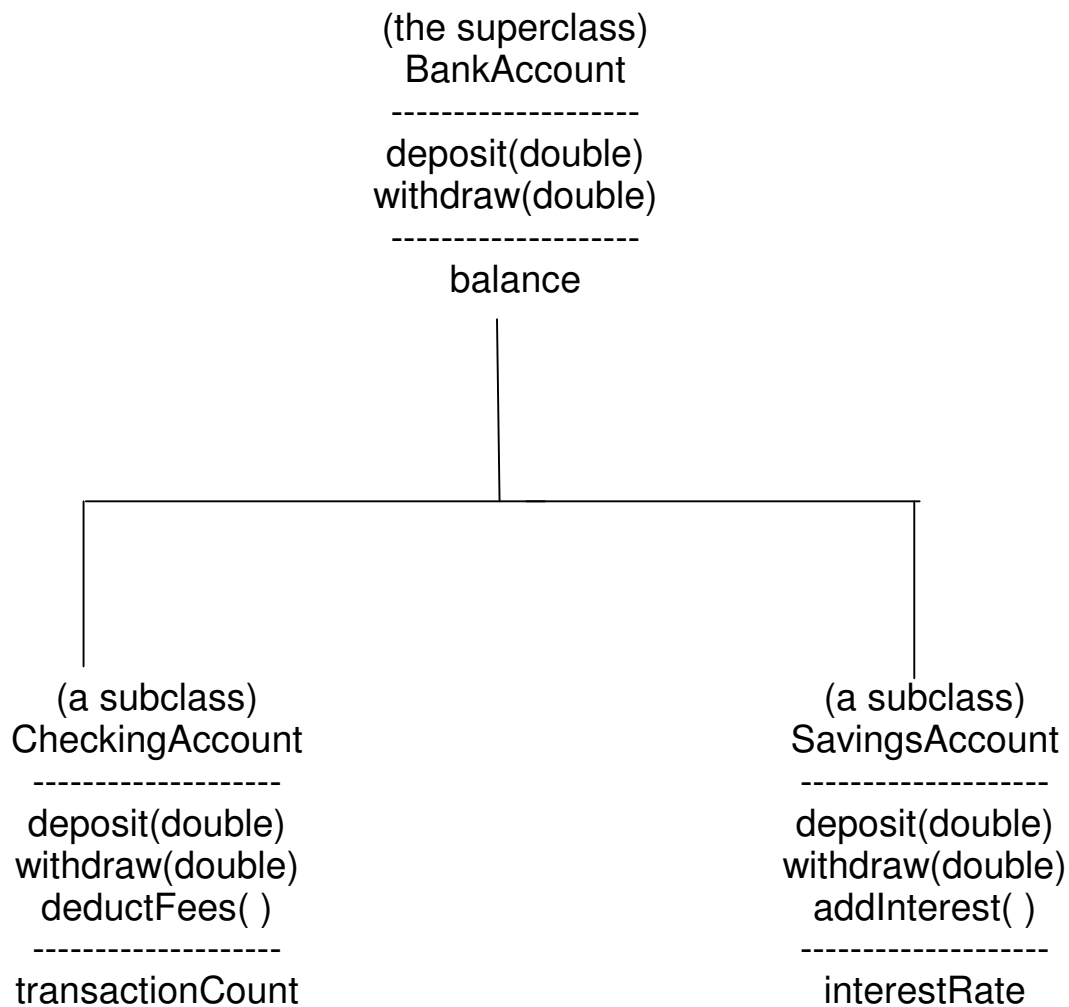


Inheritance

I. An Introduction to Inheritance (11.1)

Inheritance: extend classes by adding methods and fields

Example: a savings account *is* a bank account with interest.



```
class SavingsAccount extends BankAccount
{
    new methods – see next page
    new instance fields – see next page
}
```

SavingsAccount automatically inherits all methods and instance fields of BankAccount

```
SavingsAccount collegeFund = new SavingsAccount(10);
```

Note: This a Savings Account with 10% interest, the 10 would need to be stored an instance field using a SavingsAccount(**double**) constructor

```
collegeFund.deposit(500);
```

Note: It would be OK to use BankAccount deposit method with a SavingsAccount object but interest needs to be taken care of

Extended class = **superclass** (ex. BankAccount),
Extending class = **subclass** (ex. Savings)

Inheriting from class is different than implementing an interface:

- an interface has no behavior and state
- an inheriting subclass inherits all behavior and state

One advantage of inheritance is code reuse.

Note: Every class extends the Object class either directly or indirectly.
(More about this later this chapter.)

In a subclass, you can add new instance fields, new methods, and change or override methods.

ex.1)

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }
    public void addInterest( )
    {
        double interest = getBalance( )*interestRate/100;
        deposit(interest);
    }
    private double interestRate;
}
```

(Note: BankAccount has an instance field of balance.)

A SavingsAccount object inherits the balance instance field from BankAccount, and gains one additional instance field: interestRate.

SavingsAccount has a new method that was not in BankAccount called addInterest.

A) Inheriting Methods

1) Overridden method:

- Supply a different implementation of a method that exists in the superclass
- Must have same signature (same name and same parameter number and data types)
- If a method is applied to an object of the subclass type, the overriding method is executed

2) Inherited method:

- Don't supply a new implementation of a method that exists in superclass
- Superclass method can be applied to the subclass objects

3) Added method:

- Supply a new method that doesn't exist in the superclass
- New method can be applied only to subclass objects ex.) addInterest()

B).Inheriting Instance Fields

- Can't override fields
- Inherited field: All fields from the superclass are automatically inherited but are probably directly inaccessible due to being private
- Added field: Supply a new field that doesn't exist in the superclass
- What if you define a new field with the same name as a superclass field?
 - Each object would have two instance fields of the same name where the fields could hold different values
 - Legal but extremely undesirable

ex.2) Implementing the CheckingAccount Class

- Overrides deposit and withdraw to increment the transaction count:
- **public class** CheckingAccount extends BankAccount
- {
- **public void** deposit(double amount) { . . . } //overrides BankAccount method
- **public void** withdraw(double amount) { . . . } //overrides BankAccount method
- **public void** deductFees() { . . . } // new method
- **private int** transactionCount; // new instance field
- }

Each CheckingAccount object has two instance fields:

- balance (inherited from BankAccount)
- transactionCount (new to CheckingAccount)

You can apply four methods to CheckingAccount objects:

- getBalance() (inherited from BankAccount)
- deposit(**double** amount) (overrides BankAccount method)
- withdraw(**double** amount) (overrides BankAccount method)
- deductFees() (new to CheckingAccount but can not be used with a BankAccount object)

ex.3) Consider deposit method of CheckingAccount

