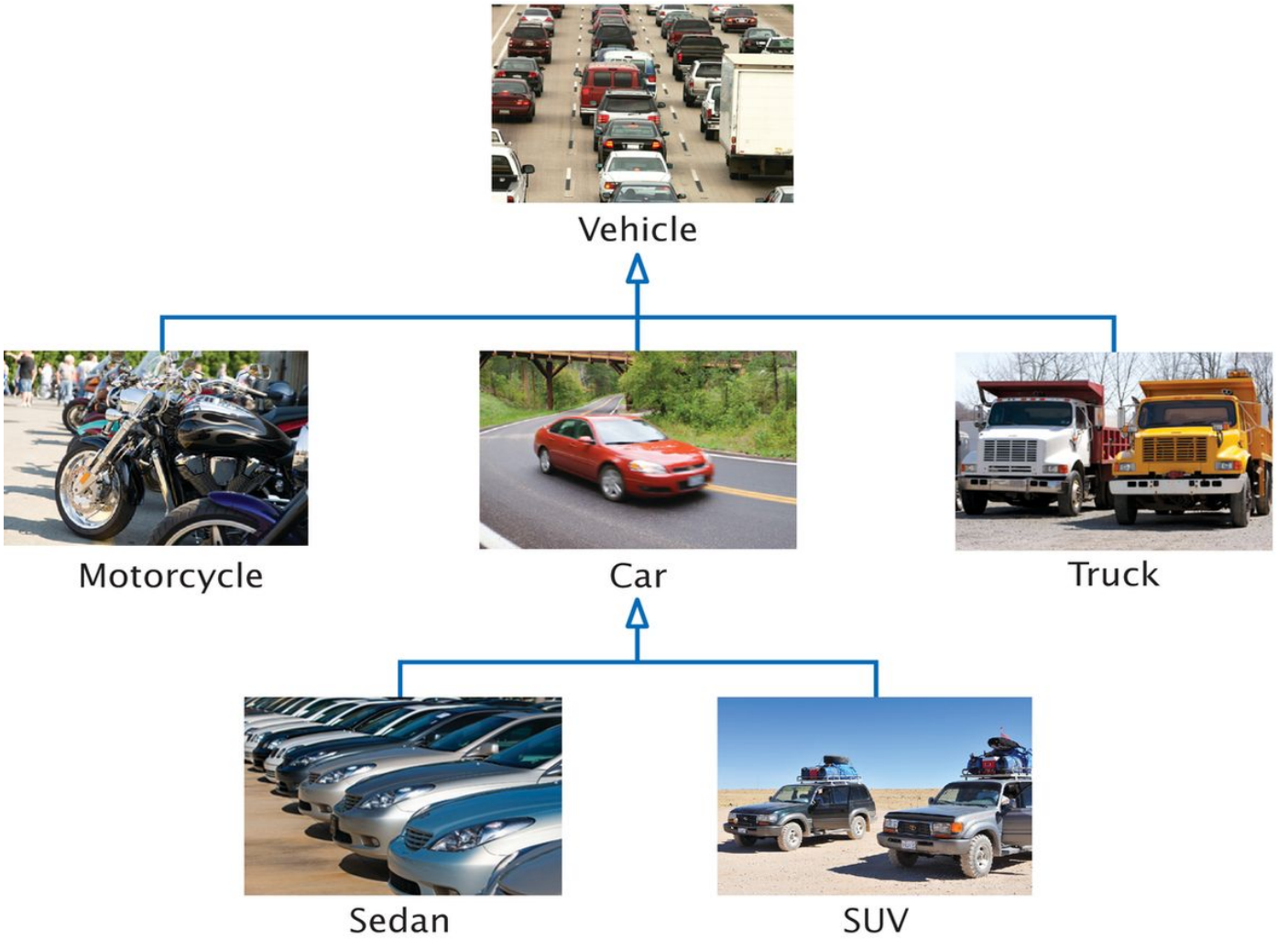
Objective

Inheritance

- To learn about inheritance
- To understand how to inherit and override super-class methods
- To be able to invoke super-class constructors
- To learn about protected and package access control

This lecture covers chapter 9 of your text book.

Inheritance Hierarchies



An Introduction to Inheritance

- Inheritance: extend classes by adding methods and fields
- Example: Suv is a Car with added more features.

