



CS 112 – MiraCosta College

# Introduction to Computer Science II Java

## Module 8 – UML, Sorting, Patterns, Inner Classes & Interfaces

Chris Merrill

Computer Science Dept  
[cmerrill@miracosta.edu](mailto:cmerrill@miracosta.edu)

# Agenda

- Review
  - Homework, Module 7
  - Any questions on recursion?
- UML
- Sorting and Patterns
- Inner Classes and Interfaces
- Midterm next class!
- Lab 8, if time

# Binary Search - Revisited

We can use a “binary search” method to zero-in on a value of a transcendental function such as calculating the square root of a number greater than 0.

1. Set reasonable high and low (range) values
2. Is the function's new value greater or less than the midpoint?
3. Reset either high or low to the midpoint and iterate
4. Continue until we can't improve the results

# Binary Search - Revisited

Show two methods (iterative and recursive) to determine the square root of any number greater than zero using binary divide-and-conquer techniques.

*(SquareRoot)*

# Introduction to UML and Patterns

- UML and patterns are two software design tools that can be used within the context of any OOP language
- UML is a graphical language used for designing and documenting OOP software
- A pattern in programming is a kind of template or outline of a software task
  - A pattern can be realized as different code in different, but similar, applications

# UML

- Pseudocode is a way of representing a program in a linear and algebraic manner
  - It simplifies design by eliminating the details of programming language syntax
- Graphical representation systems for program design have also been used
  - *Flowcharts* and *structure diagrams* for example
- *Unified Modeling Language (UML)* is yet another graphical representation formalism
  - UML is designed to reflect and be used with the OOP philosophy

# History of UML

- As OOP has developed, different groups have developed graphical or other representations for OOP design
- In 1996, Brady Booch, Ivar Jacobson, and James Rumbaugh released an early version of UML
  - Its purpose was to produce a standardized graphical representation language for object-oriented design and documentation

# History of UML

- Since then, UML has been developed and revised in response to feedback from the OOP community
  - Today, the UML standard is maintained and certified by the Object Management Group (OMG)



# UML Class Diagrams

- Classes are central to OOP, and the *class diagram* is the easiest of the UML graphical representations to understand and use
- A class diagram is divided up into three sections
  - The top section contains the class name
  - The middle section contains the data specification for the class
  - The bottom section contains the actions or methods of the class

# UML Class Diagrams

- The data specification for each piece of data in a UML diagram consists of its name, followed by a colon, followed by its type
- Each name is preceded by a character that specifies its access type:
  - A minus sign (–) indicates private access
  - A plus sign (+) indicates public access
  - A hash tag (#) indicates protected access
  - A tilde (~) indicates “package” or default access

# UML Class Diagrams

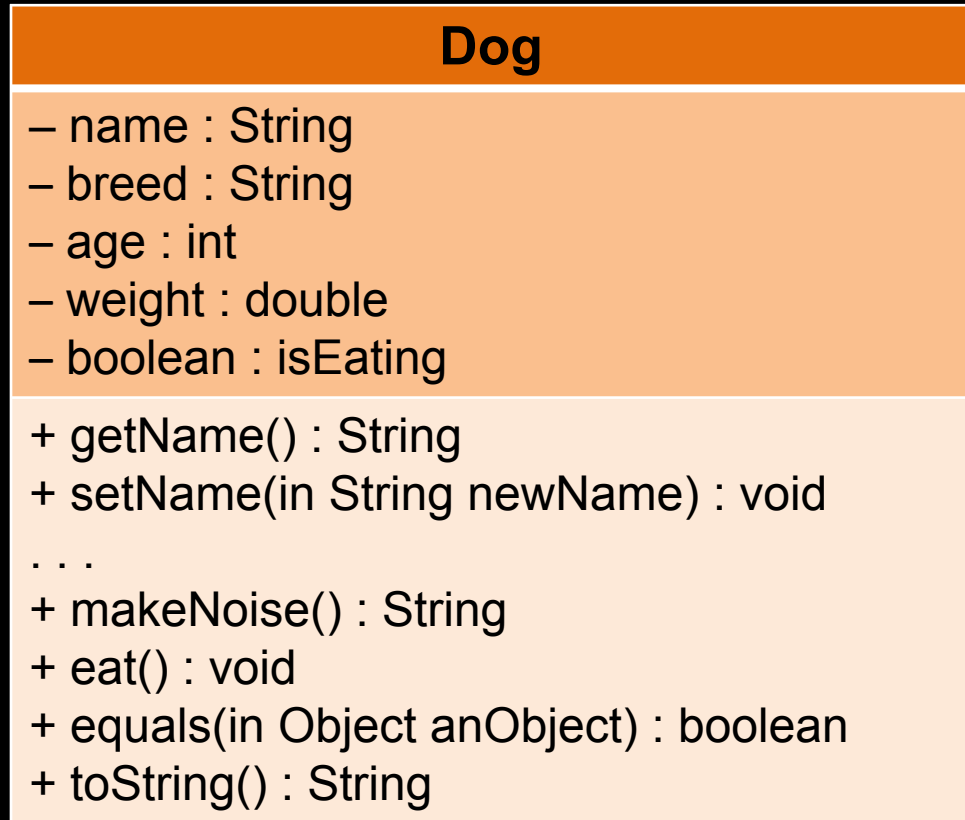
- Each method in a UML diagram is indicated by the name of the method, followed by its parenthesized parameter list, a colon, and its return type
- Access type of each method is indicated in the same way as for data
- In the textbook, parameters preceded by the word “in” followed by the type and variable name
- Another common way to document parameters is the same way that data members are documented

# UML Class Diagrams

A class diagram need not give a complete description of the class

- If a given analysis does not require that all the class members be represented, then those members are not listed in the class diagram
- Missing members are indicated with an ellipsis (three dots)

# UML Class Diagram Example



Implies that several  
other methods (i.e., →  
getters and setters)  
are available

# UML Class Diagram Example

In the prior slide:

+ setName(in String newName) : **void**

could be replaced by:

+ setName(newName : String) : **void**

and

+ equals(in Object anObject) : **boolean**

could be replaced by:

+ equals(anObject : Object) : **boolean**

# Class Interactions

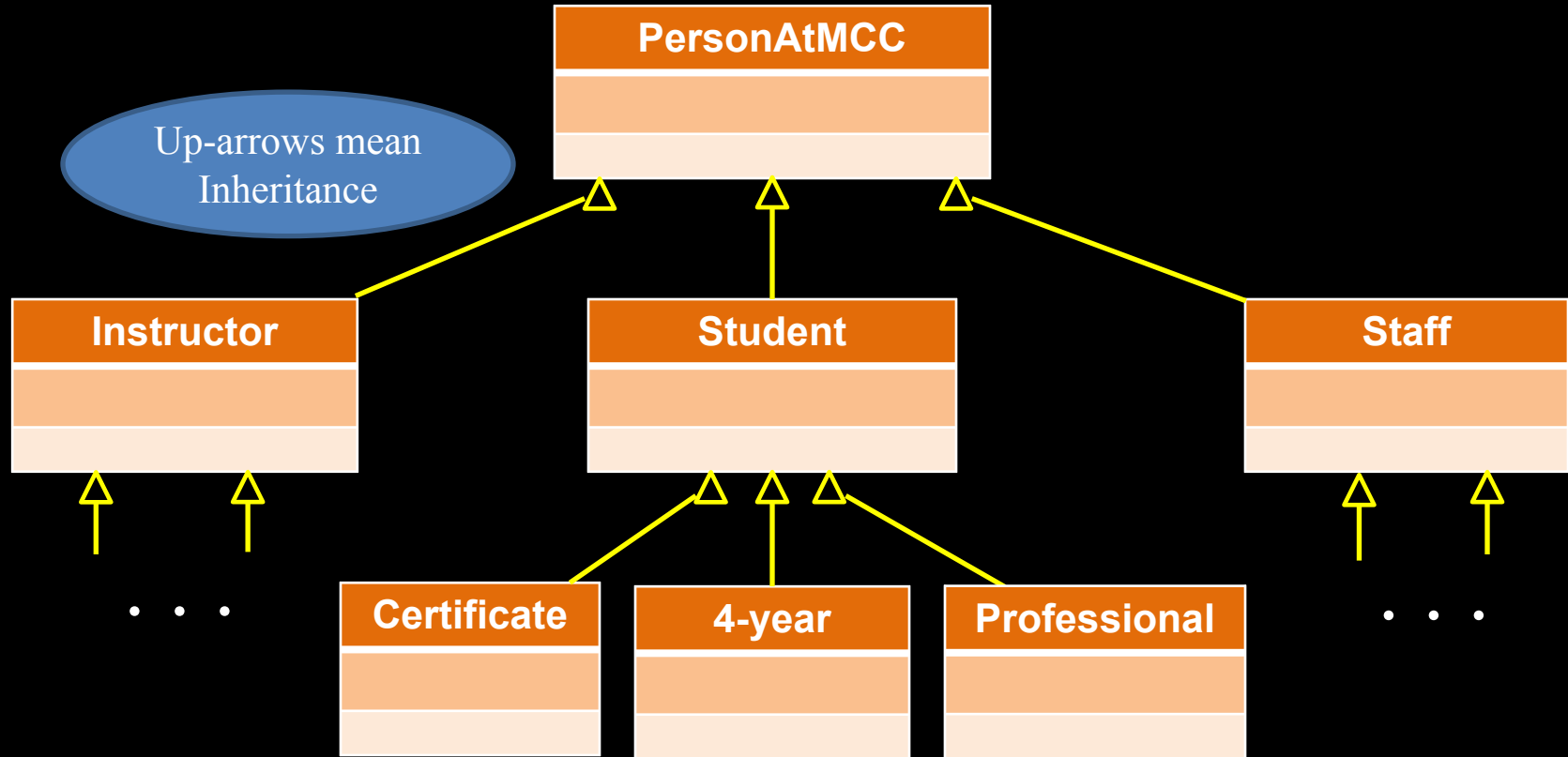
- Rather than show just the interface of a class, class diagrams are primarily designed to show the interactions among classes
- UML has various ways to indicate the information flow from one class object to another using different sorts of annotated arrows
- UML has annotations for class groupings into packages, for inheritance, and for other interactions
- In addition to these established annotations, UML is extensible

# Inheritance Diagrams

- An *inheritance diagram* shows the relationship between a base class and its derived class(es)
  - Normally, only as much of the class diagram is shown as is needed
  - Note that each derived class may serve as the base class of its derived class(es)
- Each base class is drawn above its derived class(es)
  - An upward pointing arrow is drawn between them to indicate the inheritance relationship



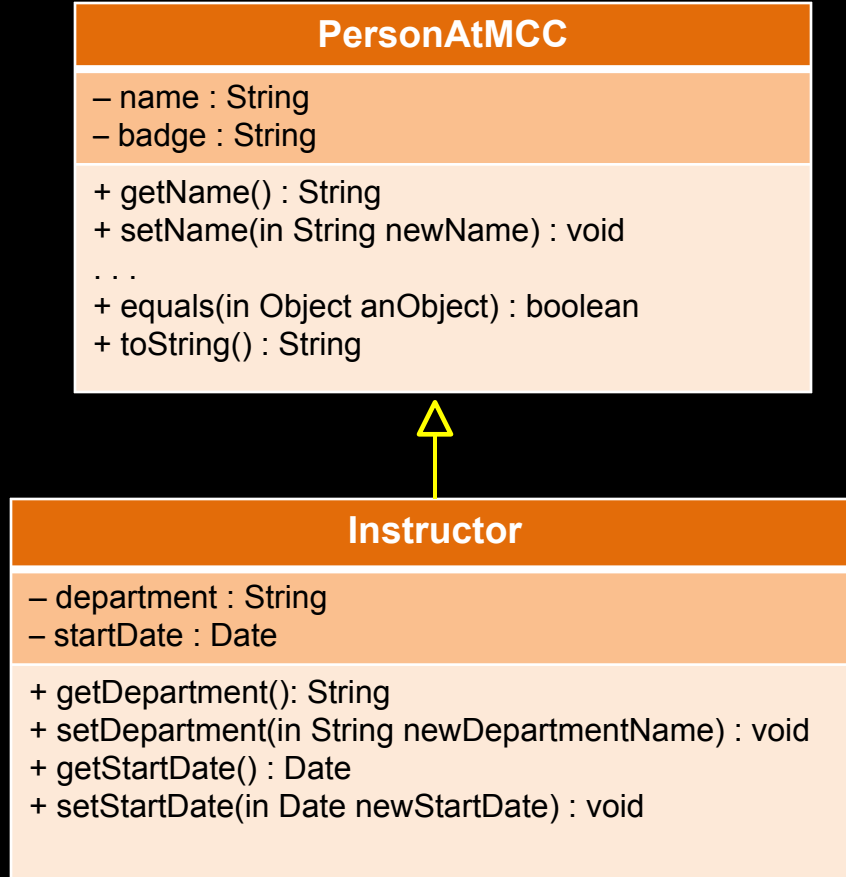
# A Class Hierarchy in UML Notation



# Inheritance Diagrams

- The arrows also help in locating method definitions
- To look for a method definition for a class:
  - Examine the class definition first
  - If the method is not found, the path of connecting arrows will show the order and direction in which to search
  - Examine the parent class indicated by the connecting arrow
  - If the method is still not found, then examine this parent's parent class indicated by the connecting arrow
  - Continue until the method is found, or until the top base class is reached

