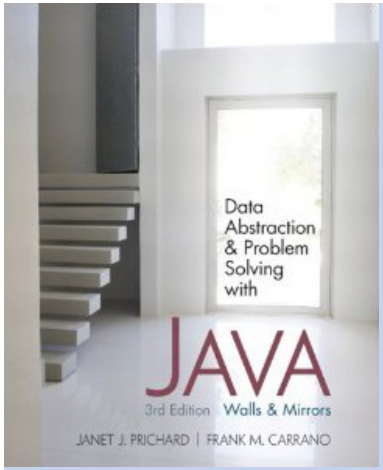

Chapter 9

Advanced Java Topics

Inheritance Revisited

- Inheritance
  - Allows a class to derive the behavior and structure of an existing class

Inheritance Revisited

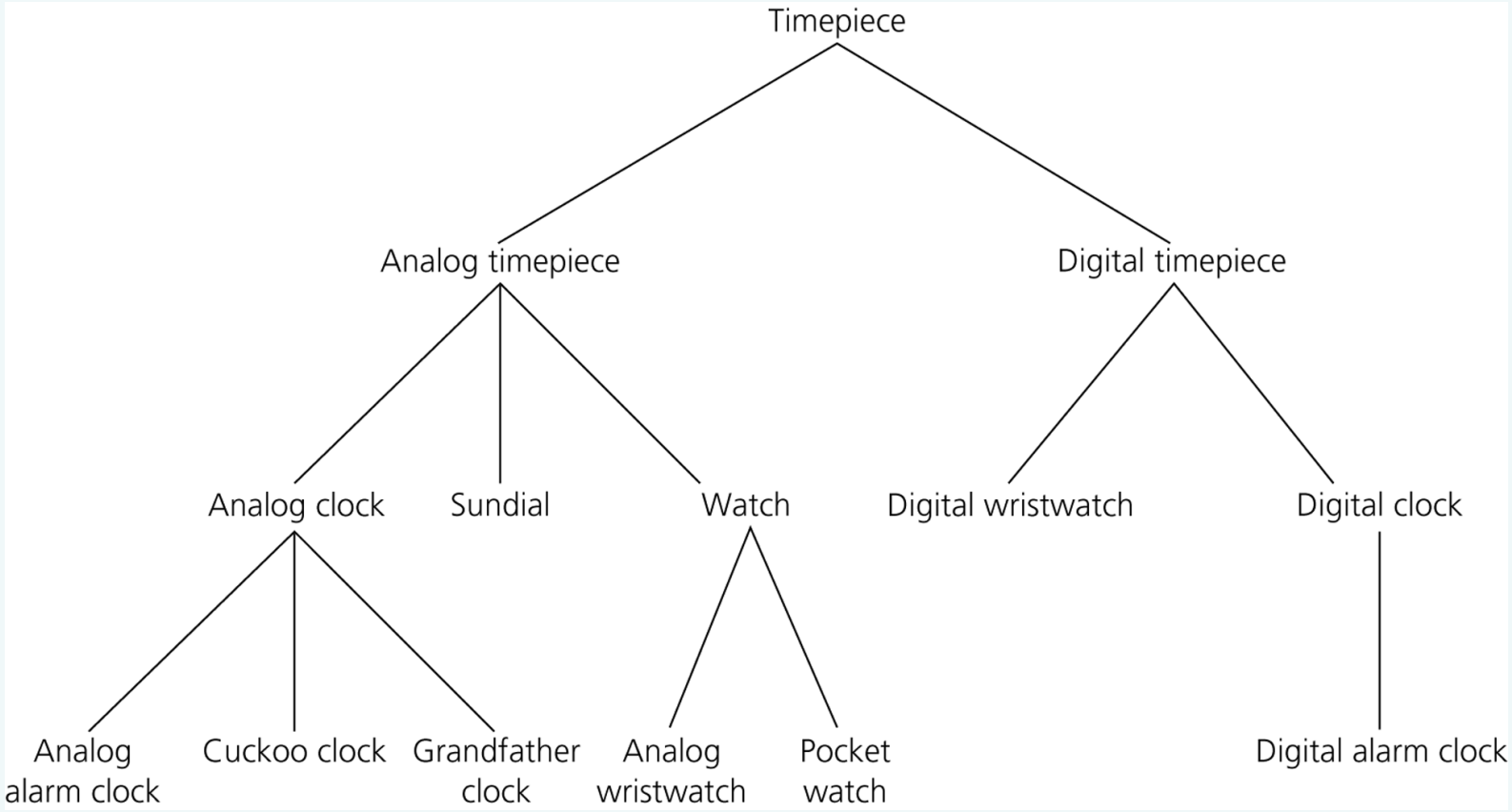

Figure 9-1

Inheritance: Relationships among timepieces

Inheritance Revisited

- Superclass or base class
  - A class from which another class is derived
- Subclass, derived class, or descendant class
  - A class that inherits the members of another class
- Benefits of inheritance
  - It enables the reuse of existing classes
  - It reduces the effort necessary to add features to an existing object

