

Review (cont.)

Program Development

- The creation of software involves four basic activities:
 - establishing the requirements
 - creating a design
 - implementing the code
 - testing the implementation
- These activities are not strictly linear – they overlap and interact

Requirements

- Software requirements specify the tasks that a program must accomplish
 - what to do, not how to do it
- Often an initial set of requirements is provided, but they should be critiqued and expanded
- It is difficult to establish detailed, unambiguous, and complete requirements
- Careful attention to the requirements can save significant time and expense in the overall project

Design

- A software design specifies how a program will accomplish its requirements
- A software design specifies how the solution can be broken down into manageable pieces and what each piece will do
- An object-oriented design determines which classes and objects are needed, and specifies how they will interact
- Low level design details include how individual methods will accomplish their tasks

Implementation

- Implementation is the process of translating a design into source code
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

Testing

- Testing attempts to ensure that the program will solve the intended problem under all the constraints specified in the requirements
- A program should be thoroughly tested with the goal of finding errors
- Debugging is the process of determining the cause of a problem and fixing it

Identifying Classes and Objects

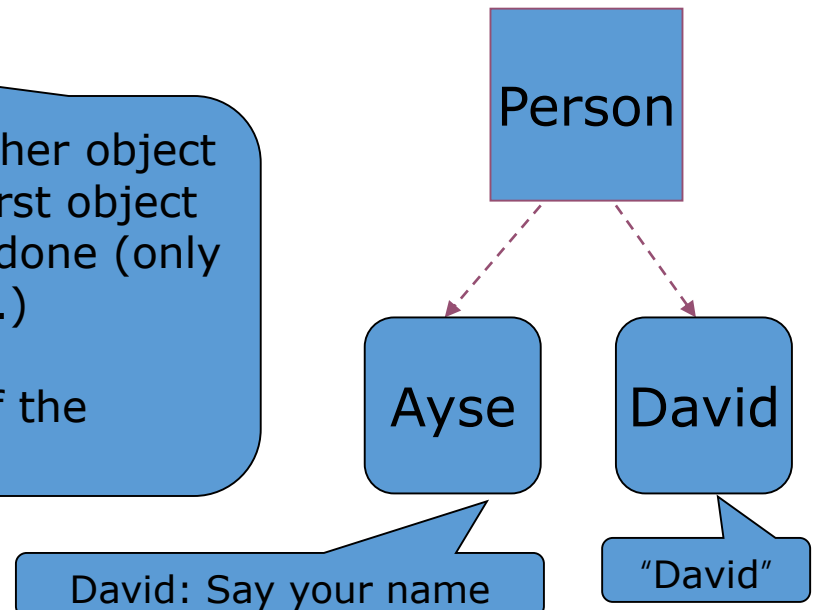
Java OOP Software

Created (instantiated)
from class definitions

- Software System
 - Set of objects
 - Which interact with each other

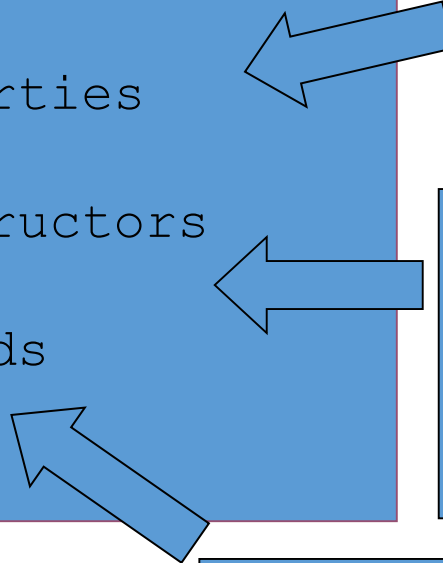
One object will send a message to another object asking it to do a particular task. The first object does not need to know how the task is done (only how to request that it be done.)

This corresponds to calling one of the second object's methods!



Coding Java Classes

```
// header  
  
public class Person {  
  
    // properties  
  
    // constructors  
  
    // methods  
  
}
```

A diagram with three blue arrows pointing from the right towards the main code block. The top arrow points to the 'properties' section, the middle arrow points to the 'constructors' section, and the bottom arrow points to the 'methods' section.

```
String    name;  
int       age;  
double    salary;  
String    comments;
```

```
public Person( String  theName,  
               int     theAge ) {  
    name = theName;  
    age  = theAge;  
    comments = "";  
}
```

```
public void sayName() {  
    System.out.println( name );  
}
```

Coding Java Classes

```
public String getName() {  
    return name;  
}
```

```
public String getComments() {  
    return comments;  
}
```

```
public void setComments( String someText) {  
    comments = someText;  
}
```

```
public void increaseAge() {  
    age = age + 1;  
}
```

```
public double getNetSalary() {  
    double netSalary;  
    netSalary = salary - TAX;  
    return netSalary;  
}
```

"get" & "set"
methods for
some properties

Variables which are
not parameters or
properties must be
defined locally.

