Computer Science 2

Lecture 3

Inheritance

Overview

- Principles of Object-Oriented Programming
- Inheritance:
  - Methods
  - Fields
- Class Converting
- Abstract Classes
- Access Control Modifiers
- Class Object

Principles of Object-Oriented Programming

- Principles discussed in the previous lectures:
  - Information Hiding and Encapsulation
  - Polymorphism
- In this lecture we introduce:
  - Inheritance

Why OOP?

- To try to deal with the complexity of programs
- To apply principles of abstraction to simplify the tasks of writing, testing, maintaining and understanding complex programs
- To increase code reuse
  - to reuse classes developed for one application in other applications instead of writing new programs from scratch ("Why reinvent the wheel?")
- Inheritance is a major technique for realizing these objectives

Inheritance

- **Inheritance is a OOP principle. It allows us to extend existing classes by adding methods and fields.**
- The more general class is called a *super class*;
- The more specialized class that inherits from the superclass is called *subclass*;

```
class SavingsAccount extends BankAccount
{
    new methods
    new instance fields
}
```

Inheritance

- A typical use of inheritance is the implementation of specialized subtypes of things
  - A subtype has all the properties and abilities of the more general type, plus a few more
- Remember the BankAccount example
- A BankAccount had a property
  - Balance
- And a BankAccount had some abilities
  - Create a new one, add money, withdraw money, check the balance
- This is fine, but there are special types of bank accounts that can do more

Inheritance

- Consider a savings account
- It can do everything a bank account can do, plus
- It has a new property
  - Interest rate
- It has some new functionality
  - It earns interest every month
- And it does some of the same things differently
  - You can start a bank account with no information, but a savings account needs to know what the interest rate is

Inheritance

- In English one might say a savings account is a special type of bank account
- Or that savings account is a subclass of ban account
- Or that a savings account is a bank account
  - Called the “is-a” relationship
- Note that this is not symmetric
  - All savings accounts are bank accounts
  - Not all bank accounts are savings accounts
- Let us look at how Java says this

Example

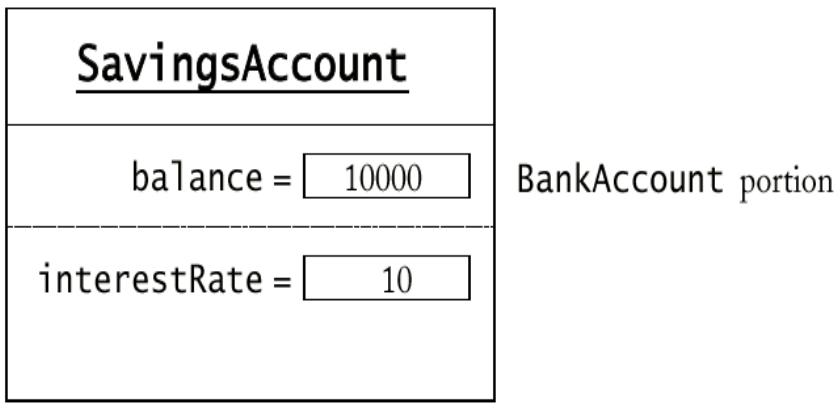
```
public class BankAccount
{ private double balance;
 public BankAccount()
 {  this.balance = 0;}
 public void deposit(double amount)
 {   balance = balance + amount; }
 public void withdraw(double amount)
 {   balance = balance - amount; }
 public double getBalance()
 {  return balance;
 }
}
```

- When defining a subclass you specify added instance fields, added methods, and changed and overridden methods.

- One advantage of inheritance is code reuse.

```
public class SavingsAccount extends
BankAccount
{ private double interestRate;

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



Example

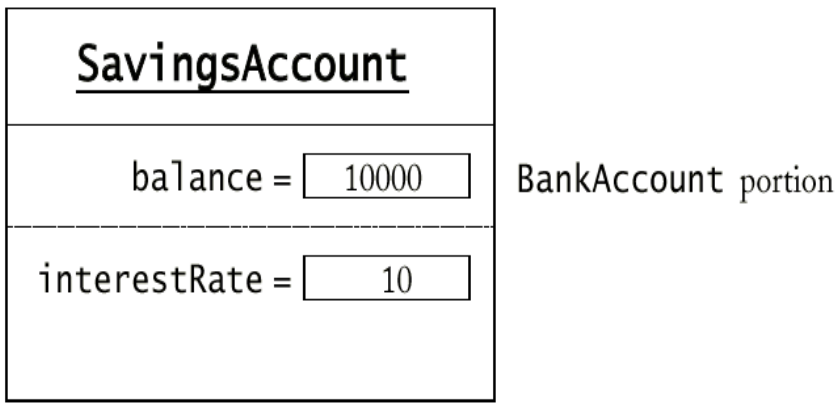
```
public class SavingsAccount extends
BankAccount
{  public SavingsAccount(double rate)
   { interestRate = rate;}