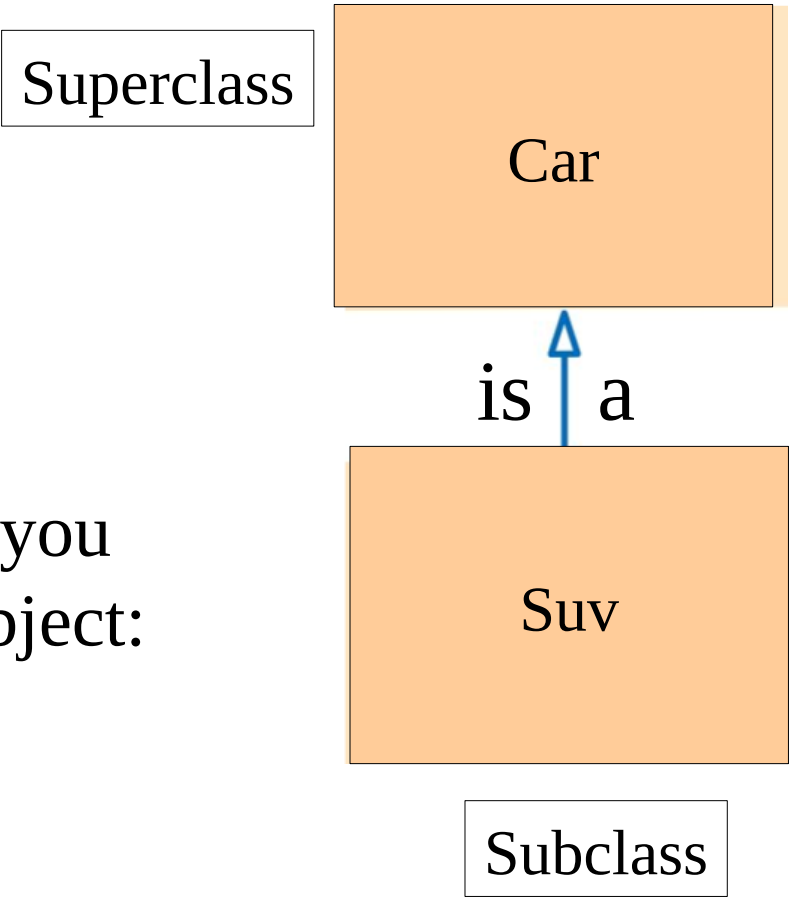
```
class Suv extends Car
{
   new methods
   new instance fields
}
```

- Suv automatically inherits all fields and methods of class Car

Continued…

Substitution Principle

you can always use a subclass object when a super class is expected.



Consider a method that takes an argument of type Vehicle:

```
void processCar (Car c)
```

Because Suv is a subclass of Car, you can call that method with a Suv object:

```
Suv mySuv = new Suv(...)
processCar (mySuv);
```