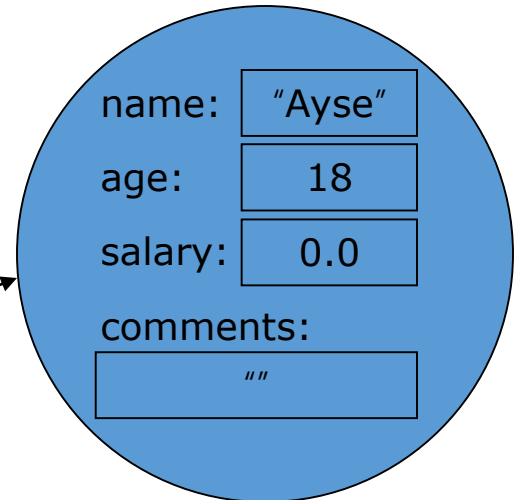
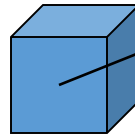
Creating & Using Objects

- Always
 - Declare variable to "hold" object
 - Create object using "new" statement
 - Call object's methods

```
Person aStudent;  
  
aStudent = new Person( "Ayse", 18);  
aStudent.sayName();
```

Put this in method
of another class,
(e.g main
method)

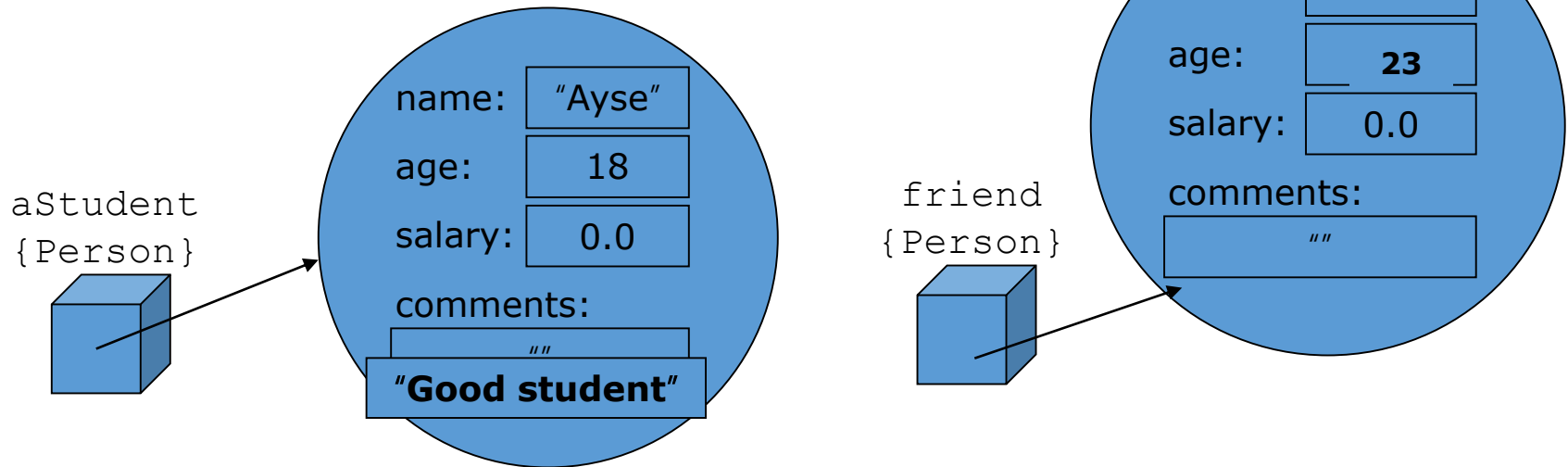
aStudent
{ Person }



Creating & Using Objects

```
Person aStudent;  
aStudent = new Person( "Ayse", 18);
```

```
Person friend;  
friend = new Person( "David", 22);
```



```
friend.increaseAge();  
aStudent.setComments( "Good student");
```

Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes and Objects

- A partial requirements document:

The **user** must be allowed to specify each **product** by its primary **characteristics**, including its **name** and **product number**. If the **bar code** does not match the **product**, then an **error** should be generated to the **message window** and entered into the **error log**. The **summary report** of all **transactions** must be structured as specified in section 7.A.

- Of course, not all nouns will correspond to a class or object in the final solution

Identifying Classes and Objects

- Remember that a class represents a group (classification) of objects with the same behaviors
- Generally, classes that represent objects should be given names that are singular nouns
- Examples: Coin, Student, Message
- A class represents the concept of one such object
- We are free to instantiate as many of each object as needed

Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class
- For example, should an employee's address be represented as a set of instance variables or as an Address object
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general Appliance class with appropriate instance data
- It all depends on the details of the problem being solved

Identifying Classes and Objects

- Part of identifying the classes we need is the process of assigning responsibilities to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- We generally use verbs for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

Static Variables and Methods

Static Class Members

- Recall that a static method is one that can be invoked through its class name
- For example, the methods of the Math class are static:
- `result = Math.sqrt(25)`
- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the static modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called class methods and static variables are sometimes called class variables
- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

`private static float price;`

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

- Because it is declared as static, the cube method can be invoked through the class name:
- `value = Helper.cube(4);`

```
public class Helper
{
    public static int cube (int num)
    {
        return num * num * num;
    }
}
```

Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the main method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

Static Class Members

- Static methods and static variables often work together
- The following example keeps track of how many Slogan objects have been created using a static variable, and makes that information available using a static method
- See SloganCounter.java
- See Slogan.java

```

//*****
//  Slogan.java          Author: Lewis/Loftus
//
//  Represents a single slogan string.
//*****

public class Slogan
{
    private String phrase;
    private static int count = 0;

    //-----
    //  Constructor: Sets up the slogan and counts the number of
    //  instances created.
    //-----
    public Slogan (String str)
    {
        phrase = str;
        count++;
    }
}

