

Writing Classes in Java

Selim Aksoy
Bilkent University
Department of Computer Engineering
saksoy@cs.bilkent.edu.tr

Writing Classes

- We've been using predefined classes. Now we will learn to write our own classes to define objects
- Chapter 4 focuses on:
 - class definitions
 - encapsulation and Java modifiers
 - method declaration, invocation, and parameter passing
 - method overloading
 - method decomposition

Summer 2004

CS 111

2

Objects

- An object has:
 - *state* - descriptive characteristics
 - *behaviors* - what it can do (or what can be done to it)
- For example, consider a coin that can be flipped so that its face shows either "heads" or "tails"
- The state of the coin is its current face (heads or tails)
- The behavior of the coin is that it can be flipped
- Note that the behavior of the coin might change its state

Summer 2004

CS 111

3

Classes

- A *class* is a blueprint of an object
- It is the model or pattern from which objects are created
- For example, the `String` class is used to define `String` objects
- Each `String` object contains specific characters (its state)
- Each `String` object can perform services (behaviors) such as `toUpperCase`

Summer 2004

CS 111

4

Classes

- The `String` class was provided for us by the Java standard class library
- But we can also write our own classes that define specific objects that we need
- For example, suppose we want to write a program that simulates the flipping of a coin
- We can write a `Coin` class to represent a coin object

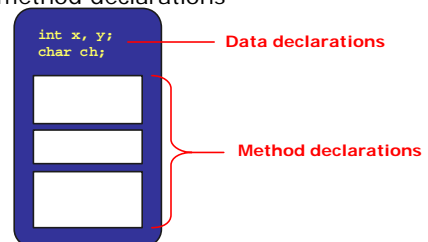
Summer 2004

CS 111

5

Classes

- A class contains data declarations and method declarations



Summer 2004

CS 111

6

The Coin Class

- In our `Coin` class we could define the following data:
 - `face`, an integer that represents the current face
 - `HEADS` and `TAILS`, integer constants that represent the two possible states
- We might also define the following methods:
 - a `Coin` constructor, to initialize the object
 - a `flip` method, to flip the coin
 - a `isHeads` method, to determine if the current face is heads
 - a `toString` method, to return a string description for printing

Summer 2004

CS 111

7

Example

```
import java.util.Random;

public class Coin
{
    private final int HEADS = 0;
    private final int TAILS = 1;
    private int face;

    // Sets up the coin by flipping it initially.
    public Coin ()
    {
        flip();
    }

    // Flips the coin by randomly choosing a face value.
    public void flip ()
    {
        face = (int) (Math.random() * 2);
    }

    // Returns true if the current face of the coin is heads.
    public boolean isHeads ()
    {
        return (face == HEADS);
    }

    // Returns the current face of the coin as a string.
    public String toString()
    {
        String faceName;
        if (face == HEADS)
            faceName = "Heads";
        else
            faceName = "Tails";
        return faceName;
    }
}
```

Summer 2004

CS 111

8

Example

```
public class CountFlips
{
    // Flips a coin multiple times and counts the number of heads
    // and tails that result.
    public static void main (String[] args)
    {
        final int NUM_FLIPS = 1000;
        int heads = 0, tails = 0;

        Coin myCoin = new Coin(); // instantiate the Coin object

        for (int count=1; count <= NUM_FLIPS; count++)
        {
            myCoin.flip();
            if (myCoin.isHeads())
                heads++;
            else
                tails++;
        }

        System.out.println ("The number flips: " + NUM_FLIPS);
        System.out.println ("The number of heads: " + heads);
        System.out.println ("The number of tails: " + tails);
    }
}
```

Summer 2004

CS 111

9

The Coin Class

- Note that the `CountFlips` program did not use the `toString` method
- A program will not necessarily use every service provided by an object
- Once the `Coin` class has been defined, we can use it again in other programs as needed

Summer 2004

CS 111

10

Data Scope

- The *scope* of data is the area in a program in which that data can be used (referenced)
- Data declared at the class level can be used by all methods in that class
- Data declared within a method can be used only in that method
- Data declared within a method is called *local data*

Summer 2004

CS 111

11

Instance Data

- The `face` variable in the `Coin` class is called *instance data* because each instance (object) of the `Coin` class has its own
- A class declares the type of the data, but it does not reserve any memory space for it
- Every time a `Coin` object is created, a new `face` variable is created as well
- The objects of a class share the method definitions, but each has its own data space
- That is the only way two objects can have different states