We cal also declare:

```
Car myCar = new Suv();
```

An Introduction to Inheritance

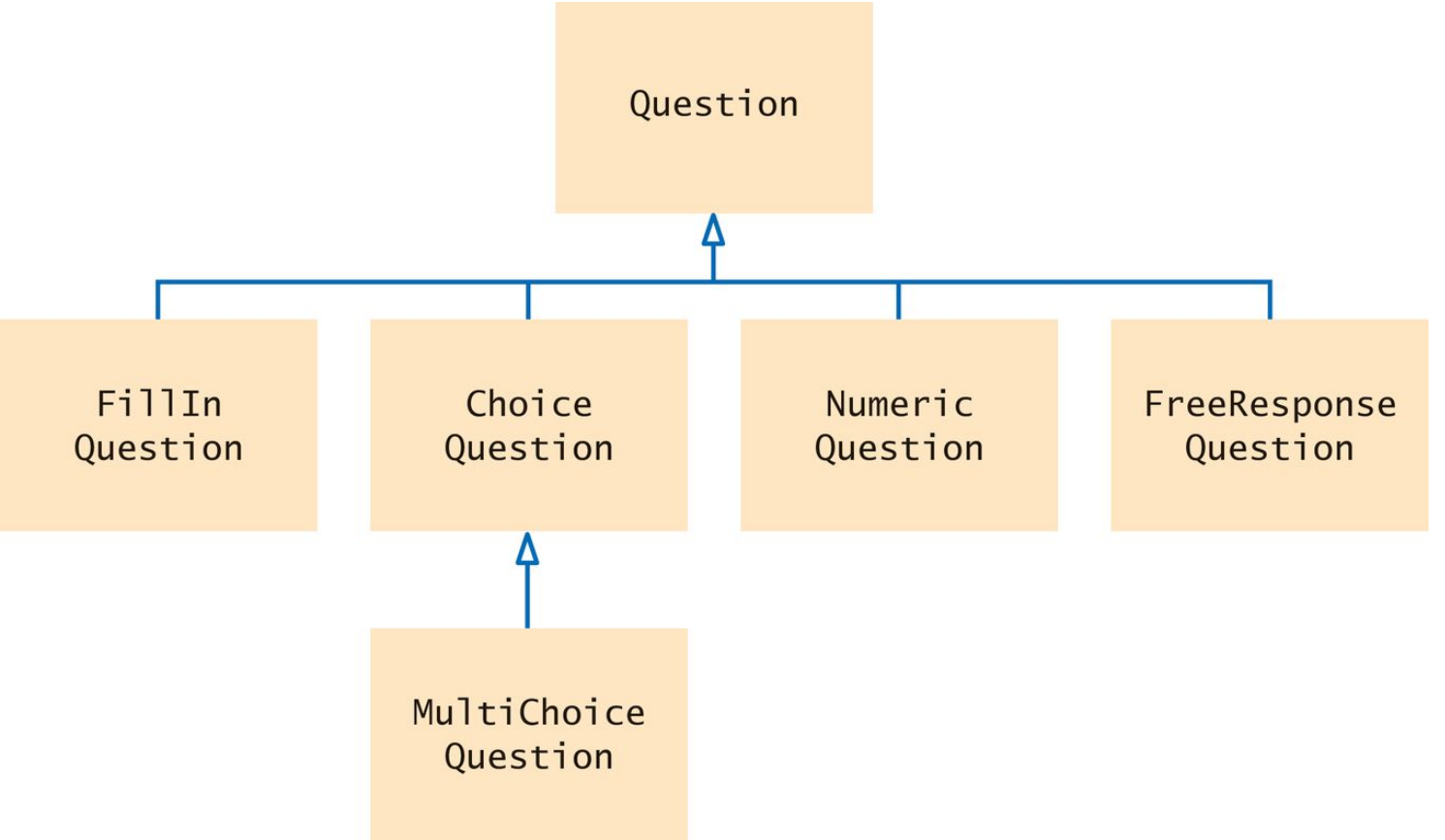
Extended class = *superclass* (`Car`),
extending class = *subclass* (Suv)

```
Suv mySuv = new Suv();
mySuv.set4wheelDrive();// Just for Suv
```

subclass inherits behavior and state

One advantage of inheritance is reuse code instead of duplicating it.

Example1



At the root of this hierarchy is class Question

Example1 (continued)

```
public class Question{
   private String text;
   private String answer;
   public Question(){
      text = "";
      answer = "";
   }
   public void setText(String questionText){
      text = questionText;
   }
   public void setAnswer(String correctResponse){
      answer = correctResponse;
   }
   public boolean checkAnswer(String response){
      return response.equalsIgnoreCase(answer);
   }
   public void display() {
      System.out.println(text);
   }
```

Example1 (continued)

```
import java.util.Scanner;
public class QuestionDemo1
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      Question q = new Question();
      q.setText("Who was the inventor of Java?");
      q.setAnswer("James Gosling");

      q.display();
      System.out.print("Your answer: ");
      String response = in.nextLine();
      System.out.println(q.checkAnswer(response));
   }
}
```

Check example1

Implementing Subclass

Suppose you want to write a class ChoiceQuetion that handles questions such as the following:

In which country was the inventor of Java born?

1. Australia
2. Canada
3. Denmark
4. Unites States

Option1: Write a ChoiceQuestion class from scratch.

Option2: Use inheritance and implement ChoiceQuestion as a subclass of of the Question class.

Implementing Subclass

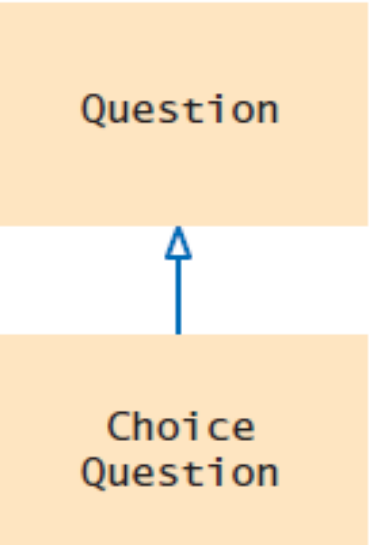
1. Subclass automatically have all instance variables of the super class. However, the private instance fields/methods of the super class are not directly accessible.