```

continue

continue

```
//-----  
// Returns this slogan as a string.  
//-----  
public String toString()  
{  
    return phrase;  
}  
  
//-----  
// Returns the number of instances of this class that have been  
// created.  
//-----  
public static int getCount ()  
{  
    return count;  
}  
}
```

```

//*****
//  SloganCounter.java          Author: Lewis/Loftus
//
//  Demonstrates the use of the static modifier.
//*****

public class SloganCounter
{
    //-----
    //  Creates several Slogan objects and prints the number of
    //  objects that were created.
    //-----
    public static void main (String[] args)
    {
        Slogan obj;

        obj = new Slogan ("Remember the Alamo.");
        System.out.println (obj);

        obj = new Slogan ("Don't Worry. Be Happy.");
        System.out.println (obj);
    }
}

```

continue

continue

```
obj = new Slogan ("Live Free or Die.");  
System.out.println (obj);  
  
obj = new Slogan ("Talk is Cheap.");  
System.out.println (obj);  
  
obj = new Slogan ("Write Once, Run Anywhere.");  
System.out.println (obj);  
  
System.out.println();  
System.out.println ("Slogans created: " + Slogan.getCount());  
}  
}
```

continue

```
obj = new Slogan("Remember the Alamo.");
System.out.println(obj);

obj = new Slogan("Don't Worry. Be Happy.");
System.out.println(obj);

obj = new Slogan("Live Free or Die.");
System.out.println(obj);

obj = new Slogan("Talk is Cheap.");
System.out.println(obj);

obj = new Slogan("Write Once, Run Anywhere.");
System.out.println(obj);

System.out.println();
System.out.println("Slogans created: " + Slogan.getCount());
}
}
```

Output

```
Remember the Alamo.
Don't Worry. Be Happy.
Live Free or Die.
Talk is Cheap.
Write Once, Run Anywhere.

Slogans created: 5
```

Why can't a static method reference an instance variable?

Why can't a static method reference an instance variable?

Because instance data is created only when an object is created.

You don't need an object to execute a static method.

And even if you had an object, which object's instance data would be referenced? (remember, the method is invoked through the class name)

Class Relationships

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Three of the most common relationships:
 - Dependency: A uses B
 - Aggregation: A has-a B
 - Inheritance: A is-a B
- Let's discuss dependency and aggregation further
- Inheritance is discussed in detail later

Dependency

- A dependency exists when one class relies on another in some way, usually by invoking the methods of the other
- We've seen dependencies in many previous examples
- We don't want numerous or complex dependencies among classes
- Nor do we want complex classes that don't depend on others
- A good design strikes the right balance

Dependency

- Some dependencies occur between objects of the same class
- A method of the class may accept an object of the same class as a parameter
- For example, the concat method of the String class takes as a parameter another String object
- `str3 = str1.concat(str2);`

Aggregation

- An aggregate is an object that is made up of other objects
- Therefore aggregation is a has-a relationship
 - A car has a chassis
- An aggregate object contains references to other objects as instance data
- This is a special kind of dependency; the aggregate relies on the objects that compose it

Aggregation

- In the following example, a Student object is composed, in part, of Address objects
- A student has an address (in fact each student has two addresses)
- See StudentBody.java
- See Student.java
- See Address.java

```

//*****
//  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 ();
        System.out.println (marsha);
    }
}

```

```

//*****
//  StudentBody.java
//
//  Demonstrates the
//*****

```

```

public class StudentB
{
    //-----
    //  Creates some A
    //-----
    public static void
    {
        Address school :
        Address jHome =
        Student john =
        Address mHome =

        Student marsha = new Student ("Marsha", "Jones", mHome, school);

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

```

Output

```

John Smith
Home Address:
21 Jump Street
Lynchburg, VA  24551
School Address:
800 Lancaster Ave.
Villanova, PA  19085

Marsha Jones
Home Address:
123 Main Street
Euclid, OH  44132
School Address:
800 Lancaster Ave.
Villanova, PA  19085

```

```

*****

```

```

*****

```

```

-----
and prints them.
-----

```

```

er Ave.", "Villanova",
;
et", "Lynchburg",
", jHome, school);
eet", "Euclid", "OH",

```

```

44132);

```

```

Student marsha = new Student ("Marsha", "Jones", mHome, school);

```

```

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

```

```
//*****  
// Student.java      Author: Lewis/Loftus  
//  
// Represents a college student.  
//*****
```

```
public class Student
```

```
{  
    private String firstName, lastName;  
    private Address homeAddress, schoolAddress;  
  
    //-----  
    // Constructor: Sets up this student with the specified values.  
    //-----  
    public Student (String first, String last, Address home,  
                    Address school)  
    {  
        firstName = first;  
        lastName = last;  
        homeAddress = home;  
        schoolAddress = school;  
    }  
}
```