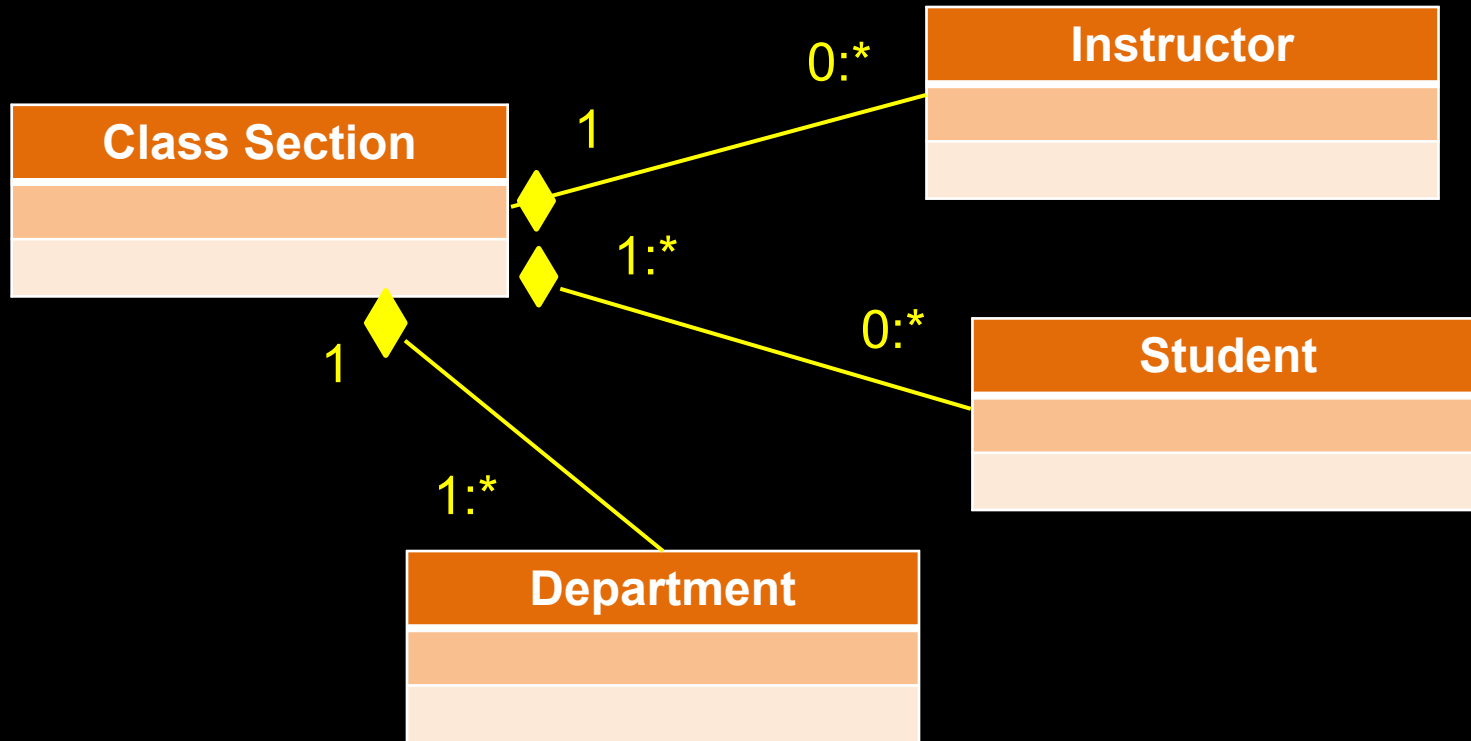
# Finding Details in a UML Class Hierarchy



Where is the  
**getStartDate**  
method located?

**setName?**

Aggregation is represented by diamond arrows



# Aggregation Models A “Has A” Relationship

- The containing class is at an end with a diamond, and the contained class the end without a diamond.
- The numbers represent the relationship between the two entities
  - A section has 1 instructor and 1 department
  - An instructor teaches 0 or more sections
  - A department offers 1 or more sections

# Patterns

*Patterns* are design outlines that apply across a variety of software applications

- To be useful, a pattern must apply across a variety of situations
- To be substantive, a pattern must make some assumptions about the domain of applications to which it applies

# Container-Iterator Pattern

*A container* is a class or other construct whose objects hold multiple pieces of data

- An array is a container
- Vectors and linked lists are containers
- A **String** value can be viewed as a container that contains the characters in the string

# Container-Iterator Pattern

- Any construct that can be used to cycle through all the items in a container is an *iterator*
  - An array index is an iterator for an array
- The *Container-Iterator* pattern describes how an iterator is used on a container



# Adaptor Pattern

- The *Adaptor* pattern transforms one class into a different class without changing the underlying class, but by merely adding a new interface
  - For example, one way to create a stack data structure is to start with an array, then add the stack interface (push, pop, empty...)

