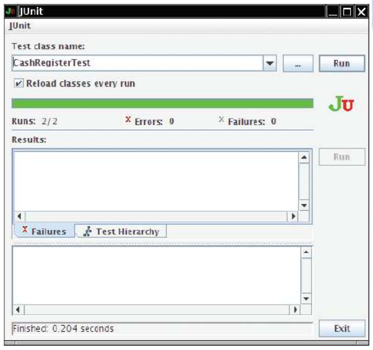
JUnit

- A framework for implementing unit testing in Java.
- Provides a uniform and hierarchical test design.
- Can even write tests before you develop the code for the classes.
- The specifics of how to create JUnit tests will be covered in the lab.
- Eclipse provides good support for JUnit
  - tests are run in a JUnit mode without the need of a `main()`
  - test results are displayed in a special JUnit view.

Unit Testing Frameworks

- Unit test frameworks simplify the task of writing classes that contain many test cases
- JUnit: `http://junit.org`
  Built into some IDEs like BlueJ and Eclipse
- Philosophy: whenever you implement a class, also make a companion test class. Run all tests whenever you change your code

Unit Testing Frameworks



Test class name: CashRegisterTest

☑ Reload classes every run

Runs: 2/2 Errors: 0 Failures: 0

Finished: 0.204 seconds

Figure 6 Unit Testing with JUnit

Junit Test Example

Provide a JUnit test class with one test case for the `Earthquake` class in Chapter 5.

Answer: Here is one possible answer, using the JUnit 4 style.

```
public class EarthquakeTest
{
   @Test public void testLevel4()
   {
      Earthquake quake = new Earthquake(4);
      Assert.assertEquals("Felt by many people, no
          destruction", quake.getDescription());
   }
}
```

In-Class Exercise

Consider the following method that determines if a year is a leap year:

```
class Year {
   …
   /**
    * Determine by the Gregorian calendar if the given
    * year is a leap year.
    * @pre year >= 1582
    */
   public static boolean isLeap(int aYear) { … }
}
```

- A year is a leap year if and only if:
  - (aYear % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0))
- What test cases would you chose to test isLeap()?

Write Tests First

- Some people advise writing your test code first
  - see Extreme Programming (XP)
- What benefits would this have?

Learning Goals Review

You will be expected to:

- compare and contrast blackbox and whitebox testing (at the level of what each type of testing provides)
- use blackbox testing with equivalence classes to test a method
- describe how unit testing is applied to a class
- write a suite of tests to apply unit testing to a class using Junit (putting the above into practice with a particular tool)

Tea break!

Class Design II: Class Diagrams

You should be able to:

- interpret UML class diagrams to identify relationships between classes
- draw a UML class diagram to represent the design of a software system
- describe the basic design principles of low coupling and high cohesion
- design a software system (expressed in UML) from a given specification that adheres to basic design principles (low coupling and high cohesion)
- identify elements of a given design that violate the basic design principles of low coupling, high cohesion

Reading:

2nd Ed:

Chapter 9: 9.1, 9.2
Chapter 17: 17.2, 17.3, 17.4

3rd and 4th Eds:

Chapter 8: 8.1, 8.2
Chapter 12: 12.2, 12.3, 12.4

Some ideas in this section come from:

"Practical Object-Oriented Development with UML and Java"
R. Lee, W. Tepfenhart, Prentice Hall, 2002.

"Object-Oriented Software Development Using Java",
Xiaoping Jia, Addison Wesley, 2002

Where are we?

- We have seen
  - how to design a single class
    - define attributes, methods, invariants, pre/post-conditions
  - how to implement a class robustly using
    - exceptions
    - unit testing
- Now we are going to discuss
  - how to identify the classes we need in order to provide a solution to a problem
  - what relationships exist between these classes
  - good design principles

Where are we?