   public void addInterest()
   {  double interest = getBalance()
         * interestRate / 100;
      deposit(interest);
   }
   private double interestRate;
}
```

`extends` keyword says we're creating a subclass (in this example a subclass of the `BankAccount` class)

The constructor of `SavingsAccount` calls implicitly the *default* constructor of `bankAccount`.

`addInterest()` is a new method. It employs inherited `getBalance()` and `deposit()`



implements vs extends

- When a class *implements an interface*
- There are no variables involved
  - All fields in an interface are final (constant)
- The class promises to implement all the methods described in the interface
  - The interface never provides bodies, so the class must provide the bodies itself
- The class can define any other fields and methods it would like
- A class can implement as many interfaces as it wants

implements vs extends

- When a class extends a class
- The subclass gets all the non-private fields defined in the superclass
  - Just as if the subclass had defined them itself
- The subclass gets all the non-private methods defined in the superclass
  - Just as if the subclass had defined them itself
- The subclass can define any other fields and methods it would like
- A class can extend only one superclass
  - Known as *single inheritance*

implements vs extends

- These rules seem simple, but can lead to some complex behavior
- We will look at some of this in the rest of this lecture
- The most important thing to remember right now is:

You can use a class that implements an interface anywhere you need an object of that interface

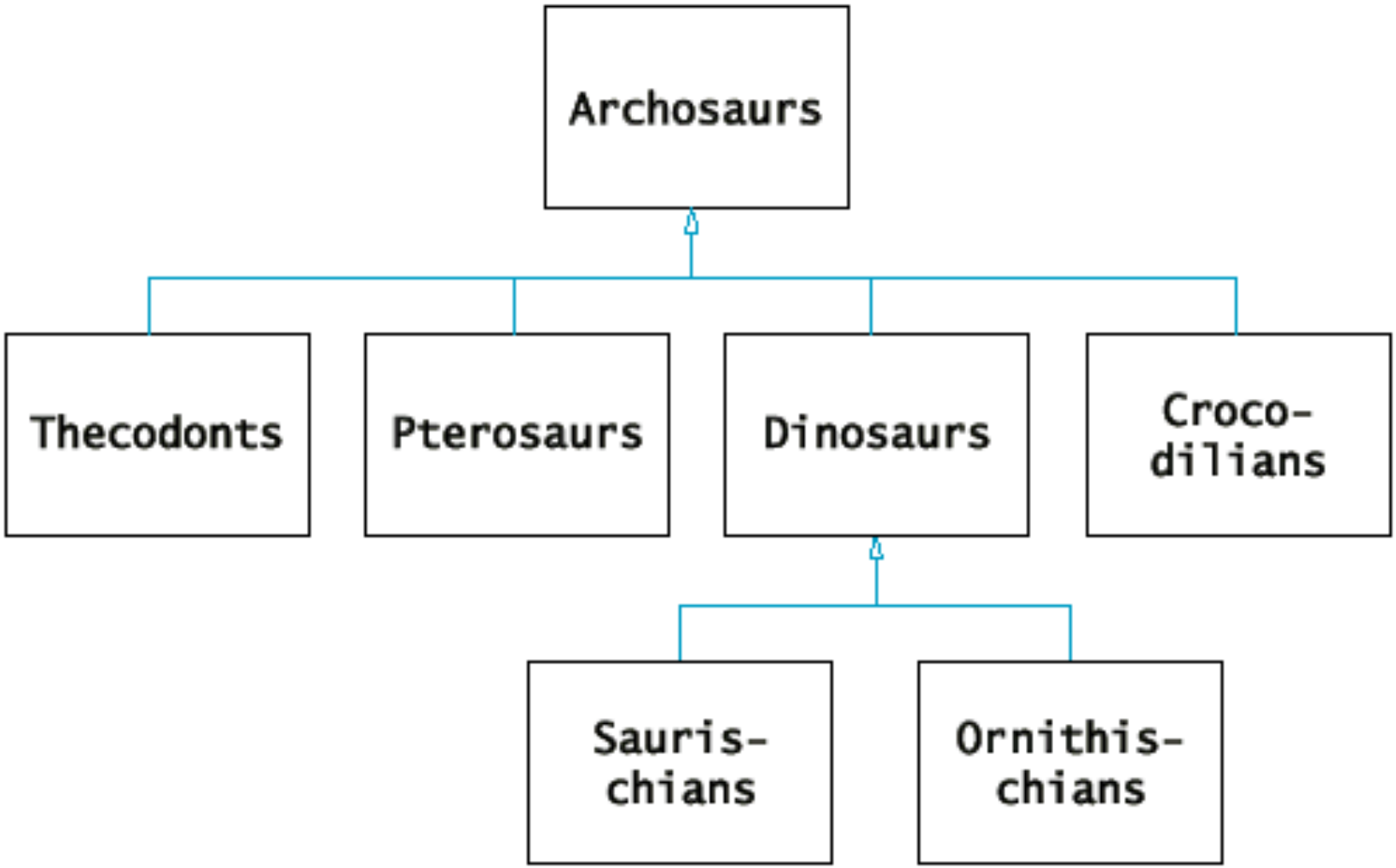
And

You can use a class that extends a superclass anywhere you need an object of that superclass

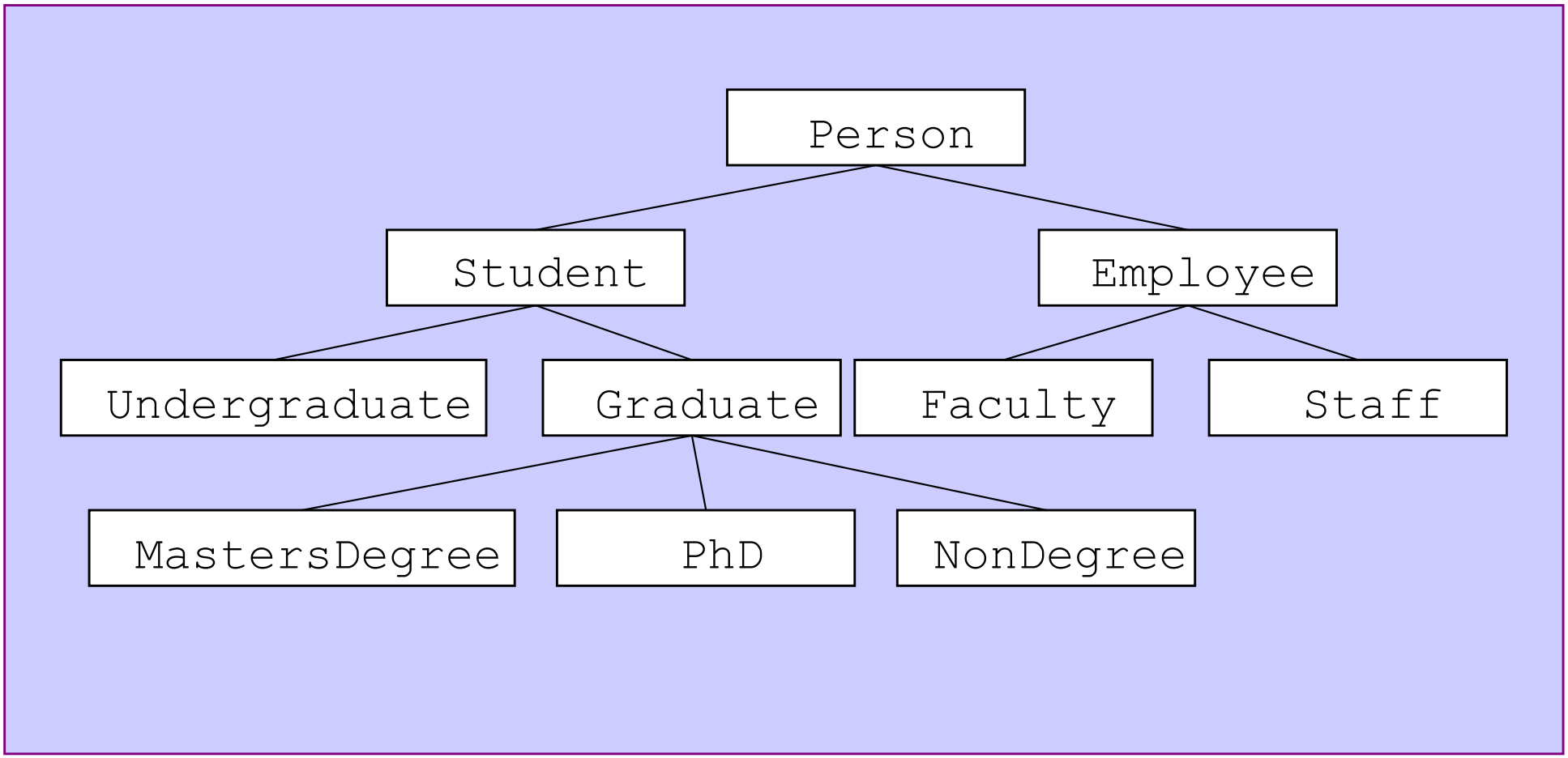