2. Subclass inherits all public methods from the superclass.

3. Add any new instance fields and methods to the subclass.

4. Override: change the implementation of of inherited methods that are not appropriate

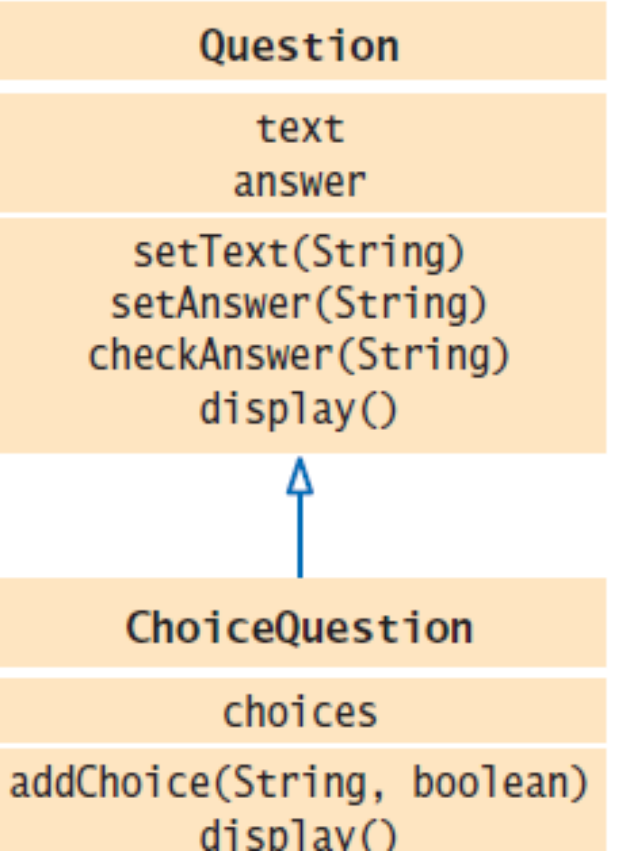
```
public class ChoiceQuestion
extends Question {
   ...
}
```



Example2: ChoiceQuestion

ChoiceQuestion class inherits from the Question class, it needs to spell out following differences:

```
public class ChoiceQuestion extends Question {
   private ArrayList<String> choices;
   public ChoiceQuestion()   }
   public void addChoice(...) {}
   public void display() {}
}
```



Example2(continued)

Subclass declaration:

Syntax

```
public class SubclassName extends SuperclassName
{
    instance variables
    methods
}
```

The reserved word extends denotes inheritance.

Subclass

Superclass

Declare instance variables that are added to the subclass.

Declare methods that are added to the subclass.

Declare methods that the subclass overrides.

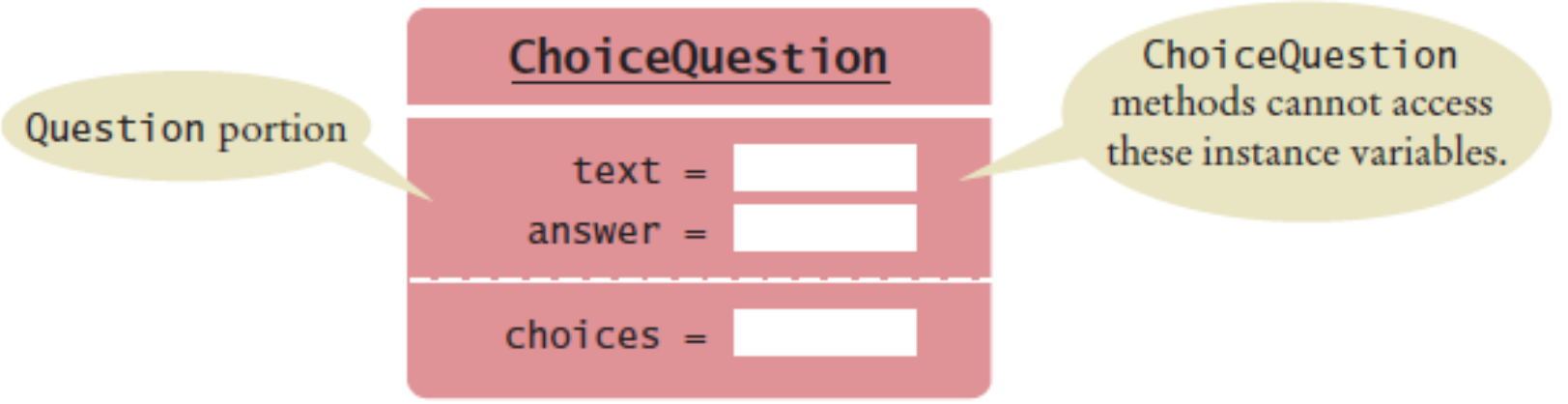
```
public class ChoiceQuestion extends Question
{
    private ArrayList<String> choices;

    public void addChoice(String choice, boolean correct) { . . . }

    public void display() { . . . }
}
```