- The overall roadmap of the course…

	Topic
✓	Design and implementation of a single class
Now	Design of multiple classes
	Collections
	Implementation techniques GUI Threads Streams

Software Design

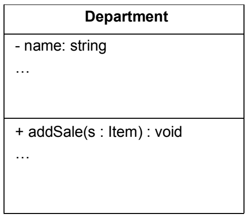
- Difficult, interesting, and important phase of software development
- Based on the requirements we have defined for a given problem, we need to identify and define:
  - **classes and their relationships**
  - **the attributes of each class**
  - **the behaviour of each class**
  - **the interactions between classes**
- We focus on the functionality and static relationships, *not* on implementation details

Representing Design: UML

- To represent the structure of a software system we need to show:
  - its classes
  - the relationships between the classes
- UML (Unified Modelling Language) is graphical modelling language that is used to describe these.
- UML allows a user to describe different **views** (aspects) of a software system
  - static view of the components, how components are deployed to different machines, etc.
- We will focus on one type of UML diagram which is called a ***Class Diagram*** and describes the static, structure (logical view) of the system

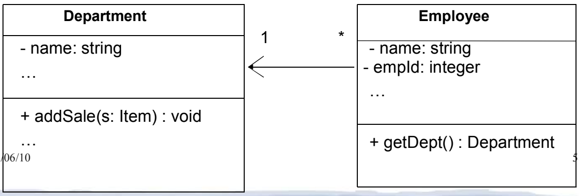
Class Diagram

- Describes the static structure of a system
  - its classes
  - relationships between classes
- Example of a class:



Relationship 1: Association

- **Association**:
  - A *structural* relationship that describes a connection between objects: each object of one type contains reference(s) to objects of the other type.
- Example: Unidirectional association
  - employee stores a reference of a department



Relationship 1: Association

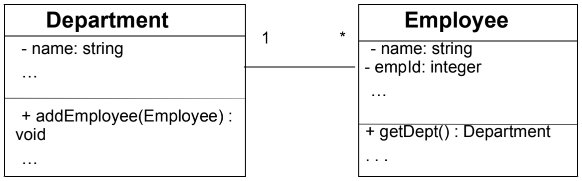
- "*Associations* are stronger than dependencies and typically indicate that one class retains a relationship to another class over an extended period of time. The lifelines of two objects linked by associations are probably not tied together (meaning one can be destroyed without necessarily destroying the other)."
  - UML 2.0 In a Nutshell

Relationship 1: Association

- Typically read as a "has a" relationship
- Associations have explicit notation to indicate navigability
- The arrows indicate whether you can navigate from one class to the other
- Relationship indicated by solid line, open arrow (no arrow if bidirectional...)
- Line may be adorned with a phrase or symbols to add information

Bidirectional Association

- Indicates that both classes reference each other
- Shown with a line without arrows
- Example:



Department

- name: string
…

+ addEmployee(Employee) : void
…

1

*

Employee

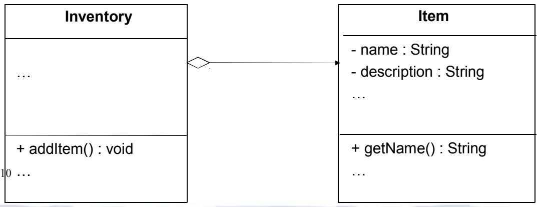
- name: string
- empId: integer
…

+ getDept() : Department
. . .

- 1 and * in the previous examples are called ***multiplicities***
  - indicate the number of objects of this side that are associated by each object of the other side
  - * means any number

Relationship 2: Aggregation

- **Aggregation**:
  - A special form of association that specifies a whole-part relationship between the aggregate (the whole) and a component (the part)
- Example:



Inventory

…

+ addItem() : void
…

Item

- name : String
- description : String
…

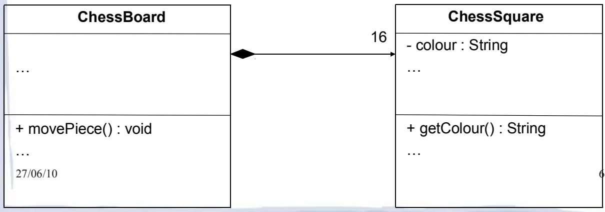
+ getName() : String
…

Relationship 2: Aggregation

- "*Aggregation* is a stronger version of association. Unlike association, aggregation typically implies ownership and may imply a relationship between lifelines."
  – UML 2.0 In a Nutshell
- Typically read as a "owns a" relationship
- Aggregation indicated by diamond shape next to owning class and solid line to owned class