# The Model-View-Controller Pattern

The *Model-View-Controller* pattern is a way of separating the I/O task of an application from the rest of the application

- The *Model* part of the pattern performs the heart of the application
- The *View* part displays (outputs) a picture of the Model's state
- The *Controller* is the input part: It relays commands from the user to the Model

# The Model-View-Controller Pattern

- Each of the three interacting parts is normally realized as an object with responsibilities for its own tasks
- The *Model-View-Controller* pattern is an example of a divide-and-conquer strategy
  - One big task is divided into three smaller tasks with well-defined responsibilities

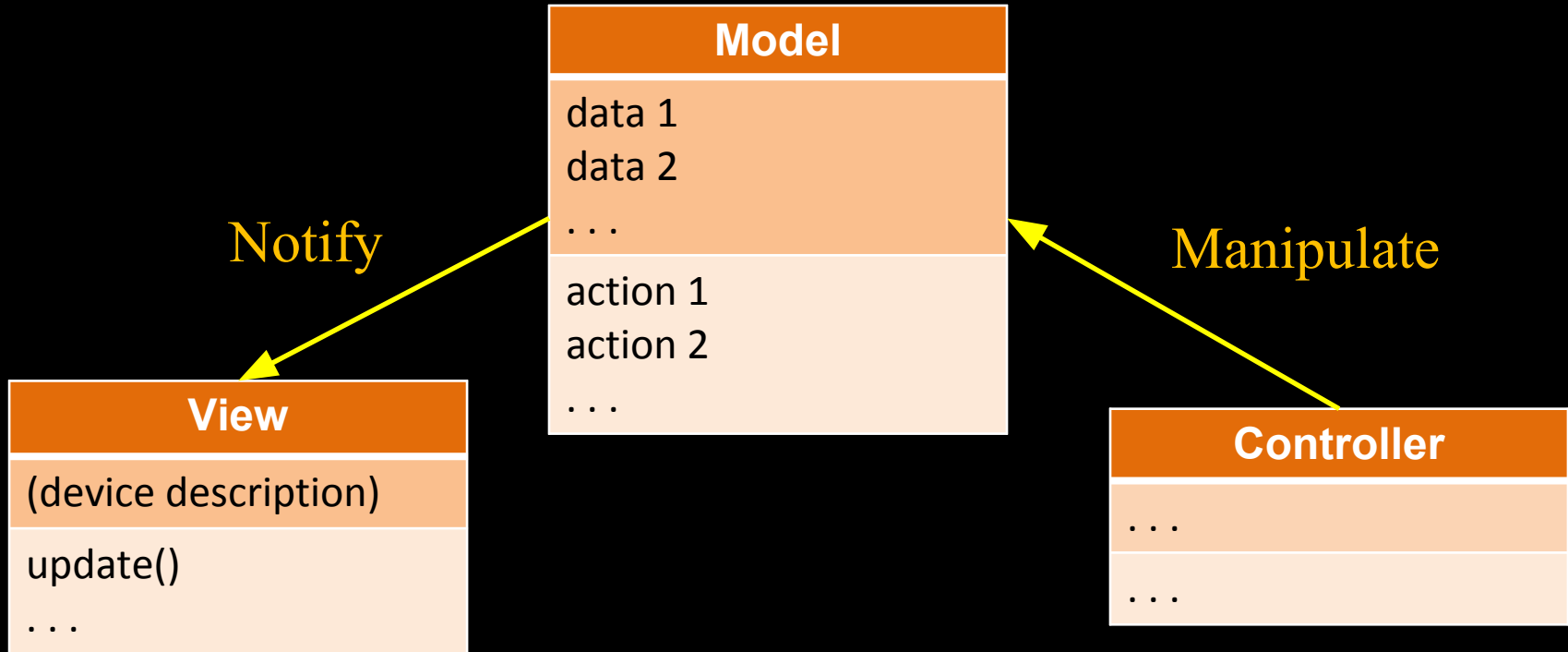
# The Model-View-Controller Pattern

- As an example, the Model might be a container class, such as an array.
- The View might display one element of the array
- The Controller would give commands to display the element at a specified index
- The Model would notify the View to display a new element whenever the array contents changed or a different index location was given

# The Model-View-Controller Pattern

- Any application can be made to fit the Model-View-Controller pattern, but it is particularly well suited to GUI (Graphical User Interface) design projects
  - The View can then be a visualization of the state of the Model

# Model-View-Controller Pattern



# A Sorting Pattern

- The most efficient sorting algorithms all seem to follow a divide-and-conquer strategy
- Given an array `a`, and using the `<` operator, these sorting algorithms:
  - Divide the list of elements to be sorted into two smaller lists (`split`)
  - Recursively sort the two smaller lists (`sort`)
  - Then recombine the two sorted lists (`join`) to obtain the final sorted list

# A Sorting Pattern

- The method `split` rearranges the elements in the interval `a[begin]` through `a[end]` and divides the rearranged interval at `splitPoint`
- The two smaller intervals are then sorted by a recursive call to the method `sort`
- After the two smaller intervals are sorted, the method `join` combines them to obtain the final sorted version of the entire larger interval



## A Sorting Pattern

- Note that the pattern does not say exactly how the methods `split` and `join` are defined
  - Different definitions of `split` will yield a different value of `splitPoint`
  - Different definitions of `join` will yield different sorting algorithms

# The Divide-and-Conquer Sorting Pattern

Replace *Type* with  
suitable type name



```
public static void sort(Type[] a, int begin, int end) {  
    if ((end - begin) >= 1) {  
        int splitPoint = split(a, begin, end) ;  
        sort(a, begin, splitPoint) ;  
        sort(a, splitPoint + 1, end) ;  
        join(a, begin, splitPoint, end) ;  
    }  
}
```

Recursion {

Changes to `split` and `join`  
result in different sorting algorithms

# Merge Sort

- The simplest realization of this sorting pattern is the *merge sort*
- The `split` method divides the array into two equal intervals without rearranging the elements
- The definition of `join` is more complicated
- Note: There is a trade-off between the complexity of the methods `split` and `join`
  - Either one can be made simpler at the expense of making the other more complicated

## Merge Sort: the `join` method

The merging starts by comparing the smallest elements in each smaller sorted interval

- The smaller of these two elements is the smallest of all the elements in either subinterval
- The method `join` makes use of a temporary array, and it is to this array that the smaller element is moved
- The process is repeated with the remaining elements in the two smaller sorted intervals to find the next smallest element, and so forth

# Merge Sort Demo

Demonstrate a Merge Sort using arrays of  
several unsorted **double** elements

*(SortDemos → MergeSort)*

# Quick Sort

In the *quick sort* realization of the sorting pattern, the definition of `split` is quite sophisticated, while `join` is utterly simple

- First, a value called the *splitting value* is chosen
  - We do this arbitrarily but other methods to select this value may be employed