continue

continue

```
//-----  
//  Returns a string description of this Student object.  
//-----  
public String toString()  
{  
    String result;  
  
    result = firstName + " " + lastName + "\n";  
    result += "Home Address:\n" + homeAddress + "\n";  
    result += "School Address:\n" + schoolAddress;  
  
    return result;  
}
```

```

//*****
//  Address.java          Author: Lewis/Loftus
//
//  Represents a street address.
//*****

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

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

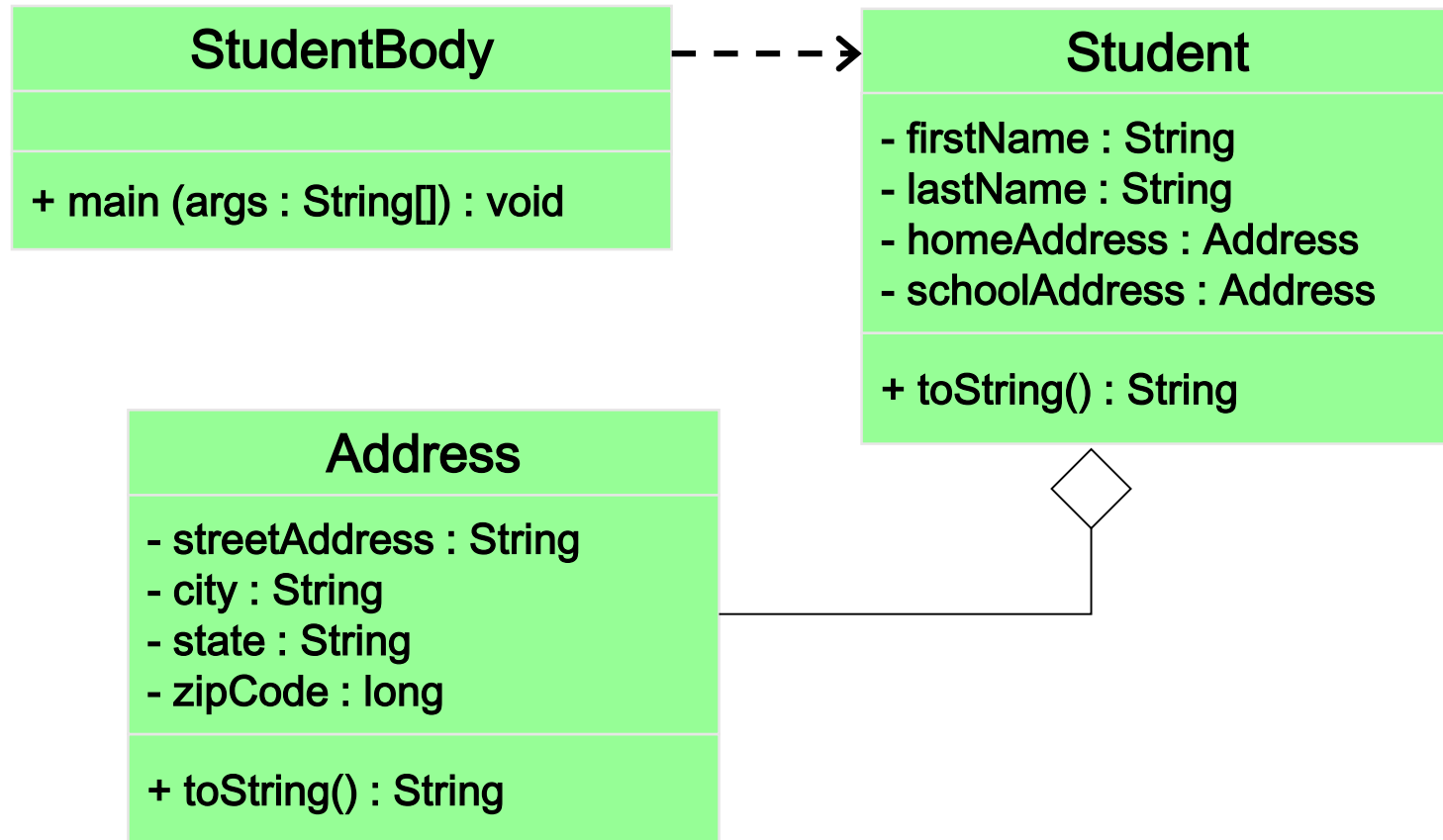
```

continue

continue

```
//-----  
//  Returns a description of this Address object.  
//-----  
public String toString()  
{  
    String result;  
  
    result = streetAddress + "\n";  
    result += city + ", " + state + "  " + zipCode;  
  
    return result;  
}  
}
```

Aggregation in UML



The this Reference

- The this reference allows an object to refer to itself
- That is, the this reference, used inside a method, refers to the object through which the method is being executed
- Suppose the this reference is used inside a method called tryMe, which is invoked as follows:
 - obj1.tryMe();
 - obj2.tryMe();
- In the first invocation, the this reference refers to obj1; in the second it refers to obj2

The this reference

- The this reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names
- The constructor of the Account class could have been written as follows:

```
public Account (String name, long acctNumber,  
                double balance)  
{  
    this.name = name;  
    this.acctNumber = acctNumber;  
    this.balance = balance;  
}
```

1- Passing this as a Parameter

```
public class MyInt {  
    private int ival;  
    public MyInt(int val) { ival=val; }  
    public boolean isGreaterThan(MyInt o2) {  
        return (ival > o2.ival);  
    }  
    public boolean isLessThan(MyInt o2) {  
        return (o2.isGreaterThan(this));  
    }  
}
```

in some other place

```
MyInt x1=new MyInt(5), x2=new MyInt(6);  
x1.isGreaterThan(x2);  
x1.isLessThan(x2);
```

Enumerated Types Revisited

Enumerated Types

- A new data type and list all possible values of that type:
- `enum Season {winter, spring, summer, fall}`
- Once established, the new type can be used to declare variables
- `Season time;`
- The only values this variable can be assigned are the ones established in the enum definition

Enumerated Types

- An enumerated type definition is a special kind of class
- The values of the enumerated type are objects of that type
- For example, fall is an object of type Season
- That's why the following assignment is valid:
- `time = Season.fall;`