An Inheritance Diagram

You can call the inherited methods of a subclass objects:

choiceQuestion.setAnswer("2");

since the private variables of the superclass are not accessible.

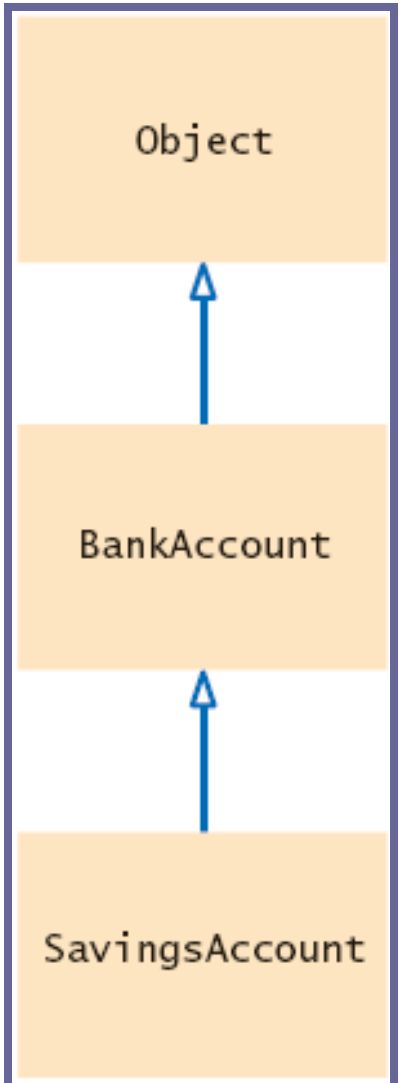


Check example2

An Inheritance Diagram

- Every class extends the Object class either directly or indirectly

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



BackAccount.java

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

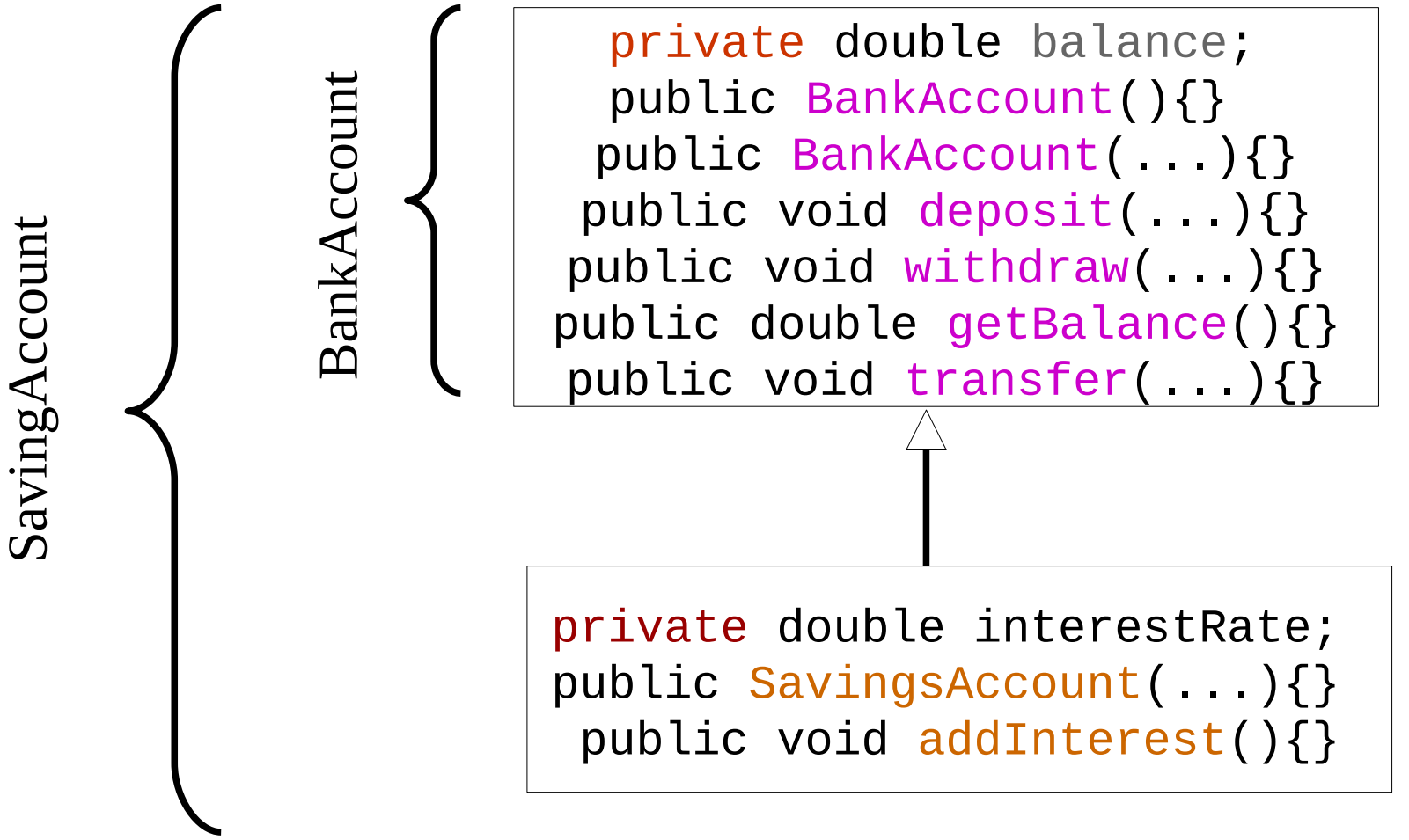
Implementing Subclass

```
public class SavingsAccount extends BankAccount  {
   private double interestRate;          // New Instance

   public SavingsAccount(double rate) {