Summer 2004

CS 111

12

Example

```
public class FlipRace
{
    //=====
    // Flip two coins until one of them comes up heads three times
    // in a row.
    //=====
    public static void main (String[] args)
    {
        final int GOAL = 3;
        int count1 = 0, count2 = 0;

        // Create two separate coin objects
        Coin coin1 = new Coin();
        Coin coin2 = new Coin();

        while (count1 < GOAL && count2 < GOAL)
        {
            coin1.flip();
            coin2.flip();
            // Print the flip results (uses Coin's toString method)
            System.out.print ("Coin 1: " + coin1);
            System.out.print ("Coin 2: " + coin2);
            // Increment or reset the counters
            count1 = (coin1.isHeads()) ? count1+1 : 0;
            count2 = (coin2.isHeads()) ? count2+1 : 0;
        }

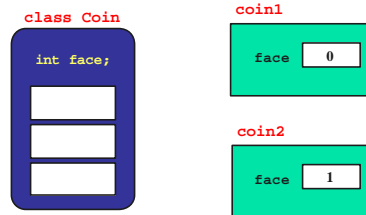
        // Determine the winner
        if (count1 < GOAL)
            System.out.println ("Coin 2 Wins!");
        else if (count2 < GOAL)
            System.out.println ("Coin 1 Wins!");
        else
            System.out.println ("It's a TIE!");
    }
}
```

Summer 2004

CS 111

13

Instance Data



Summer 2004

CS 111

14

UML Diagrams

- UML stands for the Unified Modeling Language
- UML diagrams show relationships among classes and objects
- A UML class diagram consists of one or more classes, each with sections for the class name, attributes, and methods
- Lines between classes represent associations
- Associations can show multiplicity

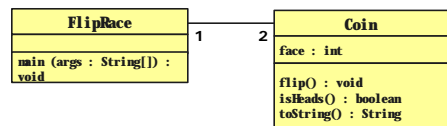
Summer 2004

CS 111

15

UML Class Diagrams

- A UML class diagram for the FlipRace program:



Summer 2004

CS 111

16

UML Diagrams

- A UML object diagram consists of one or more instantiated objects.
- It is a snapshot of the objects during an executing program, showing data values



Summer 2004

CS 111

17

Encapsulation

- We can take one of two views of an object:
 - internal - the variables the object holds and the methods that make the object useful
 - external - the services that an object provides and how the object interacts
- From the external view, an object is an encapsulated entity, providing a set of specific services
- These services define the interface to the object
- Recall from Chapter 2 that an object is an abstraction, hiding details from the rest of the system

Summer 2004

CS 111

18

Encapsulation

- An object should be *self-governing*
- Any changes to the object's state (its variables) should be made only by that object's methods
- We should make it difficult, if not impossible, to access an object's variables other than via its methods
- The user, or *client*, of an object can request its services, but it should not have to be aware of how those services are accomplished

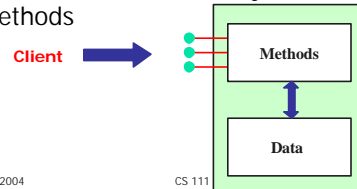
Summer 2004

CS 111

19

Encapsulation

- An encapsulated object can be thought of as a *black box*
- Its inner workings are hidden to the client, which invokes only the interface methods



Summer 2004

CS 111

20

Visibility Modifiers

- In Java, we accomplish encapsulation through the appropriate use of *visibility modifiers*
- A *modifier* is a Java reserved word that specifies particular characteristics of a method or data value
- We have used the modifier `final` to define a constant
- Java has three visibility modifiers: `public`, `protected`, and `private`
- The `protected` modifier involves inheritance, which we will discuss in CS 112

Summer 2004

CS 111

21

Visibility Modifiers

- Members of a class that are declared with *public visibility* can be accessed from anywhere
- Public variables violate encapsulation
- Members of a class that are declared with *private visibility* can only be accessed from inside the class
- Members declared without a visibility modifier have *default visibility* and can be accessed by any class in the same package
- Java modifiers are discussed in detail in Appendix F

Summer 2004

CS 111

22

Visibility Modifiers

- Methods that provide the object's services are usually declared with public visibility so that they can be invoked by clients
- Public methods are also called *service methods*
- A method created simply to assist a service method is called a *support method*
- Since a support method is not intended to be called by a client, it should not be declared with public visibility