Derived Classes: a Class Hierarchy

- Superclasses tend to be more general in terms of the sort of thing they describe
  - Car, bank account, dinosaur
- Subclasses tend to be more specific
  - Porsche, savings account, Tyrannosaurus Rex
- This leads to the idea of a class hierarchy
  - More general classes on top, more specific classes below
  - It looks like a tree
- Examples follow

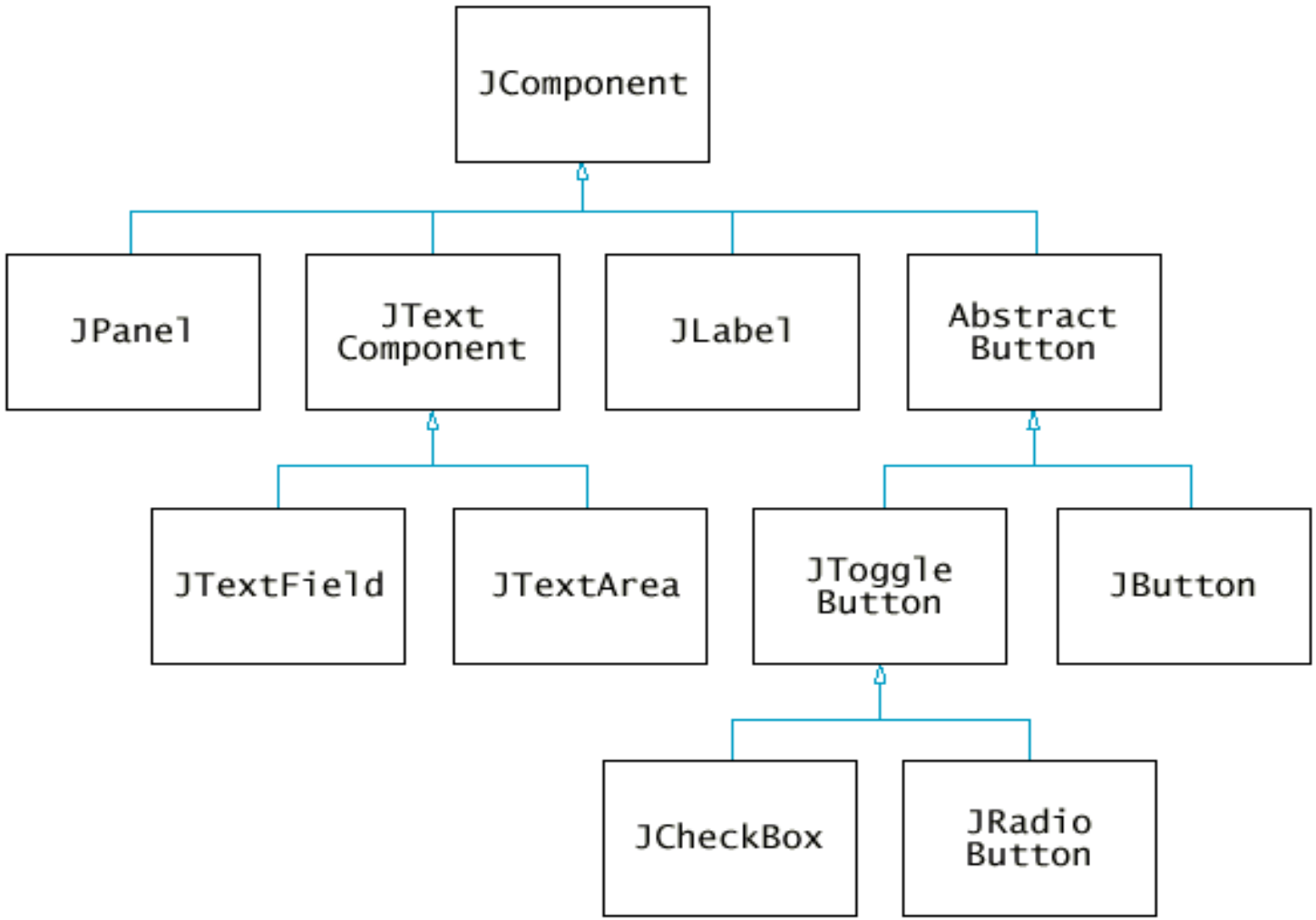
Derived Classes: a Class Hierarchy (cont.)



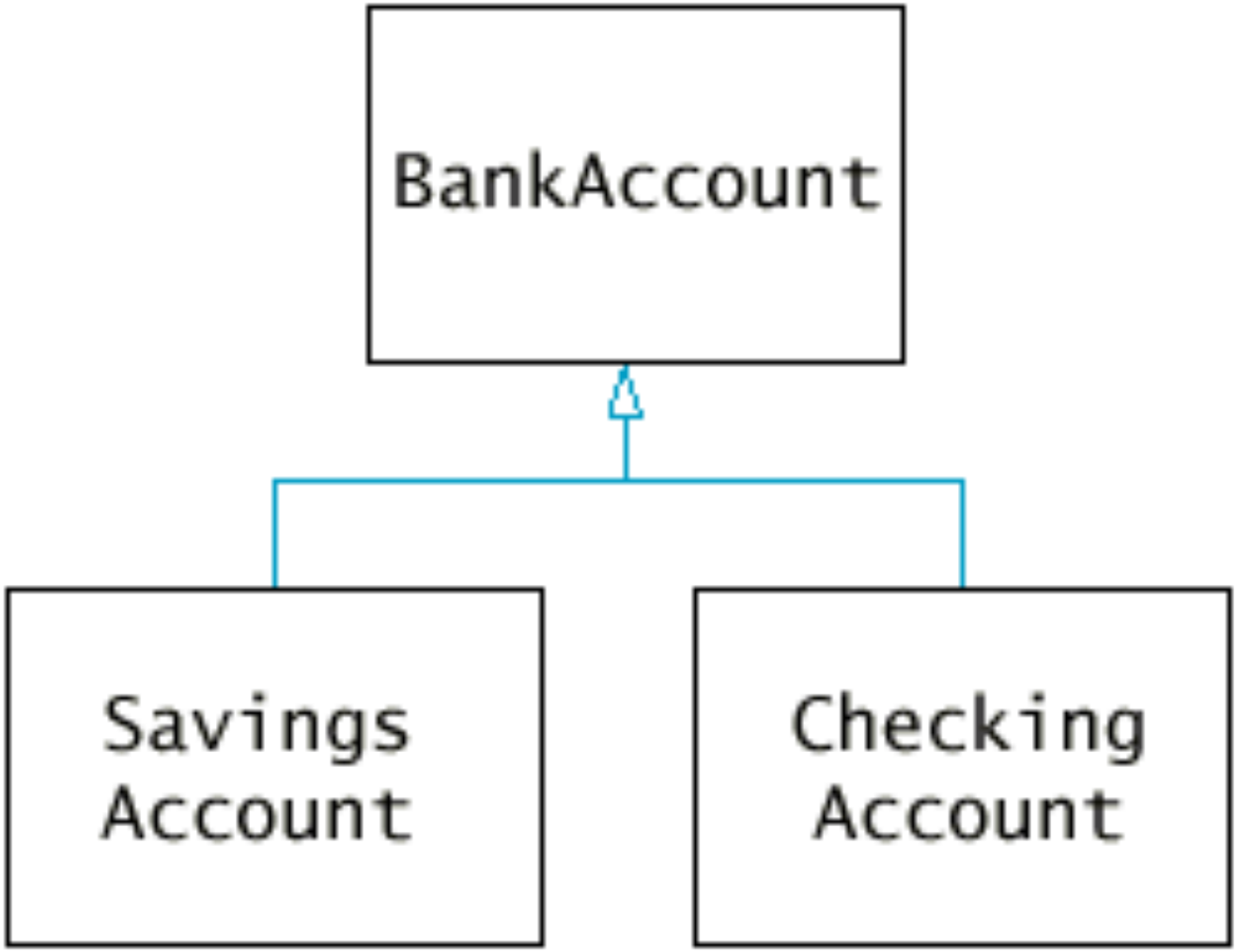
Derived Classes: a Class Hierarchy (cont.)



Derived Classes: a Class Hierarchy (cont.)



Derived Classes: a Class Hierarchy (cont.)



Derived Classes: a Class Hierarchy (cont.)

- You might note that all these examples get smaller the further up you go
  - This is how trees work
- In Java, every class has a superclass
  - Even if you do not declare one explicitly
- At the very top of the hierarchy is a class called Object
  - Technically, Object does not have a superclass
- Yes, every class inherits from Object
- This gives Java a *singly-rooted class hierarchy*
  - More on this later

Detour: Method Signature

- Every method in Java has a *signature*
- A method's signature consists of

1. The method's name
2. The number of parameters the method takes
3. The type(s) of those parameters (order matters)

- The signature does not include
  - The return type
  - Any exception it throws (more later)

Detour: Access Control Modifiers

- You have been using access control modifiers for a while now
  - public, private, etc.
- Now we can explain some more about what is going on with them
- An access control modifier (ACM) determines, for an element, what parts of a program that element is visible to
  - In other words, what parts of a program can "see" that thing