## Quick Sort (continued)

- The elements in the array are then rearranged:
  - All elements less than or equal to the splitting value are placed at the front of the array
  - All elements greater than the splitting value are placed at the back of the array
  - The splitting value is placed in between the two

# Quick Sort

Note that the smaller elements are not sorted, and the larger elements are not sorted

- All the elements before the splitting value are smaller than any of the elements after the splitting value
- The smaller elements are then sorted by a recursive call, as are the larger elements
- Then these two sorted segments are combined
  - The `join` method actually does nothing



# Quick Sort Demo

Demonstrate a Quick Sort using arrays of  
several unsorted **double** elements

*(SortDemos → QuickSort)*

# Efficiency of the Sorting Pattern

Both the merge and quick sort implementations of the sorting pattern use a `split` method which divides the array into two substantial size chunks

- The merge sort `split` divides the array into two roughly equal parts, and is very efficient
- The quick sort `split` may or may not divide the array into two roughly equal parts
  - When it does not, its worst-case running time is not as fast as that of merge sort

# Efficiency vs. Simplicity of a Sorting Pattern

The selection sort algorithm (from CS 111, Chapter 6) divides the array into two pieces: one with a single element, and one with the rest of the array interval

- Because of this uneven division, the selection sort algorithm has a poor running time
- However, it's easy to understand and program

# Selection Sort Using Sort Pattern

This demo program is a modified version of  
the Selection Sort programs from Chapter 6  
updated to use the Sort Pattern

*(SelectionSort)*

# Restrictions on the Sorting Pattern

Like all patterns, the sorting pattern has some restrictions

- It applies only to types where the `<` operator is defined
- It applies only to sorting in increasing order

The pattern can be made more general by:

- Replacing the `<` operator with a **boolean**-valued method called **compare**
- Writing a **compare** method to take two arguments of the base type of the array, and return **true** or **false** based on the comparison criteria

# Pragmatics and Patterns

- Patterns are guides, *not* requirements, so it is not necessary to follow include or follow all the fine details
- For example, quick sort was described by following the sorting pattern exactly
  - Notice that, despite the fact that method calls incur overhead, the quick sort `join` method does nothing
  - In practice calls to `join` would be eliminated
  - Other optimizations can also be done once the general pattern of an algorithm is clear

# Interfaces

- An *interface* is something like an extreme case of an abstract class
  - However, *an interface is not a class*
  - *It is a type that can be satisfied by any class that “implements” the interface*
- The syntax for defining an interface is similar to that of defining a class, except the word **interface** is used in place of **class**

# Interfaces

- An interface specifies a set of methods that any class that implements the interface must have
  - It contains only method headings and constant definitions
  - It contains no instance variables nor any complete method definitions
  - It is often referred to as a “contract” to implement a method



# Interfaces

- An interface serves a function similar to a base class, though it is not a base class
  - Some languages allow one class to be derived from two or more different base classes
  - This *multiple inheritance* is not allowed in Java
  - Instead, Java's way of approximating multiple inheritance is through interfaces

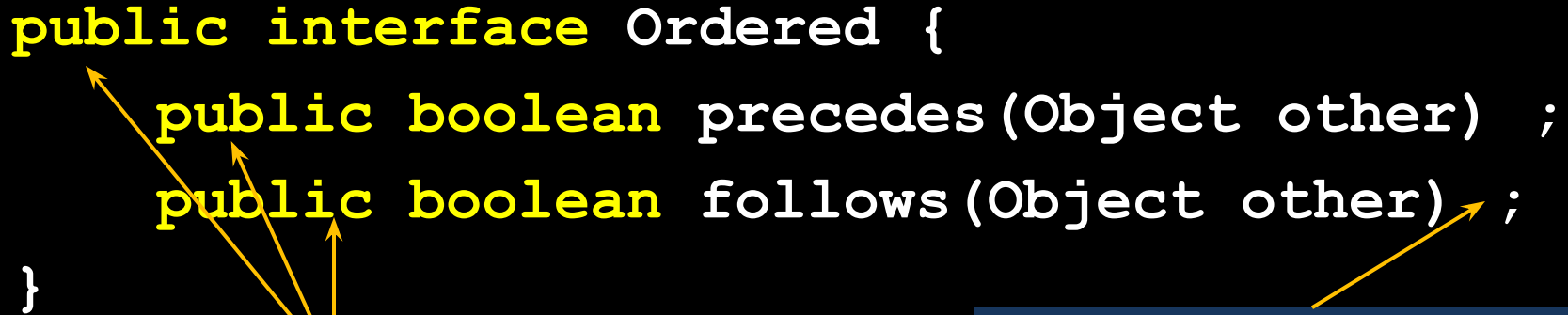
# Interfaces

- An interface and all of its method headings should be declared **public**. (They cannot be given **private**, **protected**, or “package” access)
- When a class implements an interface, it must make all the methods in the interface **public**
- Because an interface is a type, a method may be written with a parameter of an interface type, which will accept as an argument any class that implements the interface

# The Ordered Interface

(from the Textbook)

```
public interface Ordered {  
    public boolean precedes(Object other) ;  
    public boolean follows(Object other) ;  
}
```



Always **public**,  
even if omitted

Semicolon at the end of a  
method heading indicates an  
**abstract** method  
definition

# Interface Summary


- An interface provides complete abstraction as none of its methods can have body. On the other hand, an abstract class provides partial abstraction as it can have both abstract and concrete methods both.
- The `implements` keyword is used by classes to implement an interface.
- An interface cannot be instantiated in Java using the `new` operator.
- Variables can be declared of a type of interface

## Interface Summary (continued)

- Classes implementing any interface must implement all the methods in the interface, otherwise the class should be declared **abstract**.
- An interface must be declared as **public**, and its methods are **abstract** and **public** by default.
- Interface variables declared in interface are **public**, **static**, and **final** by default.
  - Interface variables must be initialized at the time of declaration, otherwise a compiler error will result

# The Salutations Interface

```
public interface Salutations {  
    String HELLO = "Hello" ,  
           GOODBYE = "Goodbye" ;  
    public void arrival() ;  
    public void departure() ;  
}
```



Interface variables are  
**public, static, and  
final** (i.e., constants)

# Interfaces

To *implement an interface*, a concrete class must do two things:

1. It must include the following phrase at the start of the interface definition

**`implements`** *Interface\_Name*

(If more than one interface is implemented, each is listed separated by commas)

2. The class must implement *all* the method headings listed in the definition(s) of the interface(s)

# Implementation of an Interface

```
public class Employee implements Salutations {  
    . . .  
    public void arrival() {  
        System.out.println(Salutations.HELLO) ;  
    }  
    public void departure() {  
        System.out.println(Salutations.GOODBYE) ;  
    }  
    . . .  
}
```



Implemented like a class  
variable

The diagram consists of two yellow arrows originating from a dark blue rectangular box at the bottom. One arrow points to the `implements` keyword in the class declaration, and the other points to the `Salutations` interface name. This visualizes the concept that the interface is implemented as a class variable.