```
public void deposit(double amount)
{
    transactionCount++;
    // now try to add amount to balance: balance += amount;
    . . .
}
```

Can not just add amount to balance (balance += amount;):

- balance is a *private* field of the superclass BankAccount
- A subclass has no access to private fields of its superclass
- A subclass must use a public interface (method) to access private fields, just like any other class has to do, even though the fields were “inherited”

C) Invoking a Superclass Method

Can't just call the BankAccount deposit(amount) because there is a deposit method in CheckingAccount. That would be the same as **this.deposit(amount)** and would end up calling the same CheckingAccount deposit method (infinite recursion).

*Instead, invoke *superclass method*: **super.deposit(amount)**;
This line now calls deposit method of the BankAccount class

Complete method:

```
public void deposit(double amount)
{
    transactionCount++;
    // Now add amount to balance by using the BankAccount deposit method:
    super.deposit(amount);
}
```

D. Common Error: Shadowing Instance Fields

Recall: A subclass has no access to the private instance fields of the superclass.

ex.1) Beginner's error: rookies "solve" this problem by adding another instance field with the same name:

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount)
    {
        transactionCount++;
        balance = balance + amount;
    }
    . . .
    private double balance; // Don't do!
}
```

Now the deposit method compiles, but it doesn't update the correct balance (the balance which is the instance field of BankAccount).

E) Subclass Construction of the Constructor Method

super followed by a parenthesis indicates a call to the superclass constructor

ex.1)

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        super(initialBalance); // this calls the superclass constructor
        transactionCount = 0; // Initialize transaction count
    }
    ...
}
```

Use of **super**:

- The use of **super** to call a superclass constructor must be the *first* statement in subclass constructor.
- If a subclass constructor doesn't call the superclass constructor, the default superclass constructor is used. (Recall: the default constructor is the constructor with no parameters.) If all constructors of the superclass require parameters, then the compiler reports an error when defaulting to the default constructor.

F) Converting Between Subclass and Superclass Types

It is OK to convert subclass reference to superclass reference

ex.1)

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
Object anObject = collegeFund;
```

Note: The three object references stored in `collegeFund`, `anAccount`, and `anObject` all refer to the same object of type `SavingsAccount`

G) The SuperClass of all SuperClasses - **Object**: The Cosmic Superclass (11.8)

In Java, every class without an explicit **extends** inherits the Object class.

The most useful methods in the Object class are toString(), equals(Object obj), and clone()

- String toString()
toString is called whenever you concatenate a string with an object and returns the string representation of object state
- **boolean** equals(Object otherObject)
checks if the state (not the reference) of the objects are the same
- Object clone() (not an AP topic - see Advanced Topic 11.6)
Makes a new object which is a *copy* of the object

ex.1)

```
BankAccount account2 = account; // just reference is copied
BankAccount clonedAccount = (BankAccount) account.clone( );
// new object now created
// Note: Must cast return value because return type is type Object
```