- For example, a private field cannot be used outside of the object it is declared in
  - Because it cannot be seen anywhere but inside that object/class
- Following is a summary table

Detour: Access Control Modifiers

	same class	same package	subclass	universal
`private`	X			
`public`	X	X	X	X
`protected`	X	X	X	
`default`	X	X		

Detour: Access Control Modifiers

- The default ACM is what happens when no ACM is specified
- You should never do this
  - Always specify an ACM
- We will never speak of the default ACM again
- When talking about inheritance, pretty much every rule applies only to public and protected elements
- It would get tedious continuously saying “non-private” everywhere
- So just accept that the following slides apply only to public and protected elements
  - Private elements will be covered separately, later

Detour: Constructors

- Constructors have their own rules for inheritance
- Thy will be covered separately from the rules for methods

Inheritance and Methods

When writing a subclass:

- *You can inherit methods of the superclass*
- *You can override methods of the superclass*
- *You can write new methods*

We consider these three cases in the next slides

– Constructors are a little different and will be covered separately later

Inheritance and Methods

A subclass inherits all methods from its superclass, if the subclass does nothing

The class `SavingsAccount` inherits the methods `getBalance` and deposit from `BankAccount`.

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

Inheritance and Methods

- ***A method of a subclass overrides a method of a superclass if both methods have the same signature.***
- Assume that we have a class `CheckingAccount` with its own method `deposit`. `deposit` overrides the method `deposit` from `BankAccount`.