Relationship 3: Composition

- **Composition**:
  - a *form of aggregation*, where the composite (whole) strongly owns the parts
  - when the whole is deleted (dies) the parts are also deleted (die)
  - A part is in exactly one whole (implicit multiplicity of 1)
- Example:



ChessBoard

…

+ movePiece() : void
…

16

ChessSquare

- colour : String
…

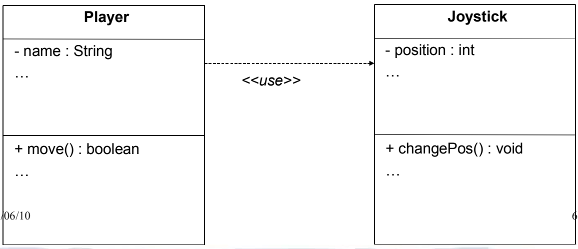
+ getColour() : String
…

Relationship 3: Composition

- “*Composition* represents a very strong relationship between classes, to the point of containment. Composition is used to capture a whole-part relationship. The “part” piece of the relationship can be involved in only one composition relationship at any given time.”
  - UML 2.0 In a Nutshell
- Typically read as a “is part of” relationship
- Indicated by filled diamond next to owner class and solid line to owned class

Relationship 4: Dependency

- **Dependency**:
  - A relationship describing that a change to the target element may require a change in the source element.
- Example:



Relationship 4: Dependency

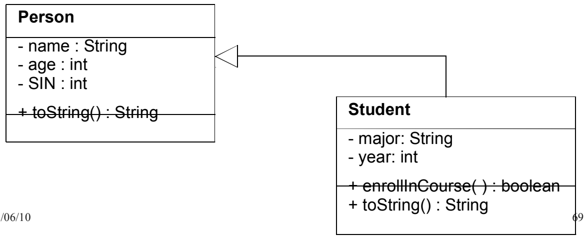
- “The weakest relationship between classes is a *dependency* relationship. Dependency between classes means that one class uses, or has knowledge of, another class. It is typically a transient relationship, meaning a dependent class briefly interacts with the target class but typically doesn't retain a relationship with it for any real length of time.”
  - UML 2.0 In a Nutshell

Relationship 4: Dependency

- Typically read as a “uses a” relationship
- Indicated by a dashed line with an arrow pointing from the dependent class to the class that is used.

Relationship 5: Generalization

- **Generalization**:
  - An **inheritance** relationship, where a subclass is a specialized form of the superclass
- Example:



Person

- name : String
- age : int
- SIN : int

+ toString() : String

Student

- major: String
- year: int

+ enrollInCourse( ) : boolean
+ toString() : String

Relationship 5: Generalization

- "A *generalization* relationship conveys that the target of the relationship is a general, or less specific, version of the source class"
  - UML 2.0 In a Nutshell
- Typically read as a "is a" relationship
- Indicated by a solid line with a closed arrow, pointing from the specific class to the general class

Relationship 5: Generalization

- Note: UML allows for multiple inheritance, but Java does not
- If we want to simulate multiple inheritance, we can use *interfaces*

Heuristics for Finding Classes

- We usually start with the problem description and map each *relevant* word as follows:

nouns	→ classes or attributes
is/are	→ inheritance
has/have	→ aggregation or association
other verbs	→ methods
must	→ constraint
adjective	→ attribute, relation

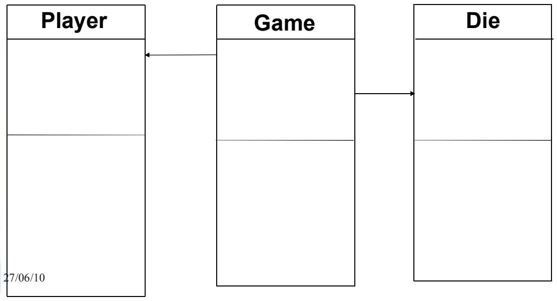
- This is called Abbott's heuristics for natural language analysis
- This is not always very accurate but it provides a good start

Simple Design Example

- Problem Description:
  We want to simulate a simple betting game. In this game, a player with money to gamble makes a bet and then rolls a single die. If a 1 is rolled, the player wins an amount equal to the bet, otherwise (s)he loses the bet.