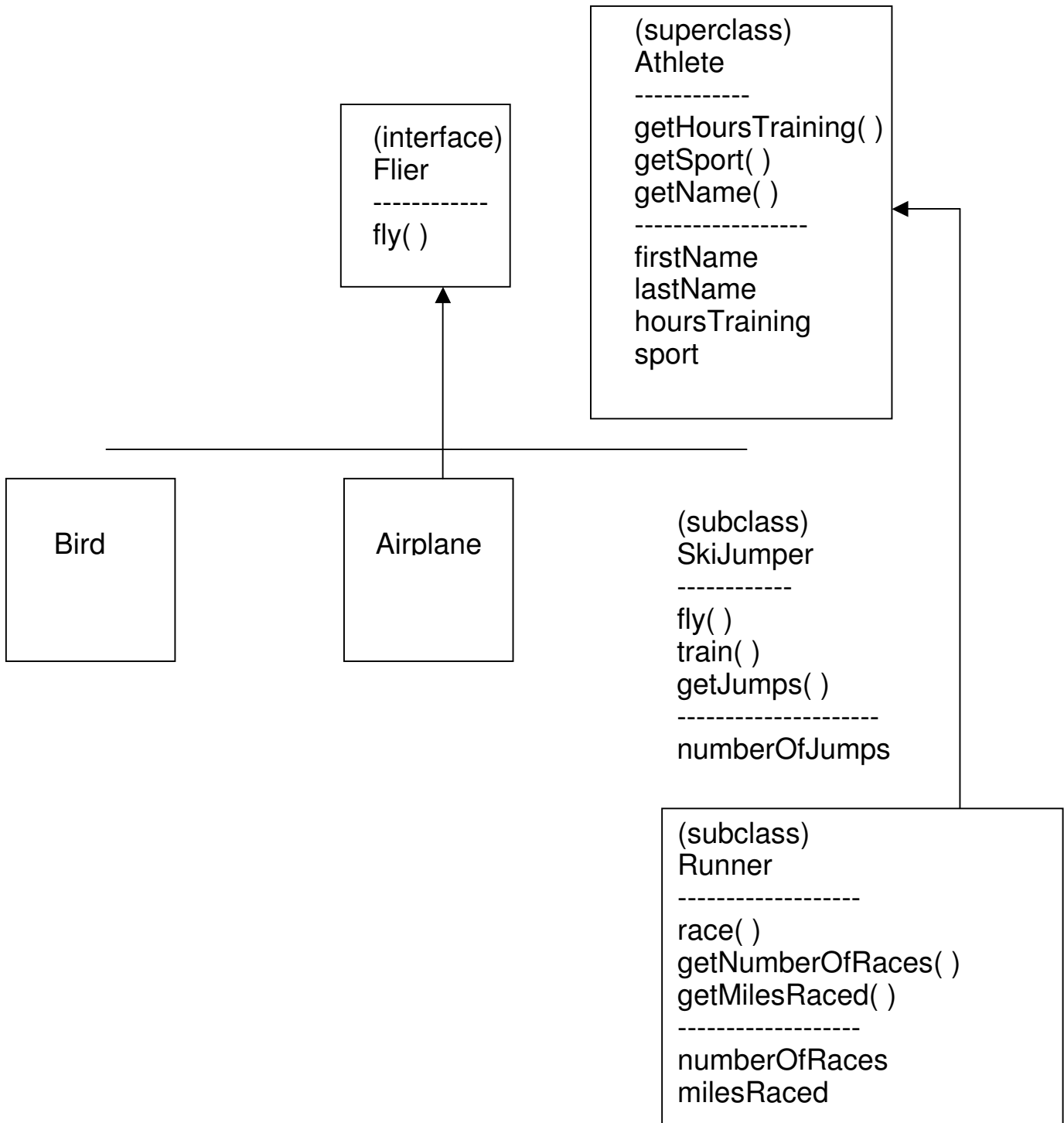
Note: It is a good idea to override these methods in your classes. See the next two overheads.

AP Programming - Chapter 11 Lecture

1) Overriding the toString Method

Recall: `System.out.println("x = " + 10);` For this concatenation to work the `+` operator converts the 10 to a string and the `toString` method is not used in this case.

The `toString` method is automatically called by `System.out.println`.



If our inheritanceTest client program invokes the statement:

```
System.out.println(jumper1);
```

Object's toString method is called and a cryptic message about SkiJumper is printed to the screen such as: SkiJumper@d24606bf (a reference to where the object is)

However, we may wish to print a meaningful message about SkiJumper that includes values of the instance variables. This will allow us to inspect the object for possible errors in our program. In order to do this, SkiJumper must override the Object toString method.

```
public String toString( )  
{  
    return "SkiJumper numberOfJumps = " + numberOfJumps;  
}
```

The statement

```
System.out.println(jumper1);
```

will now print:

```
SkiJumper numberOfJumps = 2.
```

It may be more useful to provide a toString that prints the values of all instance fields. A toString method for Athlete might be:

```
public String toString( )  
{  
    return ("class Athlete: \nname = " + firstName + " " + lastName  
        + "\nsport = " + sport + "\nhoursTraining = " + hoursTraining);  
}
```

and the more inclusive toString method for SkiJumper would be:

```
public String toString( )  
{  
    String s = super.toString( );  
    return (s + "\nclass SkiJumper:\nnumberOfJumps = " + numberOfJumps);  
}
```

The statement `System.out.println(jumper2);` would now print:

```
class Athlete:  
name = Jane Dole  
sport = Ski Jumping  
hoursTraining = 0.0  
class Skilumper:  
numberOfJumps = 3
```

You can include whatever meaningful information you need when you override Object's `toString` method. It is a good programming habit to supply a `toString` method in the classes that you write.

Common notation:

You can also do the following inside of the new **public** `String toString()` method
return "SkiJumper[numberOfJumps = " + numberOfJumps + "];

2) Overriding the equals Method

If you wish to test whether two objects have equal states (have the same contents), you need to override Object's `equals` method. An `equals` method for Athlete would be:

```
public boolean equals(Object other)  
{  
    Athlete temp = (Athlete) other;  
    return ((firstName.equals(temp.firstName)) && (lastName.equals (temp.lastName))  
        && (sport.equals(temp.sport)) && (hoursTraining == temp.hoursTraining));  
}
```

The parameter `other` needs to be cast to Athlete. Since three of the instance fields are objects (Strings), they need to be compared using the String `equals` method. The `hoursTraining` instance field is a number. Numbers are compared using the `==` operator. Objects are compared using `equals()`.

Although the implementation of the `equals` method is not part of the AP CS testable Java subset, you do need to understand the difference between object equality (`equals`) and identity (`==`) - see the next 2 pages.

3) Comparing Strings

A) Case insensitive test ("Y" or "y")

```
if (input.equalsIgnoreCase("Y"))
```

s.compareTo(t) < 0 means: s comes before t in the dictionary

Note: "car" comes before "cargo"

Note: All uppercase letters come before lowercase:

ex.1) "Hello" comes before "car"

B) == vs. equals()

Don't use == for strings!