Summer 2004

CS 111

23

Visibility Modifiers

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Summer 2004

CS 111

24

Driver Programs

- A *driver program* drives the use of other, more interesting parts of a program
- Driver programs are often used to test other parts of the software
- The Banking class contains a main method that drives the use of the Account class, exercising its services

Summer 2004

CS 111

25

Example

```
import java.text.NumberFormat;

public class Account
{
    private NumberFormat fmt = NumberFormat.getCurrencyInstance();
    private final double RATE = 0.08; // interest rate of 8%
    private long acctNumber;
    private double balance;
    private String name;

    //-----
    // Sets up the account by defining its owner, account number,
    // and initial balance.
    //-----
    public Account (String owner, long account, double initial)
    {
        name = owner;
        acctNumber = account;
        balance = initial;
    }

    //-----
    // Validates the transaction, then deposits the specified amount
    // into the account. Returns the new balance.
    //-----
    public double deposit (double amount)
    {
        if (amount < 0) // deposit value is negative
        {
            System.out.println ();
            System.out.println ("Error: Deposit amount is invalid.");
            System.out.println (acctNumber + " " + fmt.format(amount));
        }
        else
            balance = balance + amount;

        return balance;
    }
}
```

Summer 2004

CS 111

26

Example

```
//-----
// Validates the transaction, then withdraws the specified amount
// from the account. Returns the new balance.
//-----
public double withdraw (double amount, double fee)
{
    amount += fee;

    if (amount < 0) // withdraw value is negative
    {
        System.out.println ();
        System.out.println ("Error: Withdraw amount is invalid.");
        System.out.println ("Account: " + acctNumber);
        System.out.println ("Requested: " + fmt.format(amount));
    }
    else if (amount > balance) // withdraw value exceeds balance
    {
        System.out.println ();
        System.out.println ("Error: Insufficient funds.");
        System.out.println ("Account: " + acctNumber);
        System.out.println ("Requested: " + fmt.format(amount));
        System.out.println ("Available: " + fmt.format(balance));
    }
    else
        balance = balance - amount;

    return balance;
}
}
```

Summer 2004

CS 111

27

Example

```
//-----
// Adds interest to the account and returns the new balance.
//-----
public double addInterest ()
{
    balance += (balance * RATE);
    return balance;
}

//-----
// Returns the current balance of the account.
//-----
public double getBalance ()
{
    return balance;
}

//-----
// Returns the account number.
//-----
public long getAccountNumber ()
{
    return acctNumber;
}

//-----
// Returns a one-line description of the account as a string.
//-----
public String toString ()
{
    return (acctNumber + "\t" + name + "\t" + fmt.format(balance));
}
}
```

Summer 2004

CS 111

28

Example

```
public class Banking
{
    //-----
    // Creates some bank accounts and requests various services.
    //-----
    public static void main (String[] args)
    {
        Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
        Account acct2 = new Account ("Jane Smith", 69713, 40.00);
        Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

        acct1.deposit (25.85);
        double smithBalance = acct2.deposit (500.00);
        System.out.println ("Smith balance after deposit: " +
            smithBalance);
        System.out.println ("Smith balance after withdrawal: " +
            acct2.withdraw (430.75, 1.50));

        acct3.withdraw (800.00, 0.0); // exceeds balance
        acct1.addInterest();
        acct2.addInterest();
        acct3.addInterest();

        System.out.println ();
        System.out.println (acct1);
        System.out.println (acct2);
        System.out.println (acct3);
    }
}
```

Summer 2004

CS 111

29

Method Declarations

- A *method declaration* specifies the code that will be executed when the method is invoked (or called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined

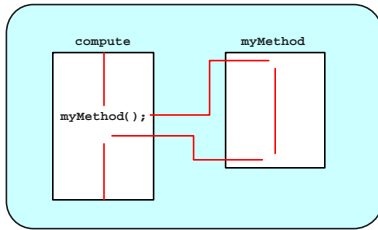
Summer 2004

CS 111

30

Method Control Flow

- The called method can be within the same class, in which case only the method name is needed



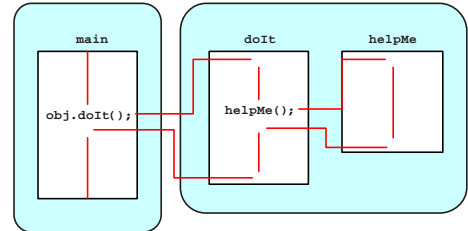
Summer 2004

CS 111

31

Method Control Flow

- The called method can be part of another class or object



Summer 2004

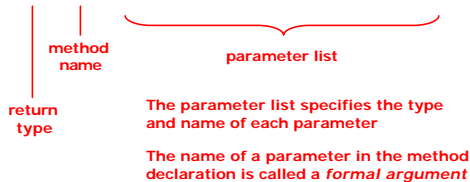
CS 111

32

Method Header

- A method declaration begins with a *method header*

```
char calc (int num1, int num2, String message)
```