```
public class CheckingAccount extends BankAccount
{   ……….
   public void deposit(double amount)
   {     transctCount++;
      super.deposit(amount);
   }
   private int transctCount ;
   ……….
}
```

Inheritance and Methods

- *A subclass can have new methods of which the names or signatures differ those from the superclass. These methods can be applied only to the objects of the subclass*.
- The class **`SavingsAccount`** has a new method **`addInterest`**.

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

Inheritance and Methods

- As a subclass, why do these things?
- Inherit
  - Because the superclass does something you want done
  - You get this functionality for free (code reuse)
- Override
  - Because the superclass kind of does the right sort of thing, but it is not exactly what you want
- New method
  - Because the subclass has functionality that the superclass does not

Inheritance and Fields

When writing a subclass of superclass:

- ***You inherit fields from the superclass***
- ***You can define new fields***

We consider these two cases in the next slides.

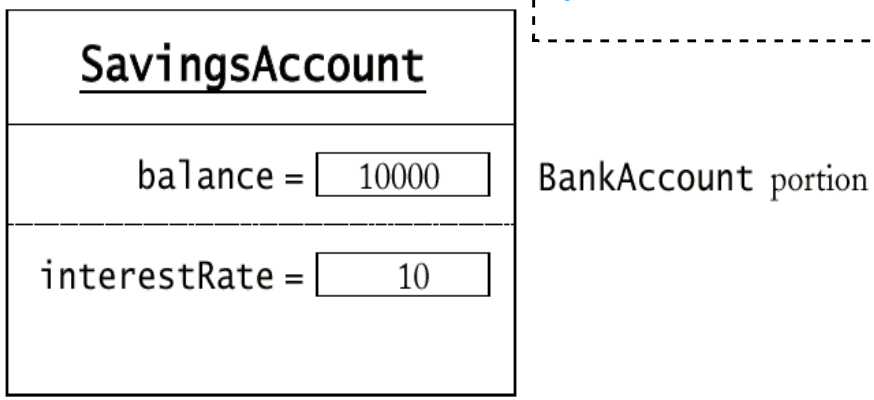
Inheritance and Fields

- BankAccount has no fields that are either public or protected
- Therefore, SavingsAccount inherits no fields
- However, BankAccount does have a private field, balance
- Since this field is private, it follows the rules for private elements described later

Inheritance and Fields

- *You can define new fields.* In the class **`SavingsAccount`** the field **`interestRate`** is new.

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



Private elements

- The rules for inheriting private elements (methods and fields) are a little different than the rules for inheriting public and protected elements
  - Not hard, just different

Inheritance and Private Methods

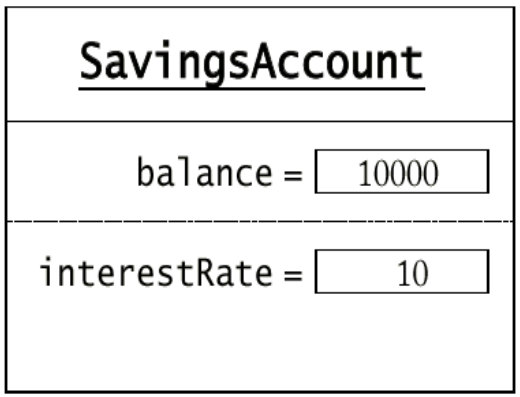
- A subclass does not inherit private methods from its superclass
  - The subclass cannot call them because they are not visible to the subclass
- The subclass can declare a method with the same signature as a private method in the superclass
- However, this is not overriding
- Do not do this
  - It is confusing

Inheritance and Private Fields

- When a superclass has private fields, the subclass cannot see them
  - The subclass cannot read or write them
- However, the subclass still "has" those fields in a very real sense
  - For example, a SavingsAccount still has a balance, even though the balance field is private in BankAccount
- The subclass has to use the setters and getters in the superclass to access the field
- An example follows

Inheritance and Private Fields

- The field **balance** is inherited in **SavingsAccount**;
- But, since balance is **private** in **BankAccount**, **SavingsAccount** uses the method `getBalance` inherited from **BankAccount**.



BankAccount portion

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

Inheritance and Private Fields

- A subclass can declare a field with the same name and type as a private field in the superclass
- This is called *variable hiding*
- Do not do this
  - It is confusing

Constructors in a Subclass

- Constructors have some special rules that come into play when extending classes
- First off, constructors are not inherited
  - SavingsAccount does not inherit the BankAccount constructors
- Second, if you do not declare a constructor, one will be automagically created for you
- It is called the *default constructor*
  - It takes no arguments
  - It does nothing but call the superclass's default constructor
- If you do declare a constructor, no default constructor will be created

Constructors in a Subclass

- A class can have a single constructor called the *no-argument constructor*
  - They did not spend a lot of time on naming this
- It takes no parameters
  - It can do anything it wants in its body, including nothing
- The first thing any constructor does is call the no-argument constructor of its superclass
  - If there is no no-argument constructor in the superclass, there will be an error
- Exception: you can call another constructor from the superclass explicitly

Constructors in a Subclass

- The subclass constructor can call a superclass constructor using the keyword **super**. The call has to be in the first line.