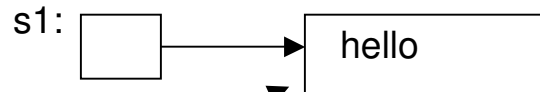
```
if (input == "Y") // WRONG!!!
```

Use equals method:

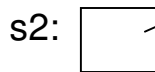
```
if (input.equals("Y"))
```

== tests identity, equals tests equal contents

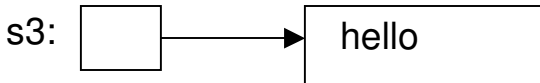
ex.1) String s1 = "hello";



String s2 = s1;



String s3 = **new** String("hello");



<u>Expression</u>	<u>Result</u>
a) s1 == s2	true
b) s1.equals(s2)	true
c) s1 == s3	false
d) s1.equals(s3)	true
e) s2 == s3	false
f) s2.equals(s3)	true

4) Comparing Objects

recall: == tests for identity, equals() for identical content

ex.1)

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
```

```
Rectangle box2 = box1;
```

```
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

box1 == box2 is true also

box1.equals(box2) is true

box1 == box3 is false but

box1.equals(box3) is true!

Note: The equals() method must be defined - i.e. code written for it - in the class. The String class has a built-in, already written, equals() method but other classes do not.

5) Testing for null

Note: null reference refers to no object

```
String middleInitial = null; // Not set
```

```
if ( . . . )
```

```
    middleInitial = middleName.substring(0, 1);
```

```
    .
```

```
    .
```

```
    .
```

```
if (middleInitial == null)
```

```
    System.out.println(firstName + " " + lastName);
```

```
else
```

```
    System.out.println(firstName + " " + middleInitial + ". " + lastName);
```

Note: null is not the same as the empty string ""

*Note: Use ==, not equals(), to test for null

H) Converting Between Subclass and Superclass Types

Superclass references don't know the full story:

ex.1) (Recall: addInterest is a method found only in SavingsAccount)

```
anAccount.deposit(1000); // OK
```

```
collegeFund.addInterest( ); // OK
```

```
anAccount.addInterest( );
```

```
// Illegal - not a method of the class to which anAccount belongs
```

When you convert between a subclass object to its superclass type:

The value of the reference stays the same - it is the memory location of the Object. But, *less information is known* about the object.

Why would anyone want to know *less* about an object?

Answer: To reuse code that knows about the superclass but not the subclass:

```
public void transfer(double amount, BankAccount other)
{
    withdraw(amount);
    other.deposit(amount);
}
```

Can be used to transfer money from any type of BankAccount instead of having to write two methods:

```
public void transfer(double amount, CheckingAccount other)
```

```
public void transfer(double amount, SavingsAccount other)
```

I) Converting Between Superclass and Subclass Types

Occasionally you may need to convert from a superclass reference to a subclass reference

ex.1) `BankAccount anAccount = (BankAccount) anObject;`

This cast is dangerous: if you are wrong and anObject is not a BankAccount reference or subclass of BankAccount, an exception is thrown

Solution: Use the **instanceof** operator.

instanceof tests whether an object belongs to a particular Class or any above it.

ex.1)

```
if (anObject instanceof BankAccount)
{
    BankAccount anAccount = (BankAccount) anObject;
    ...
}
```

ex.2) Given:

Baseball inherits from Sport i.e. Baseball extends Sport

Baseball baseballObj = **new** Baseball();

Are the following true or false?

- | | |
|---|-------|
| a) baseballObj instanceof Baseball | True |
| b) baseballObj instanceof Sport | True |
| c) baseballObj instanceof Object | True |
| d) baseballObj instanceof Swimming | False |

Abstract Classes (11.6)

An abstract method is a method whose implementation is not specified. An abstract class is a class that contains an abstract method. Abstract classes cannot be instantiated. Abstract classes can be used if you want to force programmers to define a method. Let's redefine our Athlete class one more time to demonstrate an abstract class. In the following example, Athlete is an abstract class which means it cannot be instantiated.

ex.1)

```
public abstract class Athlete
{
    public Athlete(String fName, String lName, String sportPlayed)
    {
        firstName = fName;
        lastName = lName;
        sport = sportPlayed;
        hoursTraining = 0;
    }
    // other methods here

    public abstract String getDietPlan( );

    // other code here
}
```

this example mandates that any subclass of Athlete must implement getDietPlan. If the subclass does not define this abstract method, then the subclass that is instantiated must be declared as abstract itself. Runner might define getDietPlan in the following way.

```
public String getDietPlan( )
{
    return ("25% protein, 65% carbohydrates, 10% fat");
}
```

An abstract class differs from an interface because an abstract class can have instance variables and concrete (implemented) methods that are inherited by subclasses. Although you cannot instantiate an abstract class, you can still have an object reference whose type is an abstract class. For example, the following assignment is legal.

```
Athlete jumper3 = new Ski Jumper("Mark", "Morris"); // This is OK.
```

J) Compiling

Compiler needs to check that only legal methods are invoked.

ex.1)

```
Object anObject = new BankAccount( );
anObject.addInterest( ); // Wrong! No addInterest( ) method in BankAccount,
                        // a compiler error occurs
```

ex.2)

Consider the following code segment and assume MyClass has a print() and a toString() method.

```
1 Object obj = new MyClass( );
2 obj.print( );
3 ((MyClass) obj).print( );
4 System.out.println(obj.toString( ));
```

Which of the statements below is true about the program compilation?

- a. There is a compile-time error in line 1.
- b. There is a compile-time error in line 2.
- c. There is a compile-time error in line 3.
- d. There is a compile-time error in line 4.
- e. There is no error in any of the lines.

Answer: b

Reason: There is no print() method in the Object class so it will not compile. The issue is the compiler, not the object. Objects only exist at RUNTIME, and you can not get to RUNTIME until after you compile. It will not compile if there is not a method in the designated class. A MyClass "is-a" Object, so line 1 is legal. But you could replace MyClass with any class and it would still be legal. Therefore the compiler only knows for sure that obj is an Object, which does not have a print() method. But line 3 would compile because it now knows that obj is a MyClass object and there is a print() method in My Class.