Inheritance Revisited

- A subclass
  - Can add new members to those it inherits
  - Can override an inherited method of its superclass
    - A method in a subclass overrides a method in the superclass if the two methods have the same declarations

Inheritance Revisited

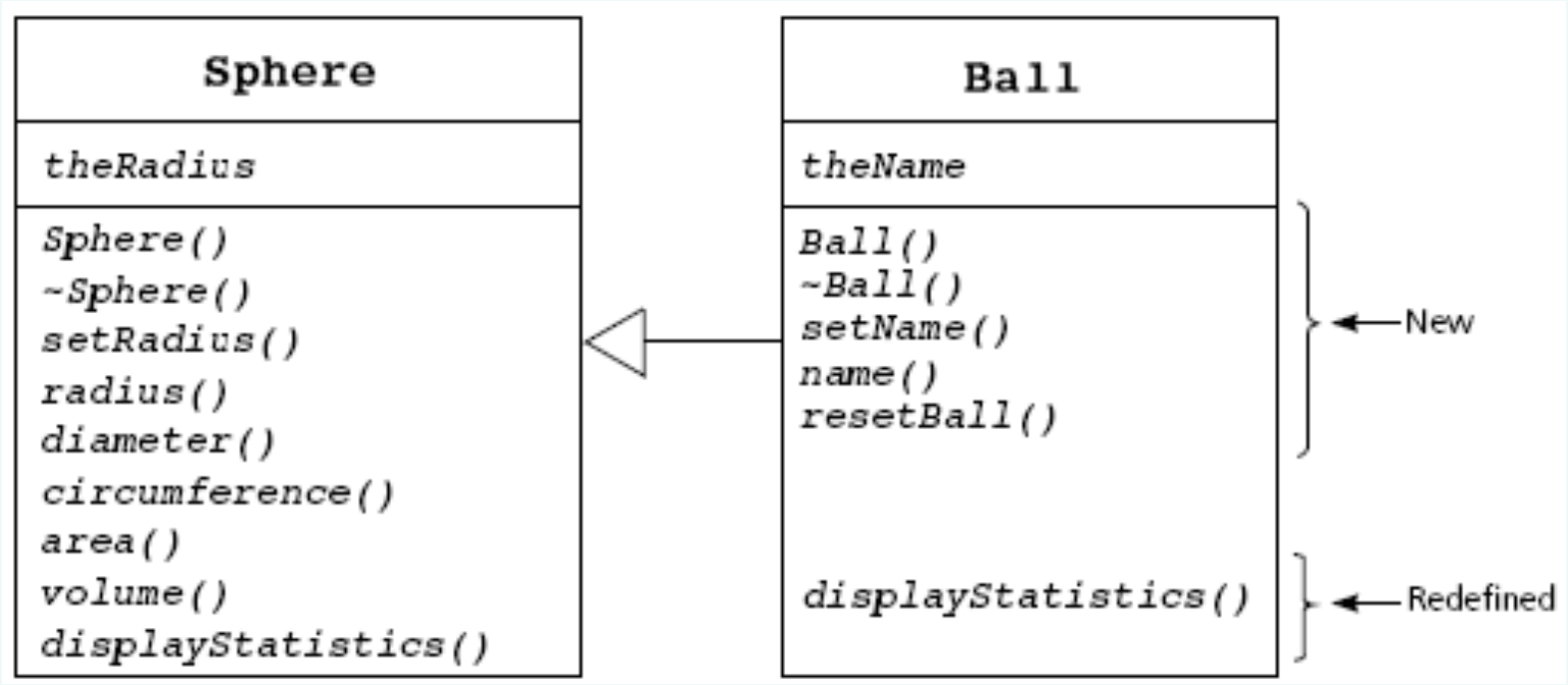

Figure 9-2

The subclass ***Ball*** inherits members of the superclass ***Sphere*** and overrides and adds methods

Inheritance Revisited

- A subclass inherits private members from the superclass, but cannot access them directly
- Methods of a subclass can call the superclass's public methods
- Clients of a subclass can invoke the superclass's public methods
- An overridden method
  - Instances of the subclass will use the new method
  - Instances of the superclass will use the original method