# Derived Interfaces

- Like classes, an interface may be derived from a base interface
  - This is called *extending* the interface
  - The derived interface must include the phrase  
**`extends BaseInterfaceName`**
- A concrete class that implements a derived interface must have definitions for all methods in both the derived and base interfaces.

## Extending an Interface

```
public interface Gestures
    extends Salutations {
    public void handShake() ;
    public void wave() ;
}
```

# Abstract Classes Implementing Interfaces

- Abstract classes may implement one or more interfaces
  - Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class *and the interface*

# An Abstract Class Implementing an Interface

```
public abstract class Greeter implements Gestures {  
    . . .  
    public void arrival() {  
        System.out.println(Salutations.HELLO) ;  
    }  
    public abstract void departure() ;  
    public abstract void handshake() ;  
    public abstract void wave() ;  
}
```

concrete

abstract

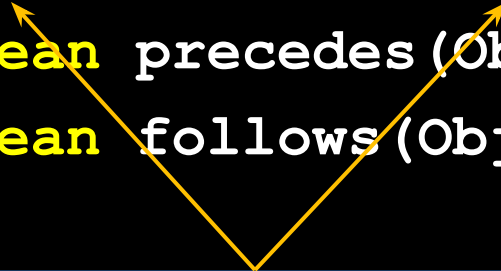
# Interface Semantics Are Not Enforced

When a class implements an interface, the compiler and run-time system check the syntax of the interface

- *Neither* checks that the body of an interface is consistent with its intended meaning
- Required semantics for an interface are normally added to the documentation for an interface
- Like any other class or method, it is the responsibility of each programmer implementing the interface to follow the semantics

# Enforcing Semantics

```
public interface Ordered {  
    // For objects of the class o1 and o2  
    // o1.follows(o2) == o2.precedes(o1)  
    public boolean precedes(Object other) ;  
    public boolean follows(Object other) ;  
}
```



Java does not enforce this restriction --  
that's up to the programmer.  
Java worries only about syntax.

# An Interface Example

A simple interface using fruits, some of  
which are “peelable”

*(FruitDemo)*

# The **Comparable** Interface

- Chapter 6 of the textbook discussed the Selection Sort algorithm, and examined a method for sorting a partially filled array of type **double** into increasing order
- This code could be modified to sort into decreasing order, or to sort integers, characters, or strings instead
  - Each of these methods would be essentially the same, but making each modification would be a nuisance – and a poor use of our time.



# The **Comparable** Interface

- If we wrote a generalized version, then the only differences would be
  - the types of values being sorted
  - the definition of the ordering
- Let's explore the use of the **Comparable** interface to provide a single sorting method that covers all these cases, and more...

# The Comparable Interface

- The Comparable interface is in java.lang, and so is automatically available to any program
- It has only one method heading which must be implemented:

```
public int compareTo(Object other) ;
```

- It is the programmer's responsibility to follow the semantics of the Comparable interface when implementing it

# The **Comparable** Interface Semantics

The method **compareTo** must return

- A negative number if the calling object "comes before" the parameter other
- Zero if the calling object "equals" the parameter
- A positive number if the calling object "comes after" the parameter

If the parameter is not of the same type as the class being defined, then a **ClassCastException** should be thrown

# The **Comparable** Interface Semantics

Any reasonable notion of “comes before” is acceptable

- In particular, all of the standard less-than relations on numbers and lexicographic ordering on **String** objects are suitable

The relationship “comes after” is just the opposite of “comes before”

- If A “comes before” B, then B “comes after” A

## The **Comparable** Interface Semantics

The "equals" of the **compareTo** method's semantics usually coincides with the **equals** method when possible, but this is not required:

## Redefining equals using compareTo

For the XYZ class which implements Comparable:

```
public boolean equals(Object other) {  
    if ((other == null) ||  
        getClass() != other.getClass()) {  
        return false ;  
    }  
    return compareTo((XYZ) other) == 0) ;  
}
```

# The **Comparable** Interface Semantics

Other ordering may be considered, as long as it is a *total ordering*, meaning that it satisfies the following:

- (*Reflexive*) any object **a** always equals itself
- (*Transitive*) If **a** comes before **b**, and **b** comes before **c**, then **a** comes before **c**
- (*Trichotomy*) For any two object **a** and **b**, one and only one of the following holds true: **a** comes before **b**, **a** comes after **b**, or **a** equals **b**

## Using the **Comparable** Interface

- The following demo reworks the **SelectionSort** class from Chapter 6
- The new **GeneralizedSelectionSort** includes a method that can sort any partially filled array whose base type implements the **Comparable** interface
  - It contains appropriate **indexOfSmallest** and **interchange** methods as well



## Using the **Comparable** Interface

- Both the **Double** and **String** classes implement the **Comparable** interface
  - In fact, all wrapper classes implement this interface
  - Interfaces apply to classes or other interfaces only
    - A primitive type (e.g., **double**) cannot implement an interface or extend a class

# Comparable Interface Demo

A slightly-modified version of the  
Comparable demo from the Textbook  
*(ComparableDemo)*

# Defined Constants in Interfaces

- An interface can contain defined constants in addition to or instead of method headings
  - Any variables defined in an interface must be **public**, **static**, and **final**
  - Because this is understood, Java allows these modifiers to be omitted
- Any class that implements the interface has access to these defined constants

# Inconsistent Interfaces

- In Java, a class can have only *one* base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- But a class can implement *any number* of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

# Inconsistent Interfaces

When a class implements two interfaces:

- One inconsistency occurs if the interfaces have constants with the same name, but different values
- Another inconsistency occurs if the interfaces contain methods with the same name but different return types
- If a class definition implements two inconsistent interfaces, then it's a compiler error – the class definition is illegal

# The **Serializable** Interface

An extreme but commonly used example of an interface is the **Serializable** interface