     interestRate = rate;
   }
   public void addInterest() {            // New Method

     double interest = getBalance() * interestRate / 100;
     deposit(interest);
   }
}
```

Check example3

In subclass, specify added instance fields, added methods, and overridden methods

Implementing Subclass



SavingAccount does <u>not</u> have access to private fields of BankAccount.

An Introduction to Inheritance

Why we are not accessing balance, and calling getBalance() instead ?

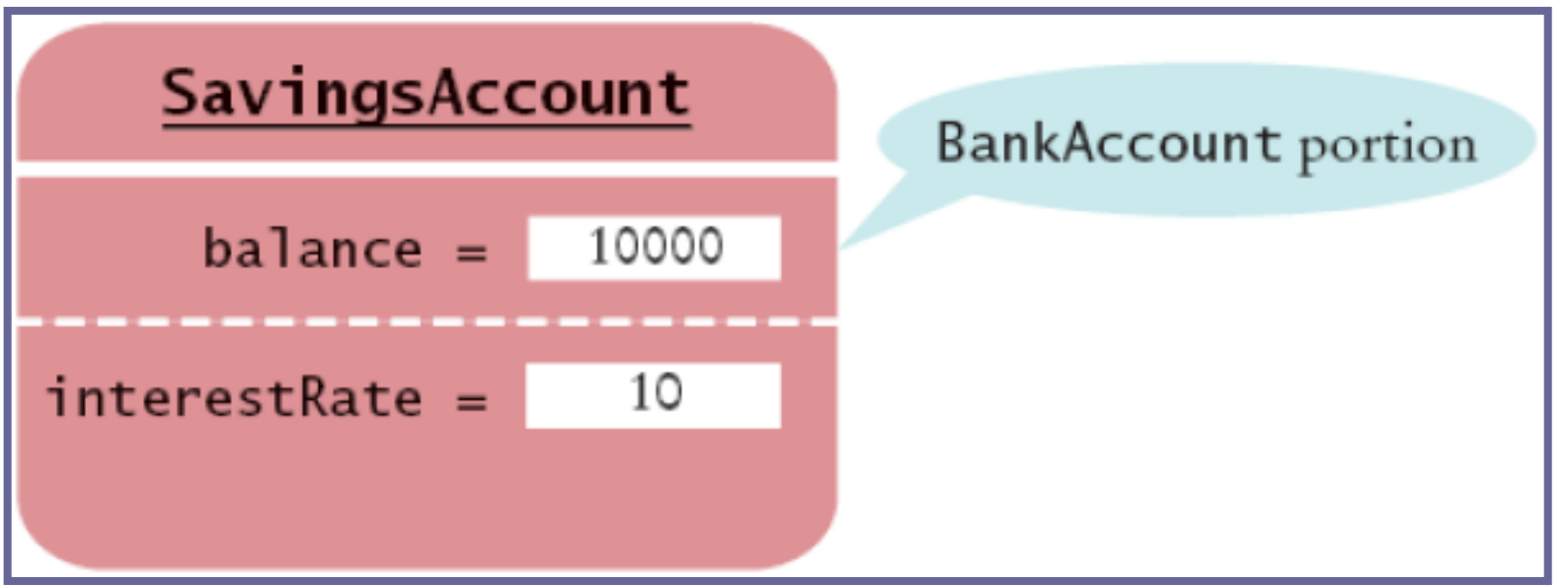
Encapsulation: `addInterest` calls `getBalance` rather than updating the `balance` field of the superclass (balance is `private`)