Inheritance Revisited

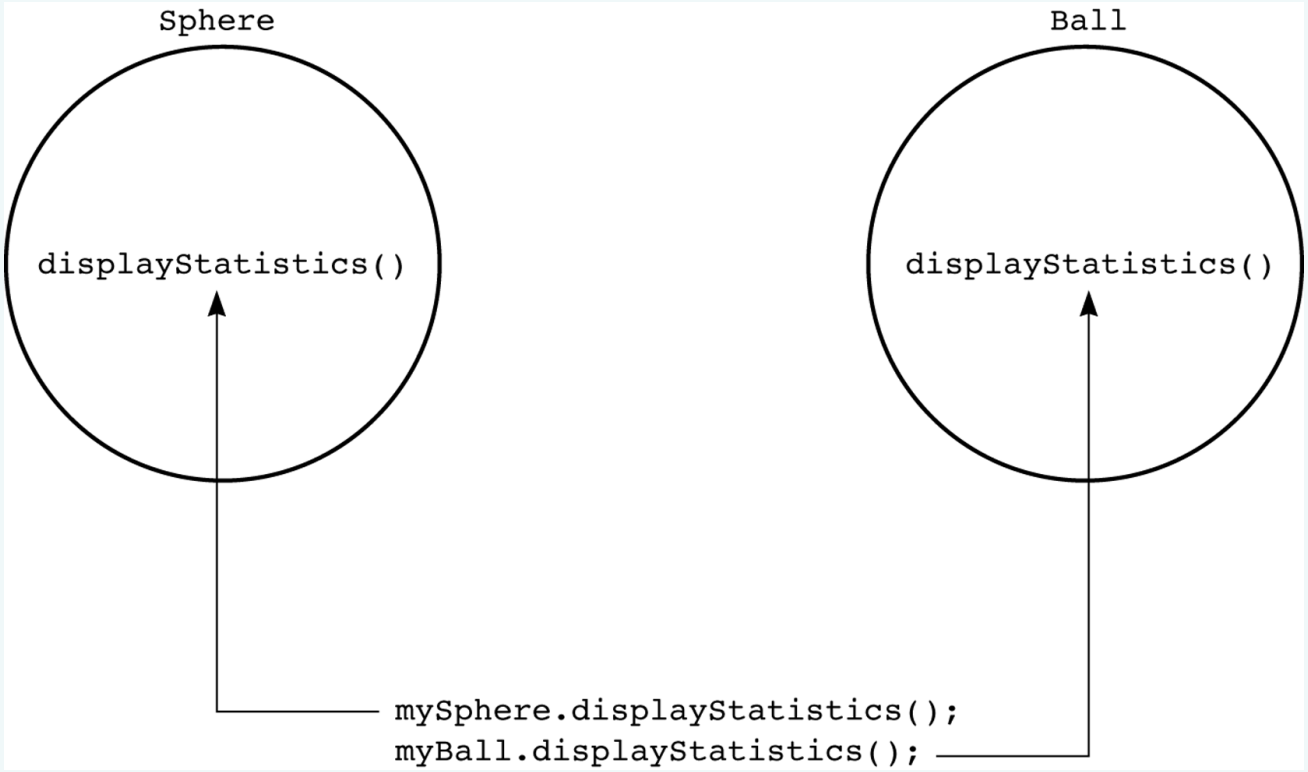

Figure 9-3

An object invokes the correct version of a method

Java Access Modifiers

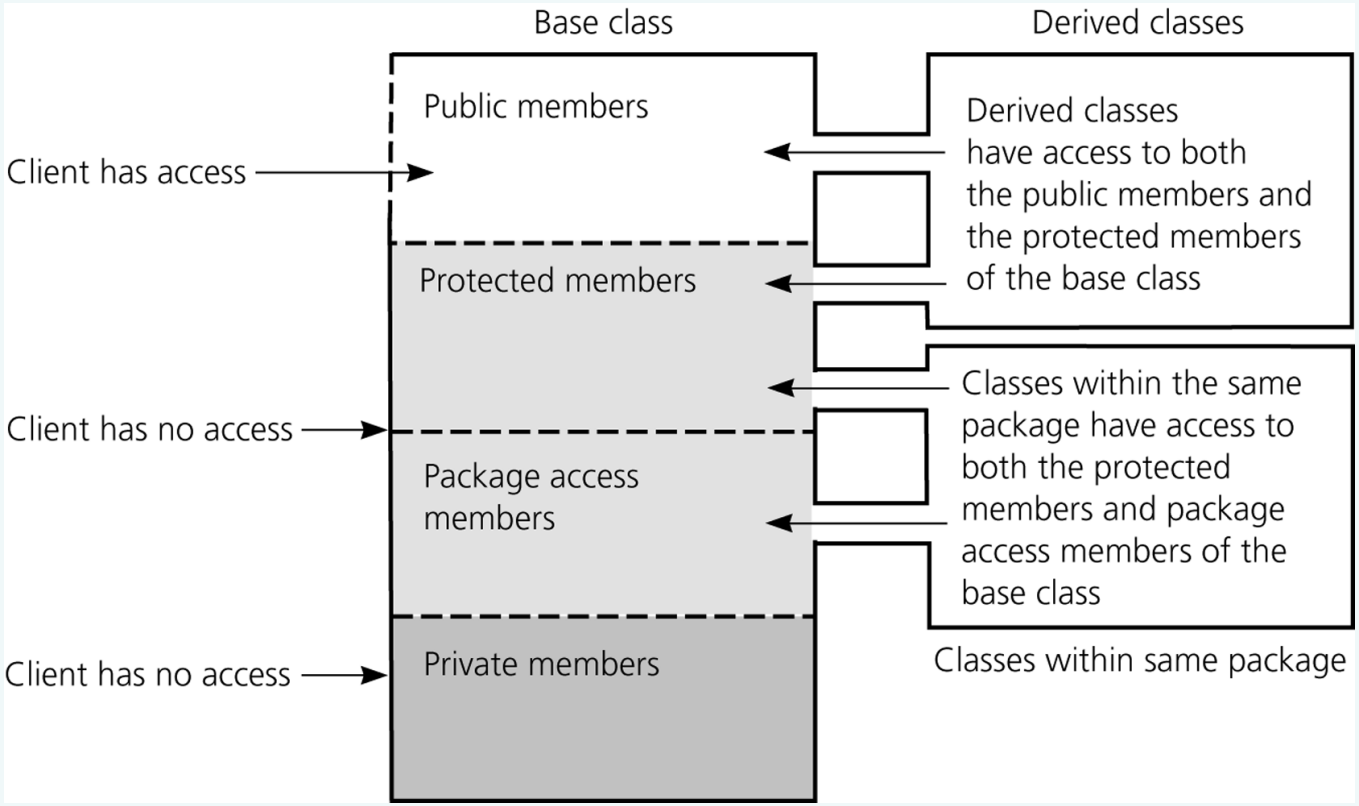

Figure 9-4

Access to public, protected, package access, and private members of a class by a client and a subclass

Java Access Modifiers

- Membership categories of a class
  - Public members can be used by anyone
  - Members declared without an access modifier (the default) are available to
    - Methods of the class
    - Methods of other classes in the same package
  - Private members can be used only by methods of the class
  - Protected members can be used only by