Summer 2004

CS 111

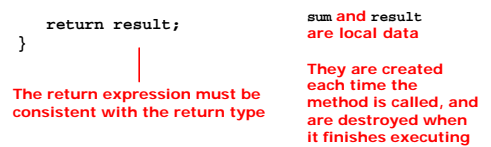
33

Method Body

- The method header is followed by the *method body*

```
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
}
```



Summer 2004

CS 111

34

The return Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a *void* return type
- A *return statement* specifies the value that will be returned


```
return expression;
```
- Its expression must conform to the return type

Summer 2004

CS 111

35

Parameters

- Each time a method is called, the *actual parameters* in the invocation are copied into the formal parameters

```
ch = obj.calc (25, count, "Hello");
```

```
char calc (int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt (sum);

    return result;
}
```

Summer 2004

CS 111

36

Local Data

- Local variables can be declared inside a method
- The formal parameters of a method create *automatic local variables* when the method is invoked
- When the method finishes, all local variables are destroyed (including the formal parameters)
- Keep in mind that instance variables, declared at the class level, exists as long as the object exists
- Any method in the class can refer to instance data

Summer 2004

CS 111

37

Constructors Revisited

- Recall that a constructor is a special method that is used to initialize a newly created object
- When writing a constructor, remember that:
 - it has the same name as the class
 - it does not return a value
 - it has no return type, not even `void`
 - it typically sets the initial values of instance variables
- The programmer does not have to define a constructor for a class

Summer 2004

CS 111

38

Overloading Methods

- *Method overloading* is the process of using the same method name for multiple methods
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters
- The compiler determines which version of the method is being invoked by analyzing the parameters
- The return type of the method is not part of the signature


Summer 2004

CS 111

39

Overloading Methods

Version 1	Version 2
<pre>float tryMe (int x) { return x + .375; }</pre>	<pre>float tryMe (int x, float y) { return x*y; }</pre>


Invocation

```
result = tryMe (25, 4.32)
```

Summer 2004

CS 111

40

Overloaded Methods

- The `println` method is overloaded:
`println(String s)`
`println(int i)`
`println(double d)`
and so on...
- The following lines invoke different versions of the `println` method:
`System.out.println("The total is:");`
`System.out.println(total);`

Summer 2004

CS 111

41

Overloading Methods

- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Summer 2004

CS 111

42

Example

```
public class Die
{
    private final int MIN_FACES = 4;
    private int numFaces; // number of sides on the die
    private int faceValue; // current value showing on the die

    //-----
    // Defaults to a six-sided die. Initial face value is 1.
    //-----
    public Die ()
    {
        numFaces = 6;
        faceValue = 1;
    }

    //-----
    // Explicitly sets the size of the die. Defaults to a size of
    // six if the parameter is invalid. Initial face value is 1.
    //-----
    public Die (int faces)
    {
        if (faces < MIN_FACES)
            numFaces = 6;
        else
            numFaces = faces;
        faceValue = 1;
    }

    //-----
    // Rolls the die and returns the result.
    //-----
    public int roll ()
    {
        faceValue = (int) (Math.random() * numFaces) + 1;
        return faceValue;
    }

    //-----
    // Returns the current die value.
    //-----
    public int getFaceValue ()
    {
        return faceValue;
    }
}
```

Summer 2004

CS 111

43

Example

```
public class SnakeEyes
{
    //-----
    // Creates two die objects, then rolls both dice a set number of
    // times, counting the number of snake eyes that occur.
    //-----
    public static void main (String[] args)
    {
        final int ROLLS = 500;
        int snakeEyes = 0, num1, num2;

        Die die1 = new Die(); // creates a six-sided die
        Die die2 = new Die(20); // creates a twenty-sided die

        for (int roll = 1; roll <= ROLLS; roll++)
        {
            num1 = die1.roll();
            num2 = die2.roll();
            if (num1 == 1 && num2 == 1) // check for snake eyes
                snakeEyes++;
        }

        System.out.println ("Number of rolls: " + ROLLS);
        System.out.println ("Number of snake eyes: " + snakeEyes);
        System.out.println ("Ratio: " + (float) snakeEyes/ROLLS);
    }
}
```