- It has no method headings and no defined constants: It is completely empty
- It is used merely as a type tag that indicates to the system that it may implement file I/O in a particular way

# The `Cloneable` Interface

The `Cloneable` interface is another unusual example of a Java interface

- It does not contain method headings or defined constants
- It is used to indicate how the method `clone` (inherited from the `Object` class) should be used and redefined

# The `Cloneable` Interface

- The method `Object.clone()` does a bit-by-bit copy of the object's data in storage
- If the data is all primitive type data or data of immutable class types (such as `String`), then this is adequate
  - This is the simple case
- The following is an example of a simple class that has no instance variables of a mutable class type, and no specified base class
  - So the base class is `Object`



## “Simple” case `clone` method

```
public class MyClass implements Cloneable {  
    . . .  
    public Object clone() {  
        try {  
            return super.clone() ;  
        }  
        catch (CloneNotSupportedException e)  
            // shouldn't happen  
        {  
            return null ;  
        }  
    }  
}
```

Invoke `Object`'s  
`clone` method



To keep Java's  
compiler happy



# The `Cloneable` Interface

- If the data in the object to be cloned includes instance variables whose type is a mutable class, then the simple implementation of `clone` would cause a *privacy leak*
- When implementing the `Cloneable` interface for a class like this:
  1. First invoke the `clone` method of the base class `Object` (or whatever the base class is)

# The **Cloneable** Interface

2. Then reset the values of any new instance variables whose types are mutable class types. This is done by making copies of the instance variables by invoking *their* clone methods
- This will work properly only if the **Cloneable** interface is implemented for the classes to which the instance variables belong, and for the classes to which any of the instance variables of the above classes belong, and so on and so forth (a *deep* copy).

Demo of “Harder” Case of **clone** Method

Using **Circle** objects with **Point** (object)  
centers, showing simple (shallow) and harder  
(deep) cloning methods

*(HarderCloneDemo)*

Diamond-pointed  
arrows represent  
aggregation

1.1

## Circle

center : Point  
radius : double

Circle(in Point center, in double radius )  
Circle(in Circle aCircle)  
clone() : Object  
... (getters and setters) ...  
equals(in Object anObject) : boolean  
toString() : String

## DeepCircle

DeepCircle(in Point center, in radius  
double)  
clone() : Object

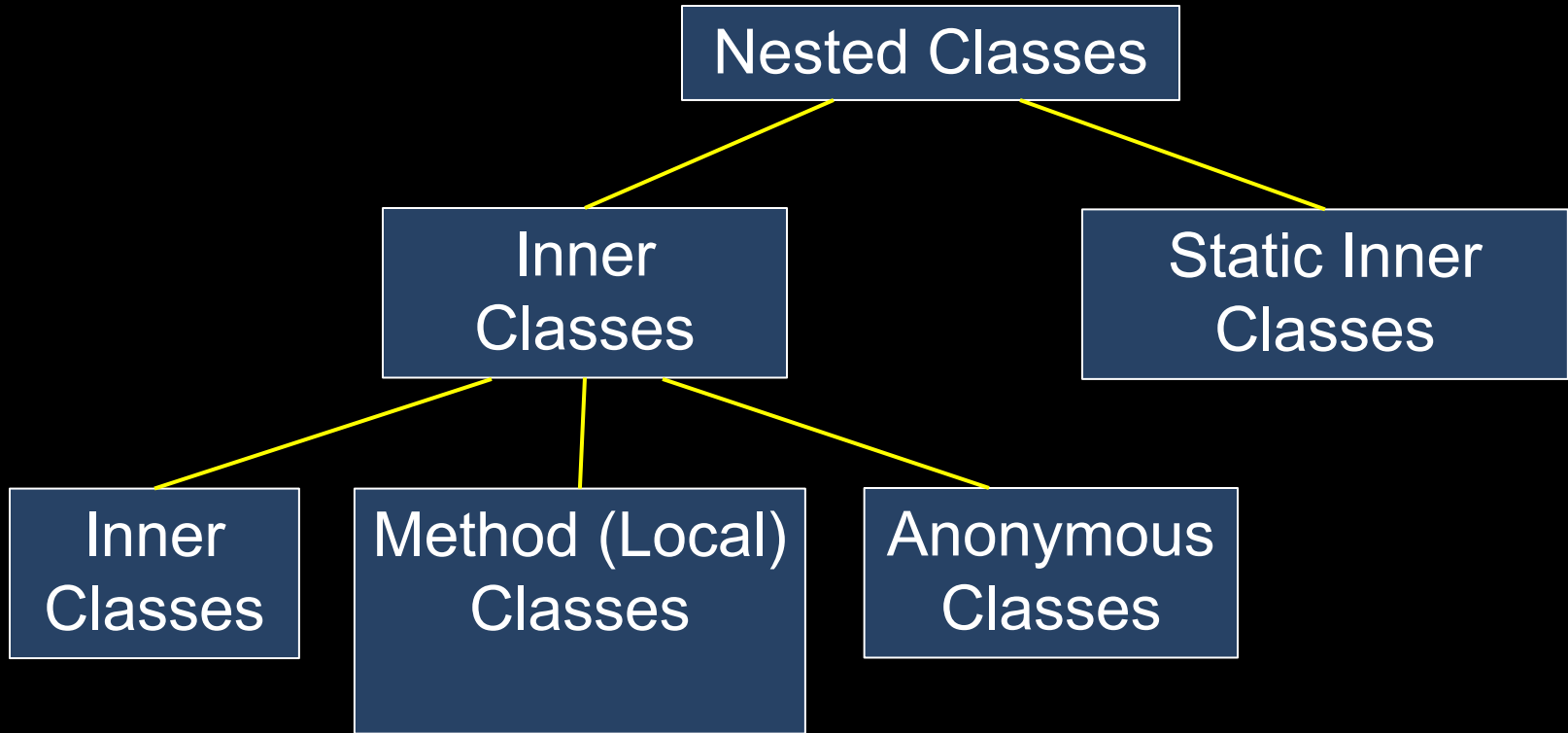
## Point

x : double  
y : double

Point(in double x, in double y)  
Point(in Point aPoint)  
clone () : Object  
...  
equals(in Object object) : boolean  
toString() : String