Now, lets imagine that MyClass has overridden the toString() method. If you change obj.print(); to System.out.println(obj.toString()); then the code will compile since Object has a toString() method. At RUNTIME, obj will know that it is a MyClass and use the toString() method from the MyClass class instead of the toString() from the Object class (late binding).

K) Polymorphism

In Java, the type of a variable doesn't completely determine the type of object to which it refers. Method calls are determined by type of actual object, not the type of object reference.

ex.1)

```
BankAccount aBankAccount = new SavingsAccount(1000);  
    // aBankAccount holds a reference to a SavingsAccount  
BankAccount anAccount = new CheckingAccount( );  
anAccount.deposit(1000); // Calls "deposit" from CheckingAccount
```

*Using a reference of one class to refer to objects of a subclass only becomes useful in the context of polymorphism.

ex.2)

Suppose, for example I have a bunch of different printers: an HP100, an Epson200 and an Okidata300. Let's suppose each of those printers can print in regular, italics and bold, and can print in various point sizes, and in various character/inch settings. I could write three classes, Hp100, Epson200, and Oki300. Within each of those classes I could include these methods:

```
public void setitalics (boolean onoff);  
public void setbold (boolean onoff);  
public void setpt (int pt);  
public void setcpi (int cpi);  
public void display (String text);
```

Using these methods I can tell a printer to turn bold/italics on/off, the points size to use, the cpi setting, and then send some text (using the display method). I can change settings, and send some more text. The setbold methods in each of the printers will send the proper command (typically an escape sequence) to the printer. The commands will likely be different for the various printers.

Okay, now I want to be able to print a document. So I want a single print method (in say a Document class) that takes a printer reference and sends commands (using the reference) to a printer. I would define a Printer class with a printer method. But what shall the parameter be? If I choose a particular printer type,

```
public void print (Oki300 printer)
```

then it will won't work with any or the other printers. So, I will create a Printer class:

```
public class Printer
{
    public abstract void setitalics (boolean onoff);
    public abstract void setbold (boolean onoff);
    public abstract void setsize (int size);
    public abstract void setcpi (int cpi);
    public abstract void display (String text);
}
```

Now let my three concrete printer classes (Hp100, Epson200, Oki300) each extend Printer. For example:

```
public class Hp100 extends Printer
{
    ...
}
```

Each of the classes would include the appropriate form of each method for the corresponding printer.

Now, the print method (in the Document class) will look like this:

```
public void print (Printer printer)
{
    // code here
}
```

That is, the print method expects a reference to Printer object. In fact, it will receive a reference to an Hp100, Epson200, or Oki300 object, but all of those classes are subclasses of Printer, so the printer reference is allowed to refer to them.

Within the print method there will be mention of any particular kind of printer (Hp100, Epson200, Oki300). All reference will be to printer. We know that Hp100, Epson200, and Oki300 all respond to the messages specified in the printer class (each provides concrete implementation for the abstract methods): setbold, setitalics, setsize, setcpi, and display.

Not only will the print method work with those three printers, but it will also work with any other printer if I write an appropriate class. If I want to use a FujiSX printer I write a FujiSX class that extends Printer and implements the five methods. The print method will then happily take a reference to a FujiSX printer and work with it!

L) Access Control (11.7)

Java has four levels of controlling access to fields, methods, and classes:

- **private** access
 - Can be accessed only by the methods of their own cla
- **protected** access
 - (not an AP topic) protected data can be accessed by all subclass methods and package methods which is its major weakness. The protected level of visibility allows for access by all subclasses and all classes in the same package. This method of access control is explained in *Advanced Topic 11.3* of your text.
- **package** access
 - The default, when no access modifier is given
 - Can be accessed by all classes in the same package
 - Good default for classes, but extremely unfortunate for fields
- **public** access
 - Can be accessed by methods of all classes

Recommended Access Levels

- Instance and static fields: Always private. Exceptions:
 - public static final constants are useful and safe (due to being final)
 - Some objects, such as System.out, need to be accessible to all programs (public)
 - Occasionally, classes in a package must collaborate very closely (give some fields package access); using inner classes (classes inside another class is usually better)
- Methods: public or private (usually public)
- Classes and Interfaces: public or package
 - Better alternative to package access: inner classes
 - In general, inner classes should not be public (some exceptions exist, e.g., Ellipse2D.Double)