Enumerated Types

- An enumerated type definition can be more interesting than a simple list of values
- Because they are like classes, we can add additional instance data and methods
- We can define an enum constructor as well
- Each value listed for the enumerated type calls the constructor
- See Season.java
- See SeasonTester.java

```
//*****  
// Season.java          Author: Lewis/Loftus  
//  
// Enumerates the values for Season.  
//*****
```

```
public enum Season  
{  
    winter ("December through February"),  
    spring ("March through May"),  
    summer ("June through August"),  
    fall ("September through November");  
  
    private String span;
```

continue

continue

```
//-----  
//  Constructor: Sets up each value with an associated string.  
//-----  
Season (String months)  
{  
    span = months;  
}  
  
//-----  
//  Returns the span message for this value.  
//-----  
public String getSpan()  
{  
    return span;  
}  
}
```

```
//*****
//  SeasonTester.java          Author: Lewis/Loftus
//
//  Demonstrates the use of a full enumerated type.
//*****

public class SeasonTester
{
    //-----
    //  Iterates through the values of the Season enumerated type.
    //-----
    public static void main (String[] args)
    {
        for (Season time : Season.values())
            System.out.println (time + "\t" + time.getSpan());
    }
}
```

Output

```
//*****  
// SeasonTest  
//  
// Demonstration  
//*****  
winter  December through February  
spring  March through May  
summer  June through August  
fall    September through November  
*****  
  
public class  
{  
    //-----  
    // Iterates through the values of the Season enumerated type.  
    //-----  
    public static void main (String[] args)  
    {  
        for (Season time : Season.values())  
            System.out.println (time + "\t" + time.getSpan());  
    }  
}
```

Enumerated Types

- Every enumerated type contains a static method called `values` that returns a list of all possible values for that type
- The list returned from `values` can be processed using a for-each loop
- An enumerated type cannot be instantiated outside of its own definition
- A carefully designed enumerated type provides a versatile and type-safe mechanism for managing data

Method Design

Methods

- A method can be:
 - an instance method (declared without using keyword static), or
 - a class method (declared using keyword static, it is also known as a static method).
- An instance method is associated with an object.
 - If it accesses an instance variable, it accesses of the copy of that instance variable in the current object.
- A static method is a class-method and there is only one copy for it.
 - All instances of that class share that single copy.
 - A static method cannot access an instance variable or an instance method.

Method Design

- As we've discussed, high-level design issues include:
 - identifying primary classes and objects
 - assigning primary responsibilities
- After establishing high-level design issues, its important to address low-level issues such as the design of key methods
- For some methods, careful planning is needed to make sure they contribute to an efficient and elegant system design

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 public service method of an object may call one or more private support methods to help it accomplish its goal
- Support methods might call other support methods if appropriate

Method Decomposition

- Let's look at an example that requires method decomposition – translating English into Pig Latin
- Pig Latin is a language in which each word is modified by moving the initial sound of the word to the end and adding "ay"
- Words that begin with vowels have the "yay" sound added on the end

book → ookbay

table → abletay

item → itemyay

chair → airchay

Method Decomposition

- The primary objective (translating a sentence) is too complicated for one method to accomplish
- Therefore we look for natural ways to decompose the solution into pieces
- Translating a sentence can be decomposed into the process of translating each word
- The process of translating a word can be separated into translating words that:
 - begin with vowels
 - begin with consonant blends (sh, cr, th, etc.)
 - begin with single consonants

Method Decomposition

- In a UML class diagram, the visibility of a variable or method can be shown using special characters
- Public members are preceded by a plus sign
- Private members are preceded by a minus sign
- See PigLatin.java
- See PigLatinTranslator.java

```
//*****
//  PigLatin.java      Author: Lewis/Loftus
//
//  Demonstrates the concept of method decomposition.
//*****

import java.util.Scanner;

public class PigLatin
{
    //-----
    //  Reads sentences and translates them into Pig Latin.
    //-----
    public static void main (String[] args)
    {
        String sentence, result, another;

        Scanner scan = new Scanner (System.in);

continue
```

continue

```
do
{
    System.out.println ();
    System.out.println ("Enter a sentence (no punctuation):");
    sentence = scan.nextLine();

    System.out.println ();
    result = PigLatinTranslator.translate (sentence);
    System.out.println ("That sentence in Pig Latin is:");
    System.out.println (result);

    System.out.println ();
    System.out.print ("Translate another sentence (y/n)? ");
    another = scan.nextLine();
}
while (another.equalsIgnoreCase("y"));
}
```

continue

do

{

Syst

Syst

sent

Syst

resu

Syst

Syst

Syst

Syst

anot

}

while

}

}

Sample Run

Enter a sentence (no punctuation):

Do you speak Pig Latin

That sentence in Pig Latin is:

oday ouyay eakspay igpay atinlay

Translate another sentence (y/n)? y

Enter a sentence (no punctuation):

Play it again Sam

That sentence in Pig Latin is:

ayplay ityay againyay amsay

Translate another sentence (y/n)? n

```
//*****
//  PigLatinTranslator.java          Author: Lewis/Loftus
//
//  Represents a translator from English to Pig Latin. Demonstrates
//  method decomposition.
//*****
```

```
import java.util.Scanner;
```

```
public class PigLatinTranslator
```

```
{
```

```
    //-----
```

```
    //  Translates a sentence of words into Pig Latin.
```

```
    //-----
```

```
    public static String translate (String sentence)
```

```
    {
```

```
        String result = "";
```

```
        sentence = sentence.toLowerCase();
```

```
        Scanner scan = new Scanner (sentence);
```

```
        while (scan.hasNext())
```

```
        {
```

```
            result += translateWord (scan.next());
```

```
            result += " ";
```

```
        }
```