Note that `addInterest` calls `getBalance` without specifying an implicit parameter (the calls apply to the same object)

Layout of a Subclass Object

`SavingsAccount` object inherits the `balance` instance field from `BankAccount`,

and gains one additional instance field: `interestRate`



Syntax: Inheritance

```
class SubclassName extends SuperclassName
{
   added/override methods
   added instance fields
}
```

Example4

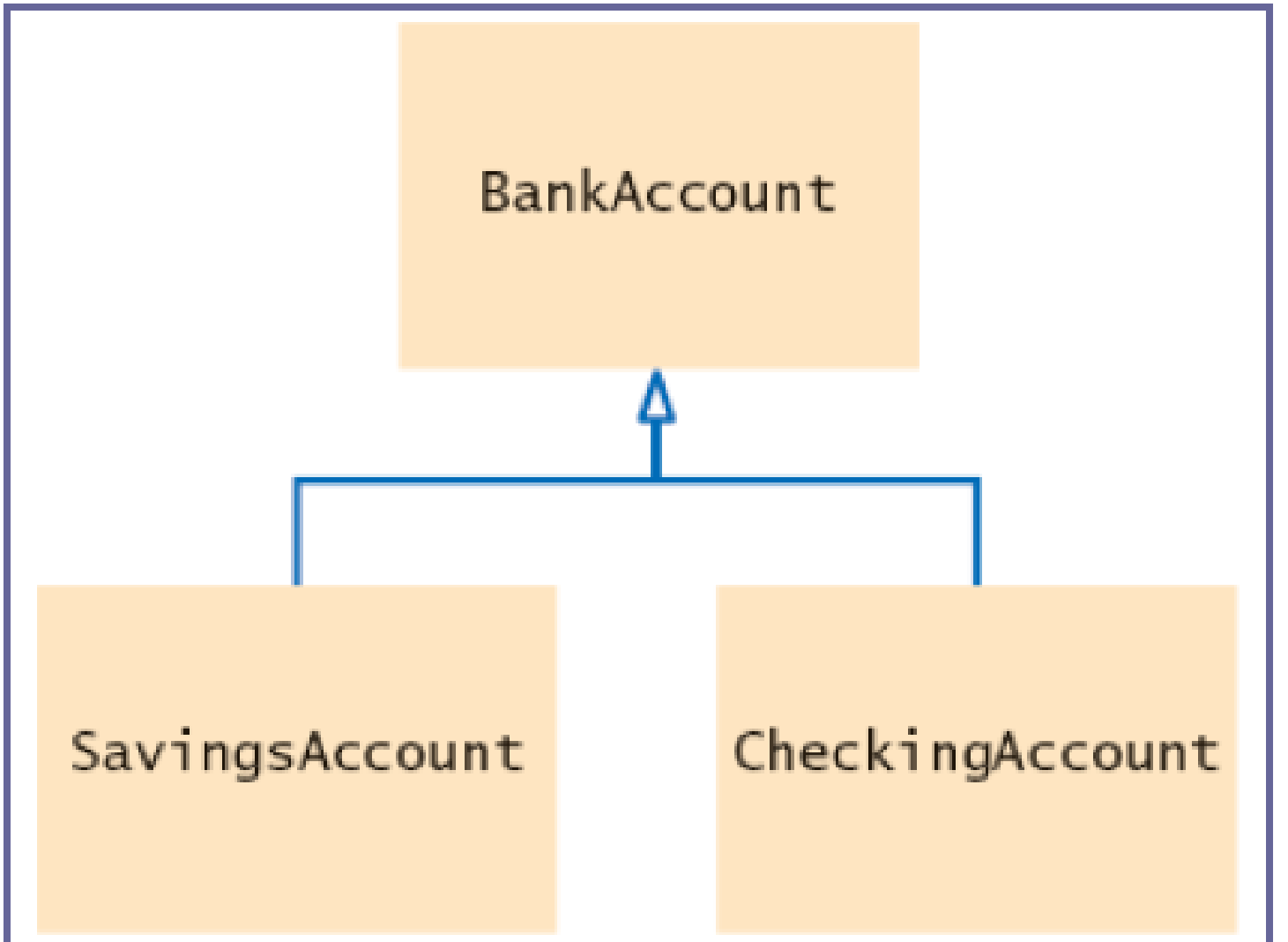
A Simpler Hierarchy: Hierarchy of Bank Accounts