Summer 2004

CS 111

44

Method Decomposition

- A method should be relatively small, so that it can be understood as a single entity
- A potentially large method should be decomposed into several smaller methods as needed for clarity
- A service method of an object may call one or more support methods to accomplish its goal
- Support methods could call other support methods if appropriate

Summer 2004

CS 111

45

Class Diagrams Revisited

- In a UML class diagram, public members can be preceded by a plus sign
- Private members are preceded by a minus sign

Summer 2004

CS 111

46

Object Relationships

- Objects can have various types of relationships to each other
- A general *association*, as we've seen in UML diagrams, is sometimes referred to as a *use relationship*
- A general association indicates that one object (or class) uses or refers to another object (or class) in some way
- We could even annotate an association line in a UML diagram to indicate the nature of the relationship



Summer 2004

CS 111

47

Object Relationships

- Some use associations occur between objects of the same class
- For example, we might add two Rational number objects together as follows:
$$r3 = r1.add(r2);$$
- One object ($r1$) is executing the method and another ($r2$) is passed as a parameter

Summer 2004

CS 111

48

Example

```
//=====
// Rational.java Author: Lewis & Loftis
// Represents one rational number with a numerator and denominator.
//=====
public class Rational
{
    private int numerator, denominator;

    //=====
    // Sets up the rational number by specifying a numerator and denominator
    // public Rational (int num, int denom)
    // {
    //     if (denom == 0)
    //     {
    //         // Make the denominator "safe" the sign
    //         if (denom < 0)
    //         {
    //             num = -num;
    //             denom = -denom;
    //         }
    //         // Set up the rational number
    //         numerator = num;
    //         denominator = denom;
    //         reduce();
    //     }
    // }

    // Returns the numerator of this rational number.
    public int getNumerator ()
    {
        return numerator;
    }

    // Returns the denominator of this rational number.
    public int getDenominator ()
    {
        return denominator;
    }
}
```

Summer 2004

CS 111

49

Example

```
//=====
// Returns the reciprocal of this rational number.
// public Rational reciprocal ()
// {
//     return new Rational (denominator, numerator);
// }

//=====
// Adds this rational number to the one passed as a parameter.
// A common denominator is found by multiplying the individual
// denominators.
// public Rational add (Rational op2)
// {
//     int commonDenominator = denominator * op2.getDenominator();
//     int numerator1 = numerator * op2.getDenominator();
//     int numerator2 = op2.getNumerator() * denominator;
//     int sum = numerator1 + numerator2;
//     return new Rational (sum, commonDenominator);
// }

//=====
// Subtracts the rational number passed as a parameter from this
// rational number.
// public Rational subtract (Rational op2)
// {
//     int commonDenominator = denominator * op2.getDenominator();
//     int numerator1 = numerator * op2.getDenominator();
//     int numerator2 = op2.getNumerator() * denominator;
//     int difference = numerator1 - numerator2;
//     return new Rational (difference, commonDenominator);
// }
//=====
```

Summer 2004

CS 111

50

Example

```
//=====
// Multiplies this rational number by the one passed as a
// parameter.
// public Rational multiply (Rational op2)
// {
//     int num = numerator * op2.getNumerator();
//     int denom = denominator * op2.getDenominator();
//     return new Rational (num, denom);
// }

// Divides this rational number by the one passed as a parameter
// by multiplying by the reciprocal of the second rational.
// public Rational divide (Rational op2)
// {
//     return multiply (op2.reciprocal());
// }

// Determines if this rational number is equal to the one passed
// as a parameter. Assumes they are both reduced.
// public boolean equals (Rational op2)
// {
//     return (numerator == op2.getNumerator() &&
//             denominator == op2.getDenominator());
// }
//=====
```

Summer 2004

CS 111

51

Example

```
//=====
// Returns this rational number as a string.
// public String toString ()
// {
//     String result;
//     if (numerator == 0)
//     {
//         result = "0";
//     }
//     else
//     {
//         result = numerator + "/" + denominator;
//     }
//     return result;
// }

// Reduces this rational number by dividing both the numerator
// and the denominator by their greatest common divisor.
// private void reduce ()
// {
//     if (numerator != 0)
//     {
//         int common = gcd (Math.abs(numerator), denominator);
//         numerator = numerator / common;
//         denominator = denominator / common;
//     }
// }

// Computes and returns the greatest common divisor of the two
// positive parameters. Uses Euclid's algorithm.
// private int gcd (int num1, int num2)
// {
//     while (num1 != num2)
//     {
//         if (num1 > num2)
//             num1 = num1 - num2;
//         else
//             num2 = num2 - num1;
//     }
//     return num1;
// }
//=====
```