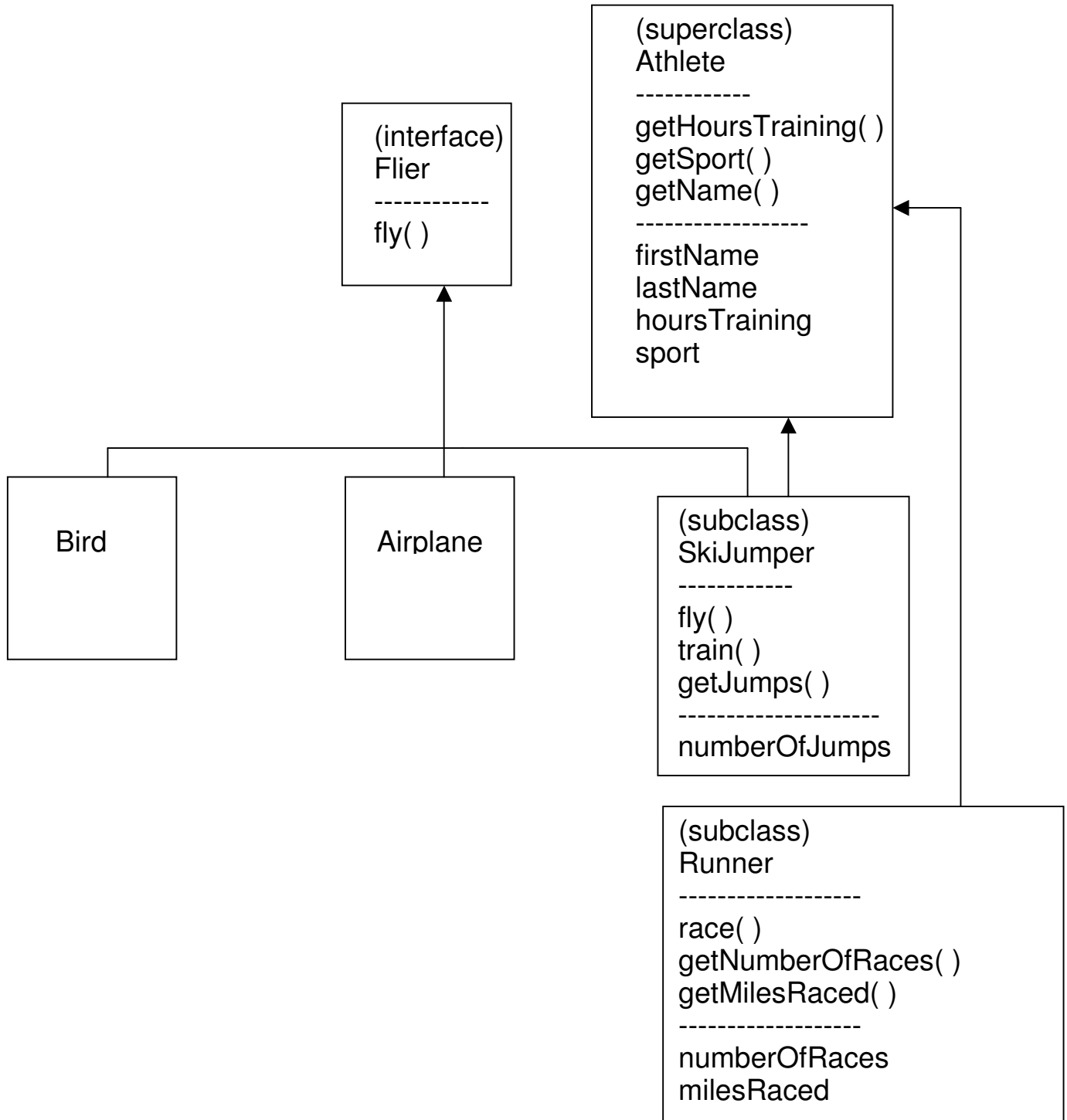
*Note:

- 1) Beware of accidental package access by forgetting the public or private modifier
- 2) A subclass can never have an access control modifier more restricted than the superclass for an overloaded method.
- 3) The AP CS Java subset does not include protected and package visibilities.

ex.1) Protected example: (see next page)

In our Athlete example, if we declared hoursTraining as protected, then the Runner and the Ski Jumper would be able to access and update this instance field of Athlete directly. In our example, the Athlete's train method is called to update this variable because it is private.

Big Example:



```
public interface Flier
{
    void fly( );
}

public class Athlete
{
    public Athlete (String fName, String lName, String sportPlayed)
    {
        firstName = fName;
        lastName = lName;
        sport = sportPlayed;
        hoursTraining = 0;
    }
    public void train(double hours)
    {
        System.out.println("Athlete training for " + hours + " hours.");
        hoursTraining += hours;
    }
    public String getName( )
    {
        return (firstName + " " + lastName);
    }
    public String getSport( )
    {
        return sport;
    }
    public double getHoursTraining( )
    {
        return hoursTraining;
    }
    private String firstName;
    private String lastName;
    private String sport;
    private double hoursTraining;
}
```

```
public class SkiJumper extends Athlete implements Flier
{
    public SkiJumper(String first, String last)
    {
        super(first, last, "Ski Jumping");
        numberOfJumps = 0;
    }
    public void fly( )
    {
        System.out.println("Using skis to take " + getName( ) + " into the air.");
        numberOfJumps++;
    }
    public void train(double hours)
    {
        System.out.println("I am on the slopes " + hours + " hours today.");
        super.train (hours);
    }
    public int getJumps( )
    {
        return numberOfJumps;
    }
    private int numberOfJumps;
}
```

```
public class Runner extends Athlete
{
    public Runner(String first, String last)
    {
        super(first, last, "Running");
        numberOfRaces = 0;
        milesRaced = 0;
    }
    public void race (double raceLength)
    {
        System.out.println(getName( ) + " is racing in a " + raceLength + " mile race.");
        numberOfRaces++;
        milesRaced += raceLength;
    }
    public int getRaces( )
    {
        return numberOfRaces;
    }
    public double getMilesRaced( )
    {
        return milesRaced;
    }
    private int numberOfRaces;
    private double milesRaced;
}
```

Notice that the Runner does not override train. The runner trains as a generic athlete trains but, in addition to hours of training, a Runner keeps track of hours racing and number of races run. The behavior and state of a Runner differ from those of the Ski Jumper.

Although the subclass inherits the private instance fields of the superclass, the subclass has no access to the private fields of the superclass. *Private* means that only the class in which the instance field or method is defined has access.

AP Programming - Chapter 11 Lecture

SkiJumper class implements the Flier interface and inherits (extends) the Athlete class. Runner class inherits (extends) the Athlete class.