    - Methods of the class
    - Methods of other classes in the same package
    - Methods of the subclass

Is-a and Has-a Relationships

- Two basic kinds of relationships
  - Is-a relationship
  - Has-a relationship

Is-a Relationship

- Inheritance should imply an is-a relationship between the superclass and the subclass
- Example:
  - If the class `Ball` is derived from the class `Sphere`
    - A ball is a sphere

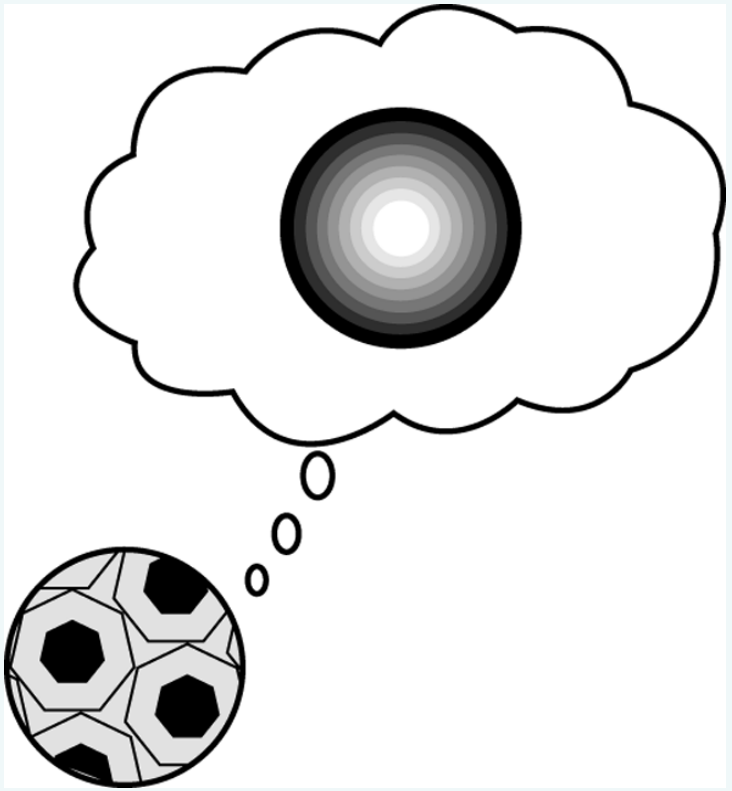

Figure 9-5

A ball “is a” sphere

Is-a Relationship

- Object type compatibility
  - An instance of a subclass can be used instead of an instance of the superclass, but not the other way around