```
public class CheckingAccount extends BankAccount
{  public CheckingAccount(int initialBalance)
   {  super(initialBalance);
    transctCount = 0;
   }
   …….
   private int transctCount ;
}
```

Constructors in a Subclass

- To call a constructor within a class use the keyword `this`.
- `this(0)` below calls constructor `CheckingAccount(int initialBalance)` with `initialBalance` equal to `0`.

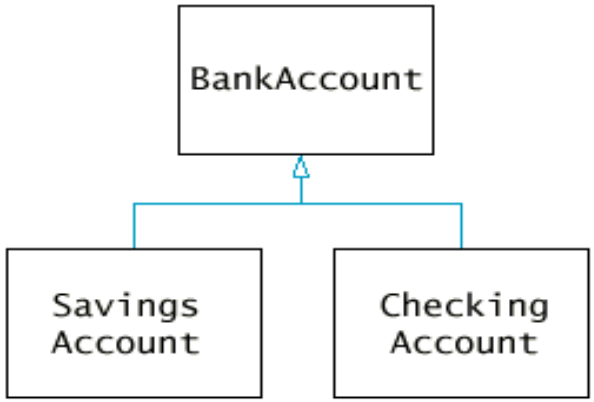
```
public class CheckingAccount extends BankAccount
{  public CheckingAccount(int initialBalance)
   {  super(initialBalance);
    transctCount = 0;
   }
   public CheckingAccount()
   {  this(0);}
   ……..
   private int transctCount ;
}
```

Call to an Overridden Method

- Use the keyword **super** to call a method from the superclass that was overridden in the subclass
- **deposit** from **CheckingAccount** overrides **deposit** from **BankAccount**. **deposit** from **BankAccount** is called in **deposit** from **CheckingAccount** using `super`.
- the keyword **super** can be used for calling **superclass** non-overriden methods as well.