continue

continue

```
    return result;
}

//-----
//  Translates one word into Pig Latin. If the word begins with a
//  vowel, the suffix "yay" is appended to the word. Otherwise,
//  the first letter or two are moved to the end of the word,
//  and "ay" is appended.
//-----
private static String translateWord (String word)
{
    String result = "";

    if (beginsWithVowel(word))
        result = word + "yay";
    else
        if (beginsWithBlend(word))
            result = word.substring(2) + word.substring(0,2) + "ay";
        else
            result = word.substring(1) + word.charAt(0) + "ay";

    return result;
}
```

continue

continue

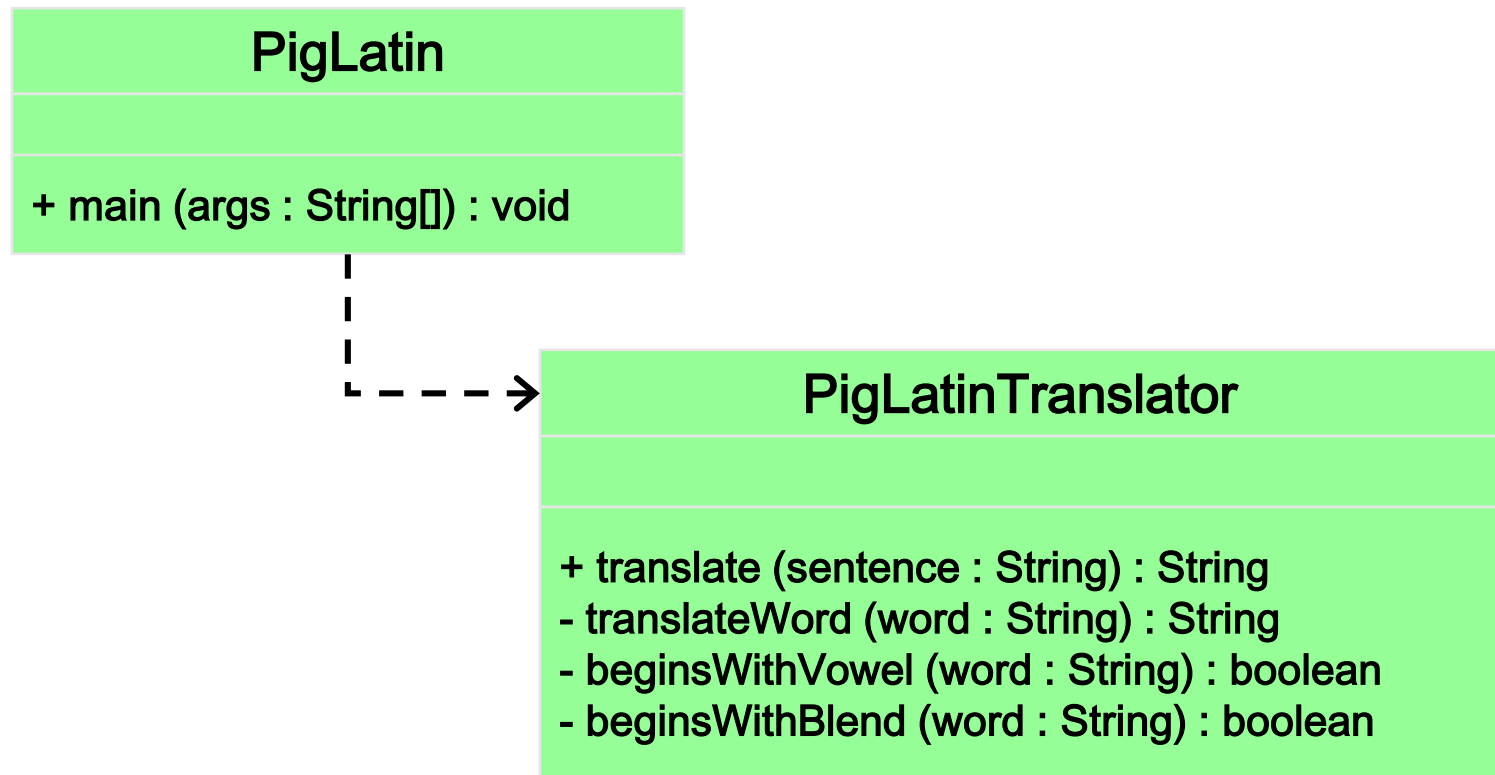
```
//-----  
//  Determines if the specified word begins with a vowel.  
//-----  
private static boolean beginsWithVowel (String word)  
{  
    String vowels = "aeiou";  
  
    char letter = word.charAt(0);  
  
    return (vowels.indexOf(letter) != -1);  
}
```

continue

continue

```
//-----  
//  Determines if the specified word begins with a particular  
//  two-character consonant blend.  
//-----  
private static boolean beginsWithBlend (String word)  
{  
    return ( word.startsWith ("bl") || word.startsWith ("sc") ||  
             word.startsWith ("br") || word.startsWith ("sh") ||  
             word.startsWith ("ch") || word.startsWith ("sk") ||  
             word.startsWith ("cl") || word.startsWith ("sl") ||  
             word.startsWith ("cr") || word.startsWith ("sn") ||  
             word.startsWith ("dr") || word.startsWith ("sm") ||  
             word.startsWith ("dw") || word.startsWith ("sp") ||  
             word.startsWith ("fl") || word.startsWith ("sq") ||  
             word.startsWith ("fr") || word.startsWith ("st") ||  
             word.startsWith ("gl") || word.startsWith ("sw") ||  
             word.startsWith ("gr") || word.startsWith ("th") ||  
             word.startsWith ("kl") || word.startsWith ("tr") ||  
             word.startsWith ("ph") || word.startsWith ("tw") ||  
             word.startsWith ("pl") || word.startsWith ("wh") ||  
             word.startsWith ("pr") || word.startsWith ("wr") );  
}
```