Has-a Relationships

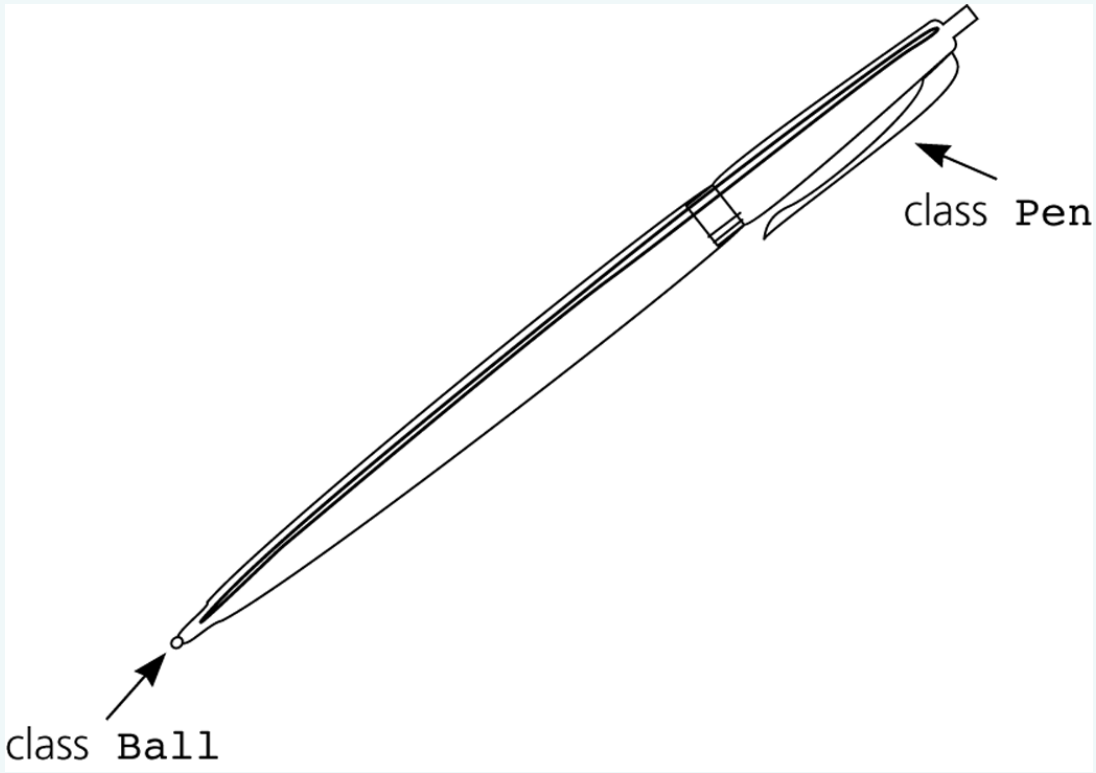

Figure 9-6

A pen “has a” or “contains a” ball

Has-a Relationships

- Has-a relationship
  - Also called containment
  - Cannot be implemented using inheritance
    - Example: To implement the has-a relationship between a pen and a ball
      - Define a data field `point` – whose type is `Ball` – within the class `Pen`

Dynamic Binding and Abstract Classes

- A polymorphic method
  - A method that has multiple meanings
  - Created when a subclass overrides a method of the superclass
- Late binding or dynamic binding
  - The appropriate version of a polymorphic method is decided at execution time

Dynamic Binding and Abstract Classes

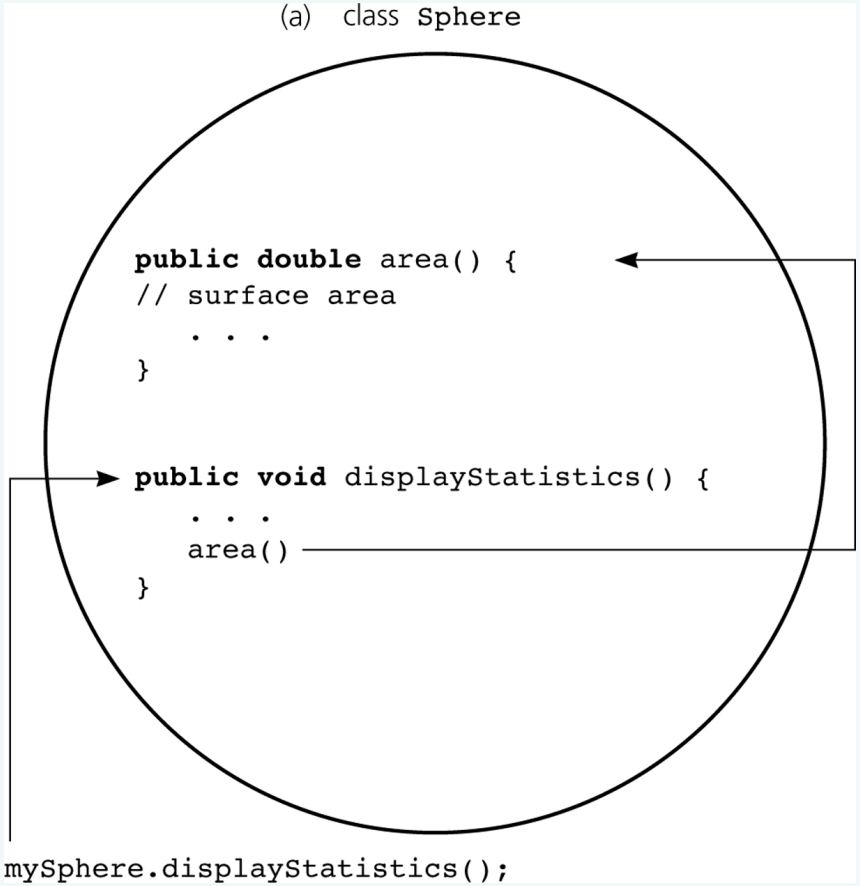

Figure 9-7a

area is overridden: a) ***mySphere.DisplayStatistics( )*** calls ***area*** in ***Sphere***