- Let us try to identify the classes and their behaviour…..
- Nouns:
  - **game**, **player**, **money**, **bet**, **die**, **amount**, **bet**
- Verbs :
  - **gamble, makes (a bet), rolls, wins, loses**

Putting it Together



Example

```
class Car extends Vehicle
{
   . . .
   private Tire[] tires;
}
```

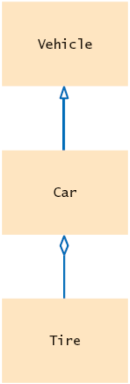


Figure 6
UML Notation for
Inheritance and Aggregation

UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance	———▷	Solid	Triangle
Interface Implementation	- - - - - - - -▷	Dotted	Triangle
Aggregation	◇———	Solid	Diamond
Dependency	- - - - - - - - ->	Dotted	Open

Printing an Invoice – Requirements

- Task: print out an invoice
- Invoice: describes the charges for a set of products in certain quantities
- Omit complexities
  - *Dates, taxes, and invoice and customer numbers*
- Print invoice
  - *Billing address, all line items, amount due*
- Line item
  - *Description, unit price, quantity ordered, total price*
- For simplicity, do not provide a user interface
- Test program: adds line items to the invoice and then prints it

Sample Invoice

```
                 I N V O I C E

 Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description          Price   Qty     Total
Toaster              29.95   3        89.85
Hair dryer           24.95   1        24.95
Car vacuum           19.99   2        39.98

AMOUNT DUE:  $154.78
```

Printing an Invoice

- Discover classes
- Nouns are possible classes

```
Invoice
Address
LineItem
Product
Description
Price
Quantity
Total
Amount Due
```

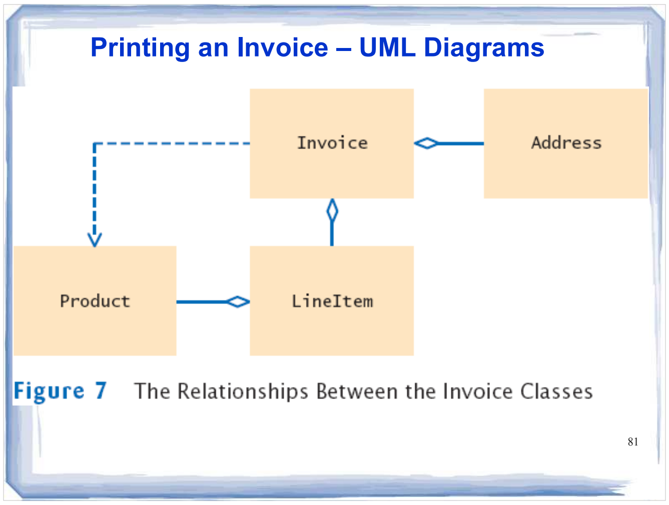
Printing an Invoice

- Analyze classes

```
Invoice
Address
LineItem // Records the product and the quantity
Product
Description // Field of the Product class
Price // Field of the Product class
Quantity // Not an attribute of a Product
Total // Computed – not stored anywhere
Amount Due // Computed – not stored anywhere
```

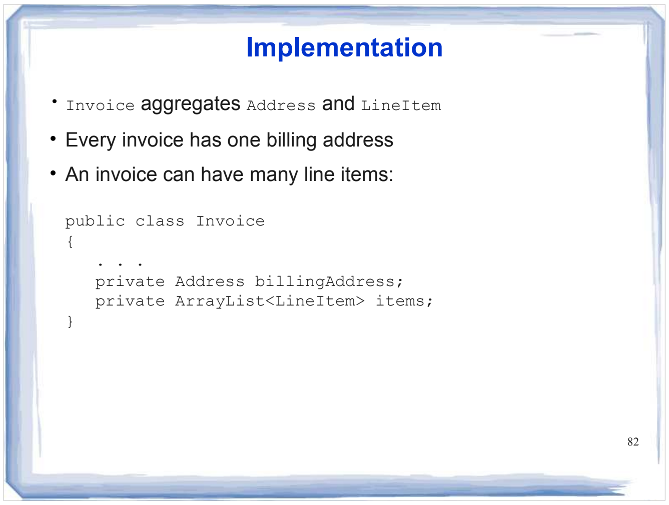
- Classes after a process of elimination

```
Invoice
Address
LineItem
Product
```