Class Diagram for Pig Latin



Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are passed by value
- A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header)
- When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other

Passing Objects to Methods

- What a method does with a parameter may or may not have a permanent effect (outside the method)
- Note the difference between changing the internal state of an object versus changing which object a reference points to
- See ParameterTester.java
- See ParameterModifier.java
- See Num.java

```
//*****
//  Num.java          Author: Lewis/Loftus
//
//  Represents a single integer as an object.
//*****

public class Num
{
    private int value;

    //-----
    //  Sets up the new Num object, storing an initial value.
    //-----
    public Num (int update)
    {
        value = update;
    }
}
```

continue

```
//*****
//  ParameterModifier.java          Author: Lewis/Loftus
//
//  Demonstrates the effects of changing parameter values.
//*****

public class ParameterModifier
{
    //-----
    //  Modifies the parameters, printing their values before and
    //  after making the changes.
    //-----
    public void changeValues (int f1, Num f2, Num f3)
    {
        System.out.println ("Before changing the values:");
        System.out.println ("f1\tf2\tf3");
        System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

        f1 = 999;
        f2.setValue(888);
        f3 = new Num (777);

        System.out.println ("After changing the values:");
        System.out.println ("f1\tf2\tf3");
        System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
    }
}
```

continue

```
//-----  
//  Sets the stored value to the newly specified value.  
//-----  
public void setValue (int update)  
{  
    value = update;  
}  
  
//-----  
//  Returns the stored integer value as a string.  
//-----  
public String toString ()  
{  
    return value + "  
}  
}
```

```
//*****  
//  ParameterTester.java          Author: Lewis/Loftus  
//  
//  Demonstrates the effects of passing various types of parameters.  
//*****
```

```
public class ParameterTester  
{
```

```
    //-----  
    //  Sets up three variables (one primitive and two objects) to  
    //  serve as actual parameters to the changeValues method. Prints  
    //  their values before and after calling the method.  
    //-----
```

```
    public static void main (String[] args)  
    {
```

```
        ParameterModifier modifier = new ParameterModifier();
```

```
        int a1 = 111;
```

```
        Num a2 = new Num (222);
```

```
        Num a3 = new Num (333);
```

continue

continue

```
System.out.println ("Before calling changeValues:");  
System.out.println ("a1\ta2\t a3");  
System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");  
  
modifier.changeValues (a1, a2, a3);  
  
System.out.println ("After calling changeValues:");  
System.out.println ("a1\ta2\t a3");  
System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");  
    }  
}
```

continue

System.out

System.out

System.out

modifier.c

System.out

System.out

System.out

}

}

Output

Before calling changeValues:

a1	a2	a3
111	222	333

Before changing the values:

f1	f2	f3
111	222	333

After changing the values:

f1	f2	f3
999	888	777

After calling changeValues:

a1	a2	a3
111	888	333

es:");

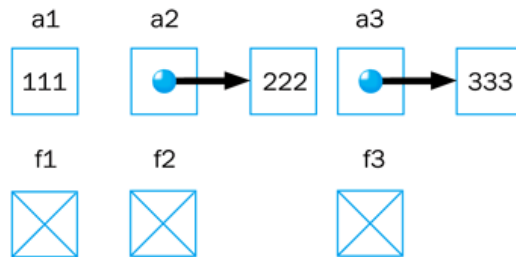
+ "\n");

s:");

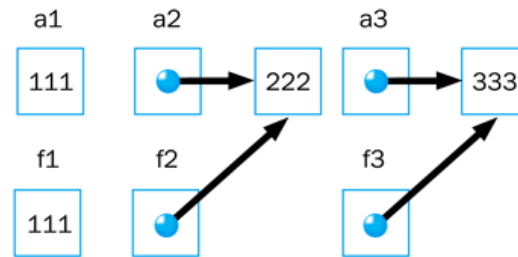
+ "\n");

STEP 1

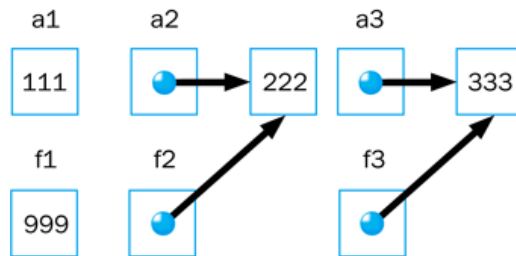
Before invoking changeValues

**STEP 2**

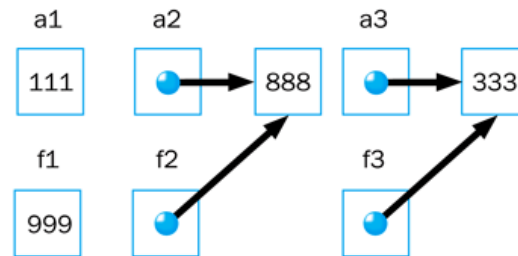
tester.changeValues (a1, a2, a3);