Dynamic Binding and Abstract Classes

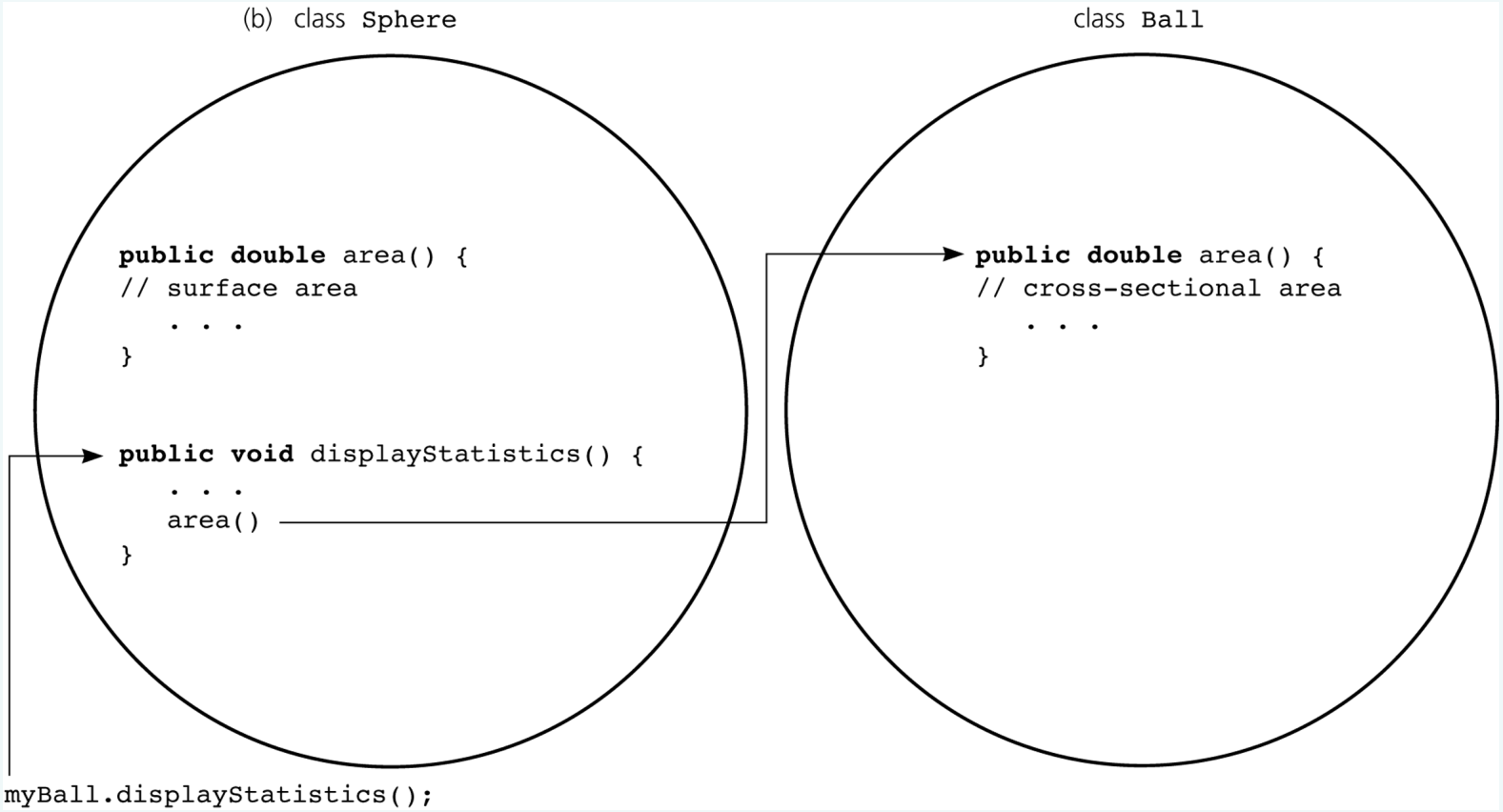

Figure 9-7b

area is overridden: b) ***myBall.displayStatistics( )*** calls ***area*** in ***Ball***

Dynamic Binding and Abstract Classes

- Controlling whether a subclass can override a superclass method
  - Field modifier `final`
    - Prevents a method from being overridden by a subclass
  - Field modifier `abstract`
    - Requires the subclass to override the method
- Early binding or static binding
  - The appropriate version of a method is decided at compilation time
  - Used by methods that are `final` or `static`

Dynamic Binding and Abstract Classes

- Overloading methods
  - To overload a method is to define another method with the same name but with a different set of parameters
  - The arguments in each version of an overloaded method determine which version of the method will be used

Abstract Classes

- Example
  - CD player and DVD player
    - Both involve an optical disk
    - Operations
      - Insert, remove, play, record, and stop such discs