Printing an Invoice – UML Diagrams

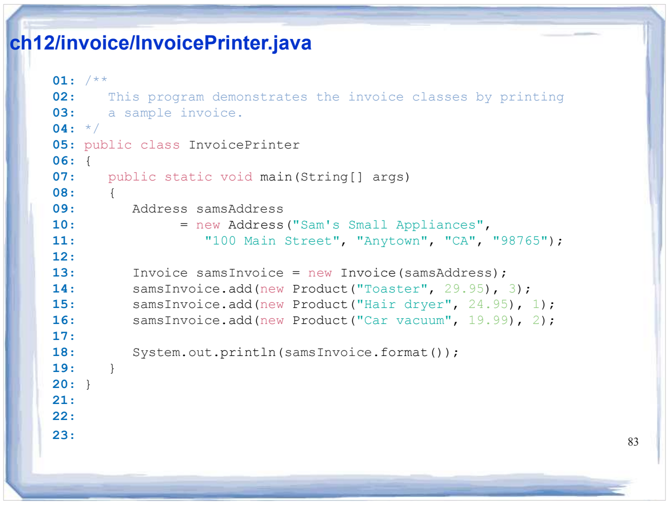
Figure 7 The Relationships Between the Invoice Classes



Implementation

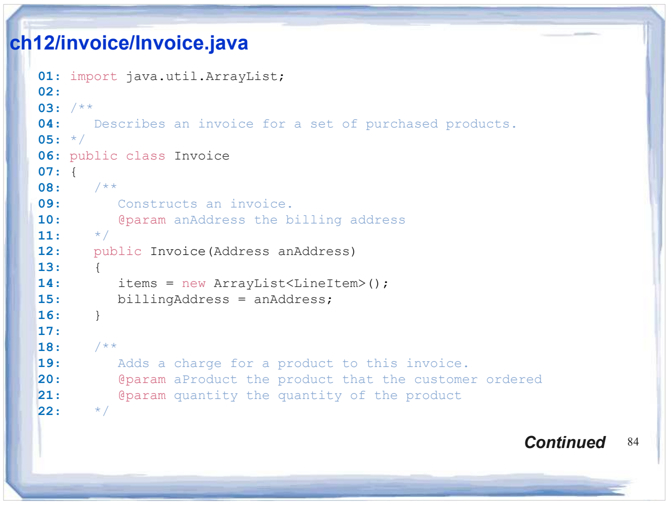
- `Invoice` aggregates `Address` and `LineItem`
- Every invoice has one billing address
- An invoice can have many line items:

```
public class Invoice
{
   . . .
   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```



ch12/invoice/InvoicePrinter.java

```
/**
   This program demonstrates the invoice classes by printing
   a sample invoice.
*/
public class InvoicePrinter
{
   public static void main(String[] args)
   {
      Address samsAddress
            = new Address("Sam's Small Appliances",
               "100 Main Street", "Anytown", "CA", "98765");

      Invoice samsInvoice = new Invoice(samsAddress);
      samsInvoice.add(new Product("Toaster", 29.95), 3);
      samsInvoice.add(new Product("Hair dryer", 24.95), 1);
      samsInvoice.add(new Product("Car vacuum", 19.99), 2);

      System.out.println(samsInvoice.format());
   }
}



```



ch12/invoice/Invoice.java

```
import java.util.ArrayList;

/**
   Describes an invoice for a set of purchased products.
*/
public class Invoice
{
   /**
      Constructs an invoice.
      @param anAddress the billing address
   */
   public Invoice(Address anAddress)
   {
      items = new ArrayList<LineItem>();
      billingAddress = anAddress;
   }

   /**
      Adds a charge for a product to this invoice.
      @param aProduct the product that the customer ordered
      @param quantity the quantity of the product
   */
```

Continued

ch12/invoice/Invoice.java (cont.)