Answer the following:

- 1) Is the following a subclass or super class?
 - a) SkiJumper Answer: subclass
 - b) Athlete Answer: superclass
 - c) Runner Answer: subclass
 - d) Flier Answer: neither - not a class

- 2) What are the inherited instance fields of the following?
 - a) SkiJumper Answer: firstName, lastName, sport, hoursTraining
 - b) Athlete Answer: none inherited
 - c) Runner Answer: firstName, lastName, sport, hoursTraining
 - d) Flier Answer: none - can not have any

- 3) What are the instance fields that are not inherited of the following?
 - a) SkiJumper Answer: number of jumps
 - b) Athlete Answer: firstName, lastName, sport, hoursTraining
 - c) Runner Answer: numberOfRaces, milesRaced
 - d) Flier Answer: none - can not have any

- 4) What are the *inherited* methods of the following that will not be overwritten?
 - a) SkiJumper Answer: getName(), getSport(), getHoursTraining()
 - b) Athlete Answer: none inherited
 - c) Runner Answer: getName(), getSport(), getHoursTraining()
 - d) Flier Answer: none - can not have any

- 5) What are the *inherited* methods of the following that must be overwritten?
 - a) SkiJumper Answer: train()
 - b) Athlete Answer: none inherited
 - c) Runner Answer: none
 - d) Flier Answer: none - can not have any

- 6) What are the methods of the following that will need to be implemented (coded) due to originally being abstract?
 - a) SkiJumper Answer: fly()
 - b) Athlete Answer: none
 - c) Runner Answer: none
 - d) Flier Answer: none - fly() is abstract but can not be coded

AP Programming - Chapter 11 Lecture

- 7) What methods were not from an interface nor inherited of the following?
a) SkiJumper Answer: getJumps ()
b) Athlete Answer: none inherited
c) Runner Answer: race(), getRaces(), getMilesRaced()
d) Flier Answer: fly()
- 8) Can Runner implement SkiJumper? (yes/no)
Answer: no - SkiJumper has instance fields and nonabstract methods
- 9) Can SkiJumper implement Runner? (yes/no)
Answer: no - SkiJumper has instance fields and nonabstract methods
- 10) Could Athlete implement flier? (yes/no)
Answer: yes - but only if fly() were implemented
- 11) From which of the following can an object be created?
a) SkiJumper
b) Athlete
c) Runner
d) Flier
Answer: all but Flier due to the abstract methods
- 12) Can a superclass have an abstract method? (yes/no)
Answer: yes - but no object could be created for it
- 13) Can a Runner object use the method fly()? (yes/no)
Answer: no - fly() is not a method associated with a Runner object.
- 14) Does a Runner object's train() method have to be the same as a SkiJumpers?
(yes/no)
Answer: no - the code would probably be different

In our examples, we update the private instance field `hoursTraining` of the superclass `Athlete` when the `Runner` trains and when the `SkiJumper` trains. This is done by invoking the public method `train` of the superclass thus preserving encapsulation. Each subclass does this in a different way.

- The `Runner` subclass does not have a `train` method defined. The `train` method that is invoked by `Runner` is the `train` method inherited from `Athlete`. `Athlete`'s `train` method updates `Athlete`'s own private instance variable `hoursTraining` that is inherited by the `Runner`.
- The `SkiJumper` implements a method `train` with the same signature as the method `train` of its superclass, `Athlete`. The `SkiJumper`'s `train` method overrides the `Athlete`'s `train` method. In order to update the inherited private instance field `hoursTraining`, the `SkiJumper`'s `train` method invokes the superclass `train` method with the call `super.train()`. The **super** keyword can be used to call a method of the superclass that is overridden by a subclass.

Subclasses should not override instance fields. Subclasses can inherit instance fields and define new instance fields but should not override existing instance fields. The consequences of overriding instance fields are explained in Section 11.3 and *Common Error 11.2* in your text.

Subclass Construction 11.4

The constructor for `Athlete` has three string parameters, a first name, a last name, and a sport. The `Athlete` constructor initializes the appropriate instance fields with the values of these parameters and initializes `hoursTraining` to 0.

When the `Runner` or `SkiJumper` constructor is called, two `String` parameters are passed, the first and last names of the athlete.

For example:

```
Runner racer1 = new Runner("Joe", "Thomas");  
SkiJumper jumper = new SkiJumper("Jane", "Smith");
```

The `Runner` and `SkiJumper` constructors invoke the `Athlete` constructor passing these two parameters and the string "Running" or "Ski Jumping" (the sport name) to the superclass constructor.

```
super(first, last, "SkiJumping");  
super(first, last, "Running");
```

Recall: The use of `super` to call a superclass constructor must be the *first* statement in subclass constructor. If a subclass constructor doesn't call the superclass constructor, the default superclass constructor is used and is done first thing. However if there is no default constructor (as with `Athlete`), an error will occur.

Note: The instance fields that are not inherited should be initialized in the subclass constructor.

ex.1) Give the classes below, what output would be produced if the following code was executed?

```
Subclass obj1 = new Subclass( );  
Subclass obj2 = new Subclass(5);  
Subclass obj3 = new Subclass(5, 10);
```

(Recall: If the super constructor call is not used then the default constructor of the super class is the *first thing automatically* called.)

```
public class TheSuperClass  
{  
    public TheSuperClass( )  
    {  
        System.out.print("Hello ");  
    }  
    public TheSuperClass(int x)  
    {  
        System.out.print("GoodBye ");  
    }  
}  
  
public class Subclass extends TheSuperClass  
{  
    public Subclass( )  
    {  
        System.out.print("Hi ");  
    }  
    public Subclass(int x)  
    {  
        System.out.print("Bye ");  
    }  
    public Subclass(int a, int b)  
    {  
        super(a);  
        System.out.print("Adios ");  
    }  
}
```

Answer:

Hello Hi Hello Bye GoodBye Adios

ex. 2) Converting from Subclasses to Superclasses (11.5)

If we wanted to have the ability to compare ski jumpers based on the number of jumps completed, the SkiJumper class would realize the Comparable interface

public class SkiJumper **extends** Athlete **implements** Flier, Comparable

and implement the compareTo method

```
public int compareTo(Object other)
{
    SkiJumper temp = (SkiJumper) other;           // *note the casting needed
    if (numberOfJumps < temp.getJumps( ))
        return -1;
    if (getJumps( ) > temp.getJumps( ))
        return 1;
    return 0;
}
```

Since we want to compare the number of jumps of two SkiJumpers, we cast the object parameter to a SkiJumper.