Abstract Classes

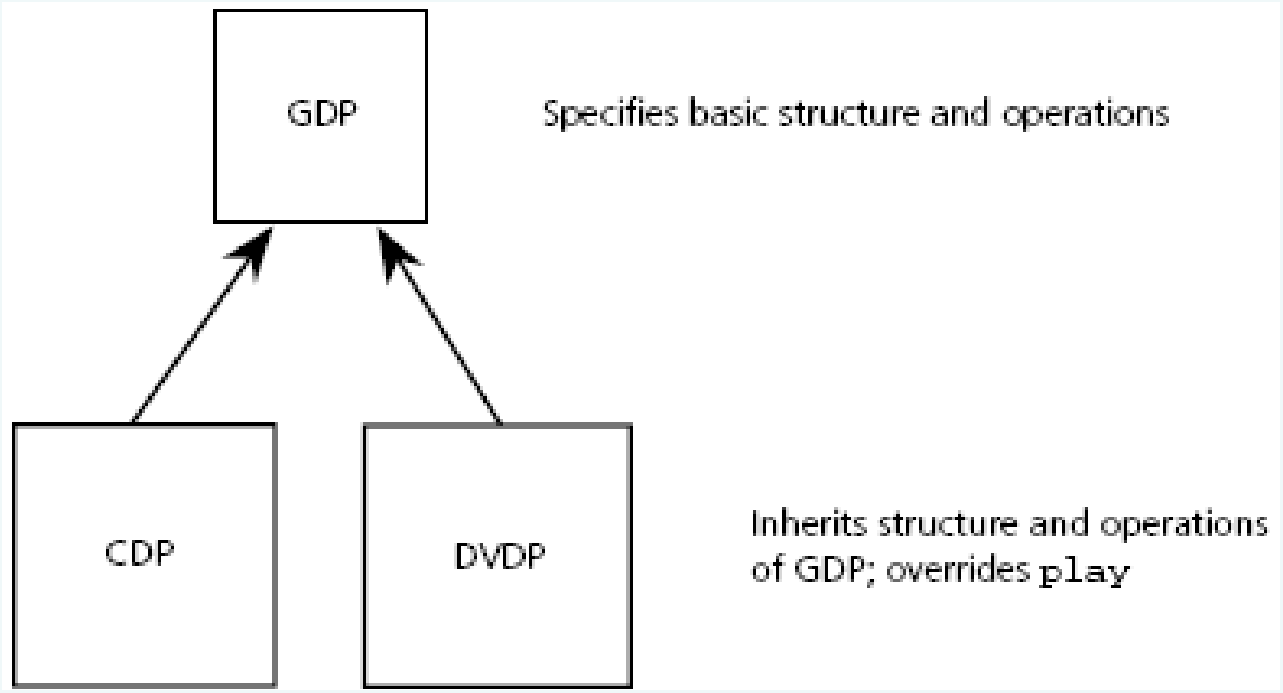

Figure 9-8

CDP and DVDP have an abstract base class GDP

Abstract Classes

- Abstract classes
  - An abstract class is used only as the basis for subclasses
    - It defines a minimum set of methods and data fields for its subclasses
  - An abstract class has no instances
  - An abstract class should, in general, omit implementations except for the methods that
    - Provide access to private data fields
    - Express functionality common to all of the subclasses

Abstract Classes

- Abstract classes (Continued)
  - A class that contains at least one abstract method must be declared as an abstract class
  - A subclass of an abstract class must be declared abstract if it does not provide implementations for all abstract methods in the superclass

Java Interfaces Revisited

- A Java interface
  - Specifies the common behavior of a set of classes
  - Common uses
    - Facilitate moving from one implementation of a class to another
      - A client can reference a class's interface instead of the class itself
    - Specify behaviors that are common to a group of classes

Java Interfaces Revisited

- Inheritance can be used to define a subinterface
- The Java API provides many interfaces and subinterfaces
  - Example: `java.util.Iterable`
    - An iterator is a class that provides access to another class that contains many objects

The ADTs List and Sorted List Revisited

- `BasicADTInterface`
  - Can be used to organize the commonalities between the ADT list and the ADT sorted list
  - `ListInterface`
    - A new interface based on `BasicADTInterface`

Implementation of the ADT Sorted List That Used the ADT List

- Operations
  - `createSortedList()`
  - `isEmpty():boolean {query}`
  - `size():integer {query}`
  - `sortedAdd(in newItem:ListItemType) throw ListException`
  - `sortedRemove(in anItem:ListItemType) throw ListException`
  - `removeAll()`
  - `get(in index:integer) throw ListIndexOutOfBoundsException`
  - `locateIndex(in anItem:ListItemType):integer {query}`

Implementations of the ADT Sorted List That Use the ADT List

- A sorted list is a list
  - With an additional operation, `locateIndex`
- A sorted list has a list as a member

Java Generics: Generic Classes

- ADT developed in this text relied upon the use of `Object` class
- Problems with this approach
  - Items of any type could be added to same ADT instance
  - ADT instance returns objects
    - Cast operations are needed
    - May lead to class-cast exceptions
- Avoid this issues by using Java generics
  - To specify a class in terms of a data-type parameter

Generic Wildcards

- Generic classes are not necessary related
- Generic `?` wildcard
  - Stands for unknown data type
- Example

```
public void process(NewClass<?> temp) {
  System.out.println("getData() => " +
   temp.getData());
} // end process
```