```
   public void add(Product aProduct, int quantity)
   {
      LineItem anItem = new LineItem(aProduct, quantity);
      items.add(anItem);
   }

   /**
      Formats the invoice.
      @return the formatted invoice
   */
   public String format()
   {
      String r =  "                     I N V O I C E\n\n"
            + billingAddress.format()
            + String.format("\n\n%-30s%8s%5s%8s\n",
               "Description", "Price", "Qty", "Total");

      for (LineItem i : items)
      {
         r = r + i.format() + "\n";
      }

```

Continued

ch12/invoice/Invoice.java (cont.)

```
      r = r + String.format("\nAMOUNT DUE: $%8.2f",
getAmountDue());

      return r;
   }

   /**
      Computes the total amount due.
      @return the amount due
   */
   public double getAmountDue()
   {
      double amountDue = 0;
      for (LineItem i : items)
      {
         amountDue = amountDue + i.getTotalPrice();
      }
      return amountDue;
   }

   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```

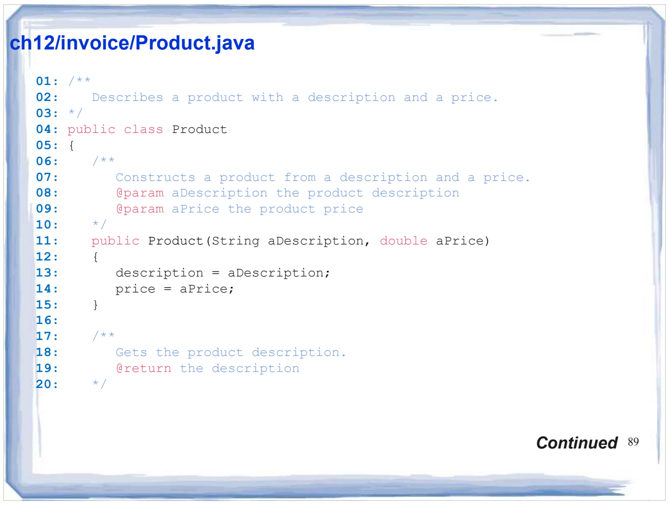
ch12/invoice/LineItem.java

```
/**
   Describes a quantity of an article to purchase.
*/
public class LineItem
{
   /**
      Constructs an item from the product and quantity.
      @param aProduct the product
      @param aQuantity the item quantity
   */
   public LineItem(Product aProduct, int aQuantity)
   {
      theProduct = aProduct;
      quantity = aQuantity;
   }

   /**
      Computes the total cost of this line item.
      @return the total price
   */
```

Continued

ch12/invoice/LineItem.java (cont.)

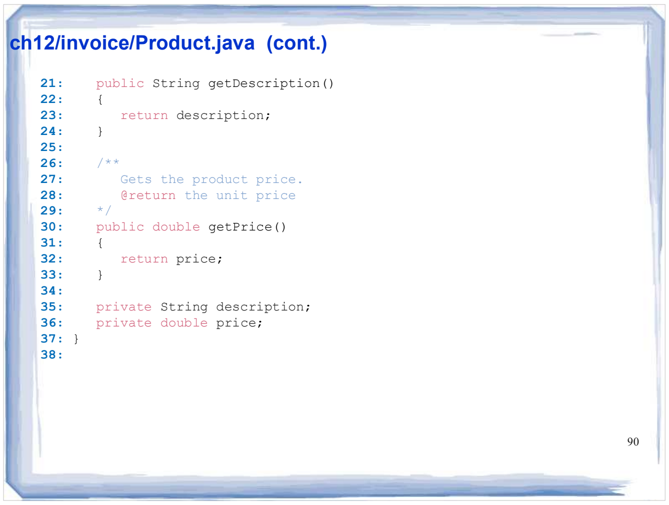
```
   public double getTotalPrice()
   {
      return theProduct.getPrice() * quantity;
   }

   /**
      Formats this item.
      @return a formatted string of this item
   */
   public String format()
   {
      return String.format("%-30s%8.2f%5d%8.2f",
            theProduct.getDescription(), theProduct.getPrice(),
            quantity, getTotalPrice());
   }

   private int quantity;
   private Product theProduct;
}
```