```
public class CheckingAccount extends BankAccount
{  public void deposit(double amount)
   {     transctCount++;
      super.deposit(amount);
   }
   public void deductFees()
   {  if (transctCount > FREE_TRANSCT)
      {  double fees = FEE *(transctCount - FREE_TRANSCT);
         super.withdraw(fees);
      }
    transctCount = 0;
   }……
}
```

```
public class CheckingAccount extends BankAccount
{  public CheckingAccount(int initialBalance)
   {  super(initialBalance);
    transctCount = 0;
   }
   public void deposit(double amount)
   {    transctCount++;
      super.deposit(amount);
   }
   public void withdraw(double amount)
   {    transctCount ++;
      super.withdraw(amount);
   }
   public void deductFees()
   {  if (transctCount > FREE_TRANSCT)
      {  double fees = FEE *(transctCount - FREE_TRANSCT);
         super.withdraw(fees);
      }
    transctCount = 0;
   }
   private int transctCount ;
   private static final int FREE_TRANSCT = 3;
   private static final double FEE = 2.0;
}
```



Converting from Subclasses to Superclasses

Subclass references can be converted to superclass references:

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

Note that superclass references don't know the full story:

```
anAccount.addInterest(); // ERROR
```

Why would anyone want to know *less* about an object? This can happen when you reuse code! See next slide!

Polymorphism

Method that wants a BankAccount object:

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

Works with BankAccount objects, and objects that are of any class that extends BankAccount:

```
BankAccount momsAcc = new BankAccount(100);
CheckingAccount sonAcc = new CheckingAccount(100);
momsAcc.transfer(100, sonAcc);
sonAcc.transfer(100, momsAcc);
```

Polymorphism

- This is standard polymorphism
  - CheckingAccount extends BankAccount, so a CheckingAccount object can be used wherever a BankAccount object could be used
- Why not just declare that parameter as `Object`?
  - After all, CheckingAccount also inherits from Object
  - Every class inherits from Object
- Because the `Object` class doesn't have the `deposit` method!

The `final` Modifier

- ***In the case of methods:*** `final` *specifies that a method definition* <u>*cannot*</u> *be overridden with a new definition in a subclass:*

```
public final void specialMethod() {
```

- ***In the case of classes:*** `final` *specifies that the class cannot be used as a base class to derive another class:*