Generic Classes and Inheritance

- You can use inheritance with a generic class or interface
- Method overriding rules
  - Declare a method with the same parameters in the subclass
  - Return type is a subtype of all the methods it overrides
- It is sometimes useful to constrain the data-type parameter to a class or one of its subclasses or an implementation of a particular interface
  - To do so, use the keyword `extends`

Abstract Classes

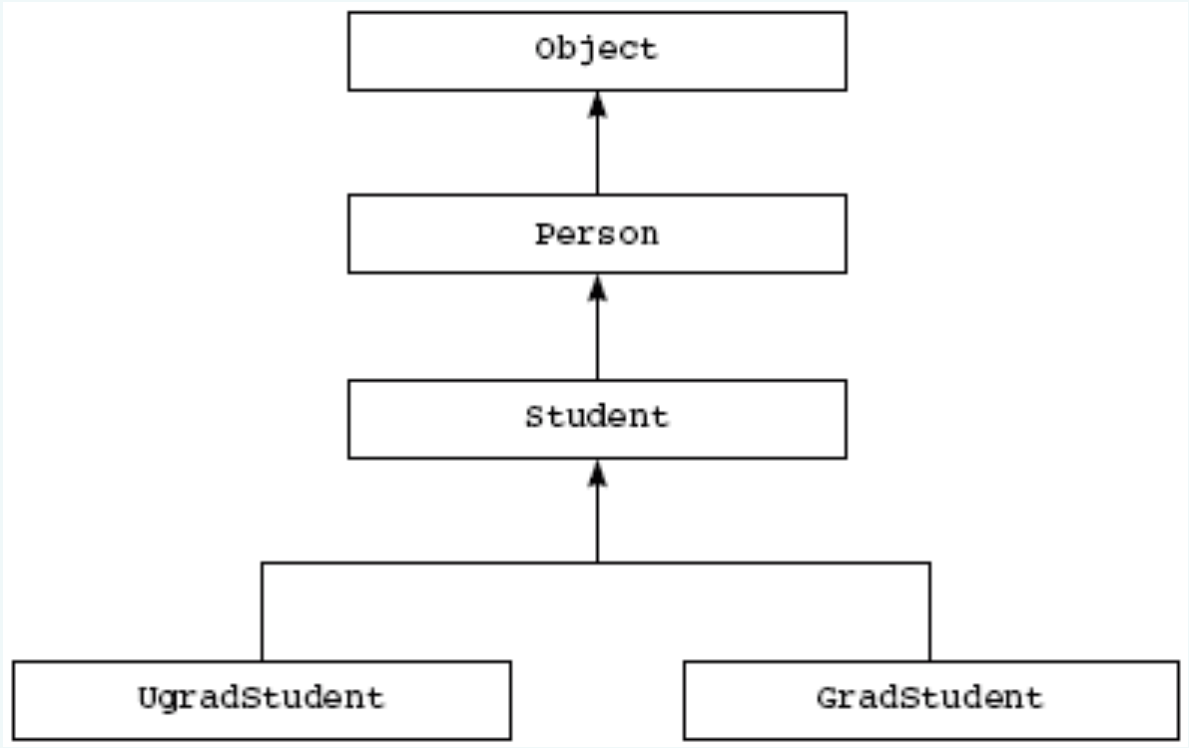

Figure 9-10

Sample class hierarchy

Generic Methods

- Method declarations can also be generic
  - Methods can use data-type parameters
- Generic methods are invoked like regular non-generic methods
- Example

```
public static <T extends Comparable<? super
  T>>
void sort(ArrayList<T> list) {
  // implementation of sort appears here
} // end sort
```

Iterators

- Iterator
  - Object that can access a collection of objects one object at a time
  - Traverses the collection of objects
- JCF defines generic interface `java.util.Iterator`
  - And a subinterface `ListIterator`

Iterators

- Defining our own Iterator class
- Implement an iterator interface
  - At a minimum, include methods for next( ), hasNext( ) and remove( ).
  - If you don't want to remove( ), you may leave method body empty.
- MyListIterator example
  - Maintain lastItemIndex to keep track of where iterator is between calls to iterator methods.
  - Initialize in constructor; increment inside next( ).

Summary

- A subclass inherits all members of its previously defined superclass, but can access only the public and protected members
- Subclasses and superclasses
  - A subclass is type-compatible with its superclass
  - The relationship between superclasses and subclasses is an is-a relationship
- A method in a subclass overrides a method in the superclass if they have the same parameter declarations

Summary

- An abstract method in a class is a method that you can override in a subclass
- A subclass inherits
  - The interface of each method that is in its superclass
  - The implementation of each nonabstract method that is in its superclass
- An abstract class
  - Specifies only the essential members necessary for its subclasses
  - Can serve as the superclass for a family of classes

Summary

- Early (static) binding: compiler determines at compilation time the correct method to invoke
- Late (dynamic) binding: system determines at execution time the correct method to invoke
- When a method that is not declared `final` is invoked, the type of object is the determining factor under late binding
- Generic classes enable you to parameterize the type of a class's data
- Iterators provide an alternative way to cycle through a collection of items