Summer 2004

CS 111

52

Example

```
public class RationalNumbers
{
    //=====
    // Creates some rational number objects and performs various
    // operations on them.
    //=====
    public static void main (String[] args)
    {
        Rational r1 = new Rational (6, 8);
        Rational r2 = new Rational (1, 3);
        Rational r3, r4, r5, r6, r7;

        System.out.println ("First rational number: " + r1);
        System.out.println ("Second rational number: " + r2);

        if (r1.equals(r2))
            System.out.println ("r1 and r2 are equal.");
        else
            System.out.println ("r1 and r2 are NOT equal.");

        r3 = r1.reciprocal();
        System.out.println ("The reciprocal of r1 is: " + r3);

        r4 = r1.add(r2);
        r5 = r1.subtract(r2);
        r6 = r1.multiply(r2);
        r7 = r1.divide(r2);

        System.out.println ("r1 + r2: " + r4);
        System.out.println ("r1 - r2: " + r5);
        System.out.println ("r1 * r2: " + r6);
        System.out.println ("r1 / r2: " + r7);
    }
}
```

Summer 2004

CS 111

53

The static Modifier

- Static methods can be invoked through the class name rather than through a particular object
- To write a static method, we apply the static modifier to the method definition
- The static modifier can be applied to variables as well
- It associates a variable or method with the class rather than with an object

Summer 2004

CS 111

54

Static Variables and Methods

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists
- All objects created from the class share static variables
- Changing the value of a static variable in one object changes it for all others
- Static methods cannot reference instance variables, because instance variables don't exist until an object exists

Summer 2004

CS 111

55

Aggregation

- An *aggregate object* is an object that contains references to other objects
- For example, an Account object contains a reference to a String object (the owner's name)
- An aggregate object represents a *has-a* relationship
- A bank account *has a* name
- Likewise, a student may have one or more addresses

Summer 2004

CS 111

56

Example

```
public class Address
{
    private String streetAddress, city, state;
    private long zipCode;

    // Sets up this Address object with the specified data.
    public Address (String street, String town, String st, long zip)
    {
        streetAddress = street;
        city = town;
        state = st;
        zipCode = zip;
    }

    // Returns this Address object as a string.
    public String toString()
    {
        String result;

        result = streetAddress + "\n";
        result += city + ", " + state + ", " + zipCode;

        return result;
    }
}
```

Summer 2004

CS 111

57

Example

```
public class Student
{
    private String firstName, lastName;
    private Address homeAddress, schoolAddress;

    // Sets up this Student object with the specified initial values.
    public Student (String first, String last, Address home,
        Address school)
    {
        firstName = first;
        lastName = last;
        homeAddress = home;
        schoolAddress = school;
    }

    // Returns this Student object as a string.
    public String toString()
    {
        String result;

        result = firstName + " " + lastName + "\n";
        result += "Home Address\n" + homeAddress + "\n";
        result += "School Address\n" + schoolAddress;

        return result;
    }
}
```

Summer 2004

CS 111

58

Example

```
.....
StudentBody.java      Author: Lewis/Loftus
// Demonstrates the use of an aggregate class.
//.....
public class StudentBody
{
    // Creates some Address and Student objects and prints them.
    //.....
    public static void main (String[] args)
    {
        Address school = new Address ("800 Lancaster Ave.", "Villanova",
            "PA", 19085);

        Address jHome = new Address ("21 Jump Street", "Lynchburg",
            "VA", 24551);
        Student john = new Student ("John", "Smith", jHome, school);
        Address mHome = new Address ("123 Main Street", "Euclid", "OH",
            44132);
        Student marsha = new Student ("Marsha", "Jones", mHome, school);

        System.out.println (john);
        System.out.println (john);
        System.out.println (marsha);
    }
}
```

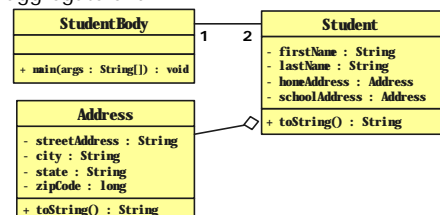
Summer 2004

CS 111

59

Aggregation in UML

- An aggregation association is shown in a UML class diagram using an open diamond at the aggregate end



Summer 2004

CS 111

60