Suppose the following declarations were made.

```
SkiJumper jumper1 = new SkiJumper("John", "Miller");  
SkiJumper jumper2 = new SkiJumper("Jane", "Dole");  
Athlete jumper3 = new SkiJumper("Mark", "Morris");  
Runner racer1 = new Runner("Mary", "Smith");
```

The following calls to `compareTo` are legal:

```
if (jumper1.compareTo(jumper2) > 0) // Code here  
if (jumper2.compareTo(jumper1) > 0) // Code here  
if (jumper1.compareTo(jumper3) > 0) // Code here
```

However, the statement

```
if (jumper3.compareTo(jumper1) > 0) // Code here  
// WRONG!
```

will not compile, `jumper3` is an `Athlete` and `compareTo` is not defined in the `Athlete` class.

The statement `if (jumper1.compareTo(racer1) > 0) // Code here`

will compile but will throw a `ClassCastException` at runtime because the `compareTo` method of the `SkiJumper` class tries to cast a `Runner` to a `SkiJumper` in the first statement.

The statements

```
jumper3.fly( );  
racer1.fly( );
```

will cause a compile-time errors because `fly` is not defined for `Athlete` or for `Runner`.

The statement: `jumper3.train(2);`

is a legal statement because `train` is defined for `Athlete`. The `train` method of the `SkiJumper` class is invoked. Method calls are always determined by the type of the actual object (`SkiJumper`), not the type of the object reference (`Athlete`). This is an example of late (dynamic) binding.

The assignment

`jumper3 = jumper1; // OK because SkiJumper is a subclass of Athlete`
is legal. An object of a subclass can be assigned to an object of its superclass.

The assignment

`jumper1 = (SkiJumper)jumper3 // OK, cast necessary`
is legal. An object of a superclass can be assigned to an object of its subclass with proper casting.

ex.3) below is a sample client program to test these concepts.

```
public class InheritanceTest
{
    public static void main(String[ ] args)
    {
        SkiJumper jumper1 = new SkiJumper("John", "Miller");
        SkiJumper jumper2 = new SkiJumper("Jane", "Dole");
        Athlete jumper3 = new SkiJumper("Mark", "Morris");
        jumper1.fly( );
        jumper1.fly( );
        jumper2.fly( );
        jumper2.fly( );
        jumper2.fly( );
        if (jumper1.compareTo(jumper2) > 0)
        {
            System.out.println(jumper1.getName( ) + " is better than "
                + jumper2.getName( ) + ".");
        }
        else
        if (jumper1.compareTo(jumper2) < 0)
        {
            System.out.println(jumper2.getName( ) + " is better than "
                + jumper1.getName( ) + ".");
        }
        else
        {
            System.out.println("The jumpers have completed an equal
                number of jumps!");
        }
        jumper3.train(2) ;
    }
}
```

AP Programming - Chapter 11 Lecture

The results of program execution are:

Using skis to take John Miller into the air.

Using skis to take John Miller into the air.

Using skis to take Jane Dole into the air.

Using skis to take Jane Dole into the air.

Using skis to take Jane Dole into the air.

Jane Dole is better than John Miller.

I am on the slopes 2.0 hours today.

Athlete training for 2.0 hours.

Final Methods and Final Classes:

Methods with a modifier of **final** cannot be overridden. ex.1) **final public void** ex1()

Classes with a modifier of **final** cannot be extended. ex.2) **final public class** Ex2

Expanded Coverage of Material That Is Not Found in the Text

*The inheritance relationship between a subclass and its superclass is an **IS-A** relationship. The subclass is more specific than the superclass. For example,

- A SkiJumper *IS A* Athlete
- A Runner *IS A* Athlete
-

Not all athletes are runners. Not all athletes are ski jumpers. A good check for subclass selection is to apply this *IS-A* phrase. Ask yourself if your *IS-A* statement is true. Chapter 16 of your text covers *IS-A* relationships in more detail.

An object can be composed of other objects. For example, an Athlete has a first name, a last name, and a sport. All three of these instance fields are string objects. This situation is an example of a *HAS-A* relationship. *HAS-A* relationships are used when a class is composed of other types of object. This is often referred to as composition. For example,

- A Ski Jumper *HAS-A* name.
- A DeckOfPlayingCards *HAS-A* Card.
- An Airplane *HAS-A* Engine.

(Note: Chapter 16 of your text covers *HAS-A* relationships in more detail.)

Subclasses and constructor rules:

Recall: Use of **super**:

- The use of `super` to call a superclass constructor must be the *first* statement in subclass constructor.
- If a subclass constructor doesn't call the superclass constructor, the default superclass constructor is automatically used so that all instance fields are initialized. (Recall: the default constructor is the constructor with no parameters.) If all constructors of the superclass require parameters, then the compiler reports an error when defaulting to the default constructor.

More constructor rules:

- Constructors are never inherited by a subclass
- If there are no constructors at all in a subclass, the default constructor in the superclass is automatically invoked. But like above, if all constructors of the superclass require parameters, then the compiler reports an error when trying to invoke the default constructor.
- If a constructor with parameters *is* provided in the subclass, and there is no default constructor, you must use that constructor when instantiating your object(s) otherwise you will get a compile-time error. The superclass default constructor is not automatically invoked in this case.