ch12/invoice/Product.java

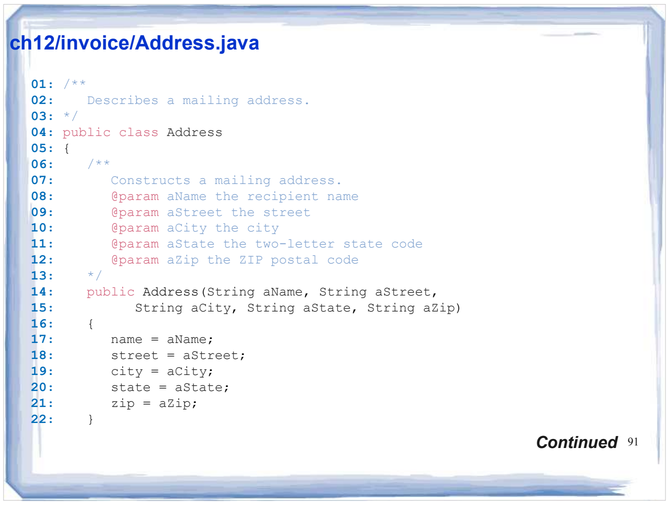
```
/**
   Describes a product with a description and a price.
*/
public class Product
{
   /**
      Constructs a product from a description and a price.
      @param aDescription the product description
      @param aPrice the product price
   */
   public Product(String aDescription, double aPrice)
   {
      description = aDescription;
      price = aPrice;
   }

   /**
      Gets the product description.
      @return the description
   */
```

Continued



ch12/invoice/Product.java (cont.)

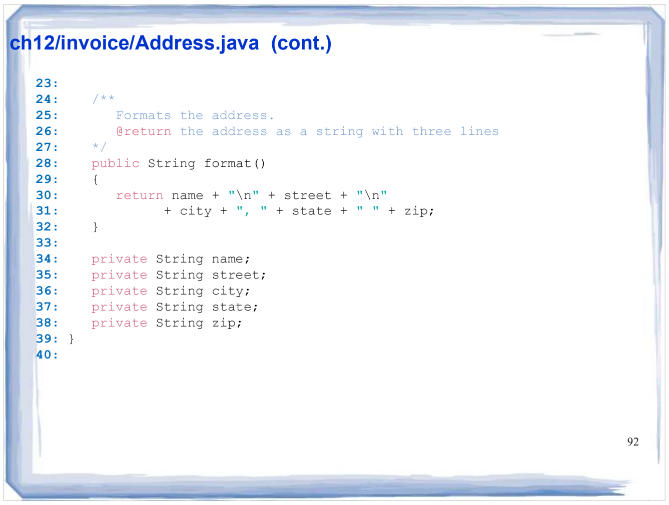
```
   public String getDescription()
   {
      return description;
   }

   /**
      Gets the product price.
      @return the unit price
   */
   public double getPrice()
   {
      return price;
   }

   private String description;
   private double price;
}
```



ch12/invoice/Address.java

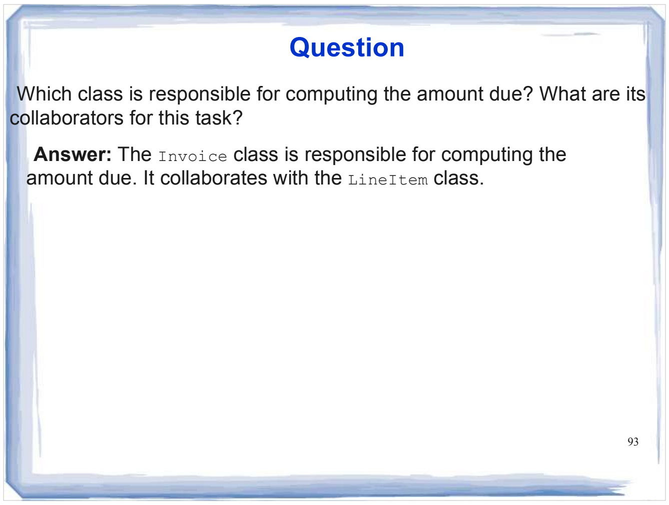
```
/**
   Describes a mailing address.
*/
public class Address
{
   /**
      Constructs a mailing address.
      @param aName the recipient name
      @param aStreet the street
      @param aCity the city
      @param aState the two-letter state code
      @param aZip the ZIP postal code
   */
   public Address(String aName, String aStreet,
         String aCity, String aState, String aZip)
   {
      name = aName;
      street = aStreet;
      city = aCity;
      state = aState;
      zip = aZip;
   }
```