```
public final class String{ ….}
```

- Allows the compiler to generate more efficient code

Abstract Classes

- An abstract class is a placeholder in a class hierarchy that represents a generic concept;
- An abstract class cannot be instantiated;
- We use the modifier `abstract` on the class header to declare a class as abstract;
- An abstract class often contains abstract methods (like an interface does), though it doesn't have to;

```
public abstract class BankAccount
{ public abstract void deductFees();
  ……….
}
```

Abstract Classes

- The subclass of an abstract class must override the abstract methods of the parent, or it too will be considered abstract;
- An abstract method cannot be defined as `final` (because it must be overridden) or `static` (because it has no definition yet)
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate
- Usually ***concrete*** classes extend abstract ones, but the opposite is also possible.

References and abstract classes

- Suppose the following two classes were defined:

```
public abstract class Figure
public class Rectangle extends Figure
```

- Are these instantiations correct?

```
Rectangle r = new Rectangle(...);//correct
Figure f    = new Rectangle(...);//correct
Figure f    = new Figure(...);   //error
```

`Object`: The Cosmic Superclass

- `Object` is the superclass of all classes
- Every class extends `Object`
- Methods defined for `Object` are inherited by all classes
- Which means your class inherits these methods whether you want them or not
- Some of these methods are worth looking at because
  - You might want to use them
  - You might want to override them
- We will look at two today

Object: The Cosmic Superclass

Partial Method Summary

- String **toString()**
  - Returns a string representation of the object
- **boolean equals(Object obj)**
  - Indicates whether some other object is "equal to" this one

toString and Inheritance

- `public String toString()`
  - Returns a string representation of the object.
  - Supply `toString()` in all classes.
  - **System.out.println()** invokes `toString()` whenever it has to print an object.
- The default toString method is not great
  - You will want to override it

toString and Inheritance

```
public class BankAccount
{
 public String toString()
 { return getClass().getName() +
        "[balance = " + balance + "]";
 }
```

```
SavingsAccount momsSavings = new SavingsAccount(0.5);
CheckingAccount harrysChecking = new
                              CheckingAccount(100);
System.out.println(harrysChecking);
System.out.println(momsSavings);
```

Print:

CheckingAccount [balance = 100.0]
SavingsAccount [balance = 0.5]

toString and Inheritance

```
public class SavingsAccount extends BankAccount
{...
   public String toString(){
   return super.toString() + "[interestRate = " +
      interestRate + "]";
   }
```

```
SavingsAccount momsSavings = new SavingsAccount(0.5);
CheckingAccount harrysChecking = new
                                 CheckingAccount(100);
System.out.println(harrysChecking);
System.out.println(momsSavings);
```

Prints:

```
CheckingAccount [balance = 100.0]
SavingsAccount [balance = 0.5][interestRate = 0.5]
```

toString and Inheritance

If NO `toString` method is provided :

```
SavingsAccount momsSavings = new SavingsAccount(0.5);
CheckingAccount harrysChecking = new CheckingAccount(100);
System.out.println(harrysChecking);
System.out.println(momsSavings);
```

results:

CheckingAccount@eee36c

SavingsAccount@194df86

inherits `Object's toString` method.

from the API: This method returns a string equal to the value of:

```
getClass().getName() + '@' +
Integer.toHexString(hashCode())
```

equals method

- You should define the equals method to test whether two objects have equal state.

```
public boolean equals (Object
  otherObject)
{
  // first statement will cast otherObject
  // code defining equal state goes here
}
```

equals method

```
public boolean equals (Object otherObject)
{
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass())
        return false;

  // cast statement goes here
  // code to compare equal state goes here

}
```

equals method

```
public class Coin{
...
public boolean equals (Object otherObject)
{
   if (otherObject == null) return false;
   if (getClass() != otherObject.getClass())
       return false;

   Coin other = (Coin) otherObject;
 return name.equals(other.name) &&
        value == other.value;
}
```

`equals` method and Inheritance

- If you do not define `equals` in your class, you inherit `Object's` equals method.
- From the API:
  - The equals method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

`equals` method and Inheritance

- When defining `equals` in a subclass, first call `equals` in the superclass.

```
public boolean equals (Object otherObject)
{
  if (! super.equals(otherObject)) return
  false;
  // cast goes here
    // code to compare states goes here.
}
```

`equals` method and Inheritance

- When defining equals in a subclass, first call `equals` in the superclass.

```
public CollectCoin extends Coin{
…
public boolean equals (Object otherObject)
{
   if (!super.equals(otherObject)) return false;
  CollectCoin other = (CollectCoin)
                                  otherObject;
  return year == other.year;
}
```

What we have learned

- Principles of Object-Oriented Programming
- Inheritance:
  - Methods
  - Fields
- Class Converting
- Abstract Classes
- Access Control Modifiers
- Class Object