**STEP 3**

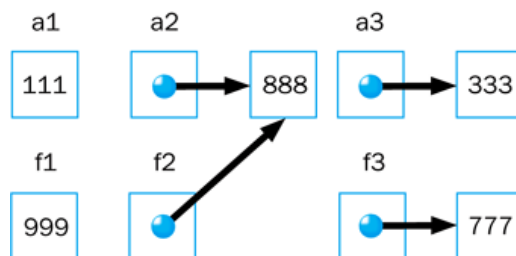
f1 = 999;

**STEP 4**

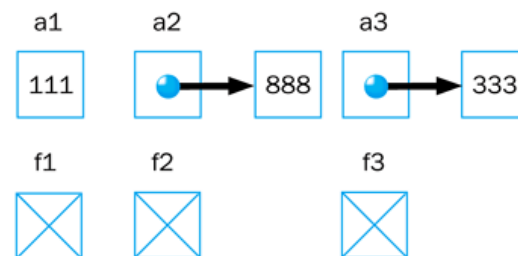
f2.setValue (888);

**STEP 5**

f3 = new Num (777);

**STEP 6**

After returning from changeValues



Method Overloading

- Let's look at one more important method design issue: method overloading
- Method overloading is the process of giving a single method name multiple definitions in a class
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The signature of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
    return x + .375;
}
```

Invocation

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

```
float tryMe(int x, float y)
{
    return x*y;
}
```



Method Overloading

- 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);

Overloaded Methods

```
static void m(int x, int y) { System.out.println("m-i-i"); }  
static void m() { System.out.println("m-noarg"); }  
static void m(double x, double y) { System.out.println("m-d-d"); }  
static void m(int x, double y) { System.out.println("m-i-d"); }  
static void m(double x, int y) { System.out.println("m-d-i"); }  
static void m(int x) { System.out.println("m-i"); }
```

- to invoke this method

```
m(1,2);
```

```
m();
```

```
m(1.1,2.2);
```

```
m(1,2.2);
```

```
m(1.1,2);
```

```
m(1);
```

Overloading Methods

- The return type of the method is not part of the signature
- That is, overloaded methods cannot differ only by their return type
- Constructors can be overloaded
- Overloaded constructors provide multiple ways to initialize a new object

Recap – Modifiers

	public	private	protected
The class itself	yes	yes	yes
Classes in the same package	yes	no	yes
Sub-classes in a different package	yes	no	yes
Non-subclasses in a different package	yes	no	no

Recap - Variables

- instance variables -- declared in the class (without using static keyword)
- class variables (static variables) - declared in the class (with using static keyword)
- local variables – declared in a method or as its formal parameters.
- An instance method of a class can refer (just using their names) to all instance variables, all static variables declared in the class, and all its local variables.
- A static method of a class cannot refer to any instance variable declared in that class. It can only refer to static variables and its local variables.

Variables (cont.)

```
class C {  
    int x;  
    static int y;  
    public void printX() { System.out.println("x: "+x); }  
    public static void printY() { System.out.println("y: "+y); }  
    public void m1(int a, int b) {  
        int c=a+b;  
        x=a; y=b;  
        printX(); printY();  
    }  
    public static m2(int a, int b) {  
        x=a;  
        y=b;  
        printX();  
        printY();  
    }  
}
```

Recall - Dot Operator (cont.)

```
class C1 {  
    public int x;  
    public static int y=5;  
    public C1() { x=1; }  
    public void setX(int val) { x=val; }  
    public static void printY() { System.out.println("y: " + y); }  
}
```

// in a method of some other class

C1 o1,o2;	o1.x = 2;
C1.y = 10;	o2.x = 3;
C1.x = 10;	o1.y = 4;
C1.printY();	o2.y = 5;
C1.setX(10);	C1.y = 6;
o1 = new C1();	o1.setX(7);
o2 = new C1();	o2.setX(8);
	o1.printY();
	o2.printY();

Testing

Testing

- Testing can mean many different things
- It certainly includes running a completed program with various inputs
- It also includes any evaluation performed by human or computer to assess quality
- Some evaluations should occur before coding even begins
- The earlier we find an problem, the easier and cheaper it is to fix

Testing

- The goal of testing is to find errors
- As we find and fix errors, we raise our confidence that a program will perform as intended
- We can never really be sure that all errors have been eliminated
- So when do we stop testing?
 - Conceptual answer: Never
 - Cynical answer: When we run out of time
 - Better answer: When we are willing to risk that an undiscovered error still exists

Reviews

- A review is a meeting in which several people examine a design document or section of code
- It is a common and effective form of human-based testing
- Presenting a design or code to others:
 - makes us think more carefully about it
 - provides an outside perspective
- Reviews are sometimes called inspections or walkthroughs

Test Cases

- A test case is a set of input and user actions, coupled with the expected results
- Often test cases are organized formally into test suites which are stored and reused as needed
- For medium and large systems, testing must be a carefully managed process
- Many organizations have a separate Quality Assurance (QA) department to lead testing efforts

Defect and Regression Testing

- Defect testing is the execution of test cases to uncover errors
- The act of fixing an error may introduce new errors
- After fixing a set of errors we should perform regression testing – running previous test suites to ensure new errors haven't been introduced
- It is not possible to create test cases for all possible input and user actions
- Therefore we should design tests to maximize their ability to find problems

Black-Box Testing

- In black-box testing, test cases are developed without considering the internal logic
- They are based on the input and expected output
- Input can be organized into equivalence categories
- Two input values in the same equivalence category would produce similar results
- Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories

White-Box Testing

- White-box testing focuses on the internal structure of the code
- The goal is to ensure that every path through the code is tested
- Paths through the code are governed by any conditional or looping statements in a program
- A good testing effort will include both black-box and white-box tests