Continued



ch12/invoice/Address.java (cont.)

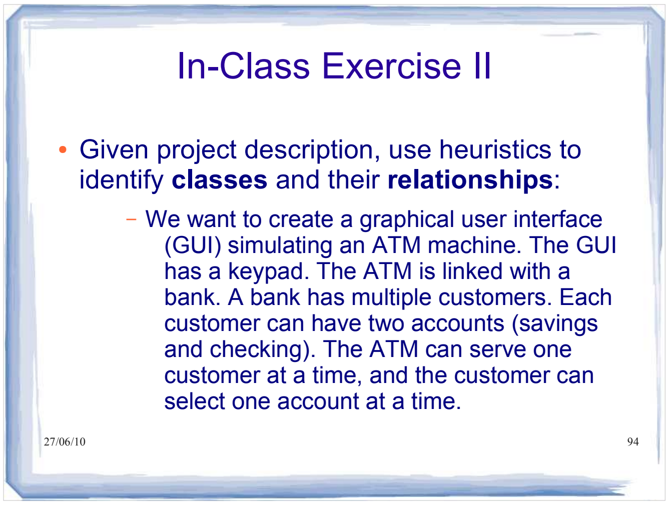
```

   /**
      Formats the address.
      @return the address as a string with three lines
   */
   public String format()
   {
      return name + "\n" + street + "\n"
            + city + ", " + state + " " + zip;
   }

   private String name;
   private String street;
   private String city;
   private String state;
   private String zip;
}
```



Question

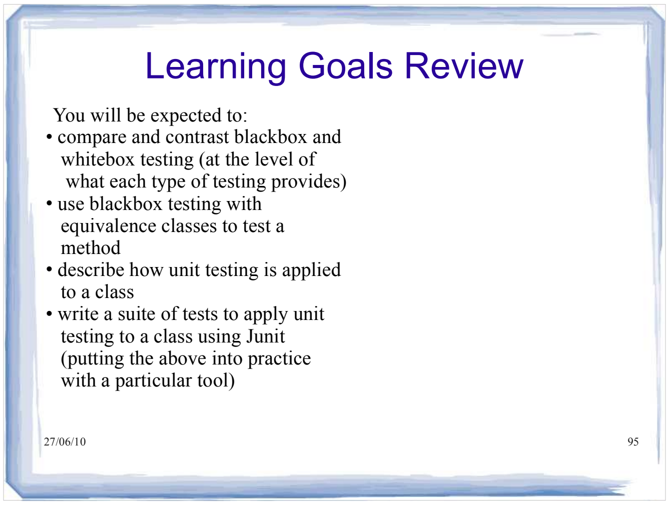
Which class is responsible for computing the amount due? What are its collaborators for this task?

Answer: The `Invoice` class is responsible for computing the amount due. It collaborates with the `LineItem` class.



In-Class Exercise II

- Given project description, use heuristics to identify **classes** and their **relationships**:
  - We want to create a graphical user interface (GUI) simulating an ATM machine. The GUI has a keypad. The ATM is linked with a bank. A bank has multiple customers. Each customer can have two accounts (savings and checking). The ATM can serve one customer at a time, and the customer can select one account at a time.



Learning Goals Review

You will be expected to:

- compare and contrast blackbox and whitebox testing (at the level of what each type of testing provides)
- use blackbox testing with equivalence classes to test a method
- describe how unit testing is applied to a class
- write a suite of tests to apply unit testing to a class using Junit (putting the above into practice with a particular tool)