# UML Hierarchy and Class Diagrams

# Types of Nested Classes



# Simple Uses of Nested Classes

Nested classes are classes defined within other classes

- The class that includes the inner class is called the outer class
- There is no particular location where the definition of the nested class (or classes) must be placed within the outer class
- Placing it first or last, however, will guarantee that it is easy to find

# Simple Uses of Nested Classes

An nested class definition is a member of the outer class in the same way that the instance variables and methods of the outer class are members

- An nested class is local to the outer class definition
- The name of a nested class may be reused for something else outside the outer class definition
- If the nested class is private, then the nested class cannot be accessed by name outside of the outer class



# Simple Uses of Nested Classes

There are two main advantages to nested classes

1. They can make the outer class more self-contained since they are defined inside a class
  2. Both of their methods have access to each other's **private** methods and instance variables
- Using a nested class as a helping class is one of the most useful applications of nested classes
    - If used as a helping class, a nested class should be marked **private**

## Inner and Outer Classes Have Access to Each Other's **private** Members

- Within the definition of the inner or outer classes, the modifiers **public** and **private** are equivalent
- Within the definition of a method of an inner class:
  - It is legal to reference a private instance variable of the outer class
  - It is legal to invoke a private method of the outer class

# Inner and Outer Classes Have Access to Each Other's **private** Members

- Within the definition of a method of the outer class
  - It is legal to reference a private instance variable of the inner class on an object of the inner class
  - It is legal to invoke a (nonstatic) method of the inner class as long as an object of the inner class is used as a calling object

# An Inner Class Example

Demonstration of **BankAccount** example  
from textbook, using an inner class named  
**Money** as a helper class

*(BankAccountDemo)*

## The `.class` File for an Inner Class

- Compiling any class in Java produces a `.class` file named `ClassName.class`
- Compiling a class with one (or more) inner classes causes both (or more) classes to be compiled, and produces two (or more) `.class` files
  - Such as `ClassName.class` and `ClassName$InnerClassName.class`

# Static Inner Classes

- A normal inner class has a connection between its objects and the outer class object that created the inner class object
  - This allows an inner class definition to reference an instance variable, or invoke a method of the outer class
- There are certain situations, however, when a nested class must be static
  - If an object of the nested class is created within a static method of the outer class
  - If the nested class must have static members

# Static Inner Classes

- Since a static inner class has no connection to an object of the outer class, within an inner class method
  - instance variables of the outer class cannot be referenced
  - non-static methods of the outer class cannot be invoked
- To invoke a static method or to name a static variable of a static inner class within the outer class, preface each with the name of the inner class and a dot

# The Purpose of Static Inner Classes

Static classes are useful where the rules of scope are relaxed (not requiring a reference to an enclosing instance)

- For example, “comparators” are used to compare two instances of the same class, often for the purpose of ordering or sorting objects of the outer class

Overall, since static classes do not have access to instance variables or non-static methods of the outer class, many programmers make then inner classes simply for “packaging” purposes such as Listeners.



# Public Inner Classes

- If an inner class is marked **public**, then it can be used outside of the outer class
- In the case of a non-static inner class, it must be created using an object of the outer class

```
BankAccount account = new BankAccount();  
BankAccount.Money amount =  
    account.new Money("41.99");
```

- Note that the prefix **account.** comes before **new**
- The new object **amount** can now invoke methods from the inner class, but only from the inner class

## Public Inner Classes

In the case of a static inner class, the procedure is similar to, but simpler than, that for non-static inner classes

```
OuterClass.InnerClass innerObject =  
    new OuterClass.InnerClass();
```

All of the following are acceptable:

```
innerObject.nonstaticMethod();
```

```
innerObject.staticMethod();
```

```
OuterClass.InnerClass.staticMethod();
```

# Referring to a Method of the Outer Class

If a method is invoked in an inner class

- If the inner class has no such method, then it is assumed to be an invocation name in the outer class
- If both inner and outer classes have a method with the same name, then it is an invocation in the inner class
- If both the inner and outer class have a method with the same name, and the intent is to invoke the method in the outer class, then use the following

*OuterClassName.this.methodName()*

# Nesting Inner Classes

It is legal to nest inner classes within other inner classes

- The rules are the same as before, but the names get longer
- Given class **A**, which has public inner class **B**, which has public inner class **C**, then the following is valid:

```
A aObject = new A();
```

```
A.B bObject = aObject.new B();
```

```
A.B.C cObject = bObject.new C();
```

# Inner Classes and Inheritance

- Given an *OuterClass* that has an *InnerClass*
  - Any *DerivedClass* of *OuterClass* will automatically have *InnerClass* as an inner class
  - In this case, the *DerivedClass* cannot override the *InnerClass*
- An outer class can be a derived class
- An inner class can be a derived class also

# Access Demo

A demonstration showing:

1. Access to a static inner class
2. Access to a non-static inner class
3. Access to a method-local inner class

*(InnerAndOuterAccess)*

# Anonymous Classes

If an object is to be created, but there is no need to name the object's class, then an *anonymous class* definition can be used

- The class definition is embedded inside the expression with the **new** operator
- That means that anonymous classes are declared and instantiated at the same time!

# Anonymous Classes

## Notes:

- Anonymous classes are good examples of polymorphism, as the variable created is of the parent type, but references an object of the anonymous (inner) class type
- The parent type is often a Java interface. In this case, the anonymous class must implement all methods in the interface.



# Anonymous Classes

- Anonymous classes are often used when the programmer wants to override an existing method of a class or interface.
- Anonymous class objects can be passed as arguments to a method just like any other object, as long as they are of the type required by the parameter list of the method

# Anonymous Class Summary

## Anonymous classes:

- are *inner* classes
- have no names
- do not have constructors (because they lack names)
- cannot be static
- can be instantiated only once
- are declared with curly braces ending with a semicolon

# Anonymous Classes

## Demo 1

An anonymous class cannot create new methods. Instead, it must override existing methods in the original class – otherwise a compiler error will result.

*(AnimalMoves)*

# Anonymous Classes

## Demo 2

Using anonymous classes and  
interfaces

*(AnonymousDogs)*

# Homework

- Complete old homework and labs.
- Complete Interface & Inner Class lab
- Homework Assignment, Module 8, projects 1 and 2
- Turn in everything at the beginning of next class after the Midterm
- Prepare for Midterm

## Lab 8 – UML, Inner Classes, Interfaces

Create UML hierarchy and class diagrams three classes (Building, House, and Office), and use the Comparable interface.