Consider a bank that offers its customers the following account types:

- Checking account: no interest; small number of free transactions per month, additional transactions are charged a small fee
- Savings account: earns interest that compounds monthly (We have already developed it)

Example4

Inheritance hierarchy:



Example4

All bank accounts support the `getBalance, deposit`, and `withdraw methods`

`SavingAccount:`

- needs a method `addInterest`

CheckingAccount:

- support the `deposit` and `withdraw` methods, but the implementations differ
- needs a method `deductFees`

Inheriting Methods

- Add method:
  - Supply a new method that doesn't exist in the superclass
  - New method can be applied only to subclass objects

Like :

```
addInterest(){} in SavingAcoount class
deductFees(){} in CheckingAcoount class
```

Inheriting Methods

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

BankAccount

```
private double balance;
public BankAccount(){}
public BankAccount(...){}
public void deposit(...){}
public void withdraw(...){}
public double getBalance(){}
public void transfer(...){}
```

Check example4

```
private int FREE_TRANSACTIONS;
private double TRANSACTION_FEE;
public CheckingAccount(){}
public void deductFees(){}
public void deposit(...){}
public void withdraw(...){}
```

CheckingAccount

```
private double interestRate;
public SavingsAccount(...){}
public void addInterest(){}
```

SavingAccount

Implementing the `CheckingAccount` Class

```
public class CheckingAccount extends BankAccount {
  private int transactionCount;    // new instance field
  public void deductFees() {. . .} // new method
  public void deposit(double amount) {...}// Overrid
  public void withdraw(double amount){...}// Override

}
```

Each `CheckingAccount` object

has two instance fields:

`balance` : inherited from `BankAccount`

`transactionCount` : new to `checkingAccount`

has a new method: `deduceFee()`

overrides two methods: `deposit()` and `withdraw()`

Inherited Fields Are Private

Consider `deposit` method of `CheckingAccount`

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

Can't just add `amount` to `balance`. Why?

`balance` is a *private* field of the superclass

A subclass has no access to private fields of its superclass

Subclass must use public interface of its super class

Continued

Invoking a Super Class Method

What is the problem with the following implementation?

```
public void deposit(double amount){
   transactionCount++;
   // now add amount to balance
   deposit(amount);  // WRONG Why?
}
```

That is the same as `this.deposit(amount)`

Calls the same method (infinite recursion)

But we should call `deposit()` method of BankAccount instead.

Invoke superclass method `super.deposit(amount)`

Continued…

Invoking a Super Class Method

Correct Implementation:

```
public void deposit(double amount){
   transactionCount++;
   // now add amount to balance
   super.deposit(amount);  // OK
}
```

Invokes *superclass method* `super.deposit(amount)`

Continued…

Syntax: Calling a Superclass Method

```
super.methodName(parameters)
```

Example:

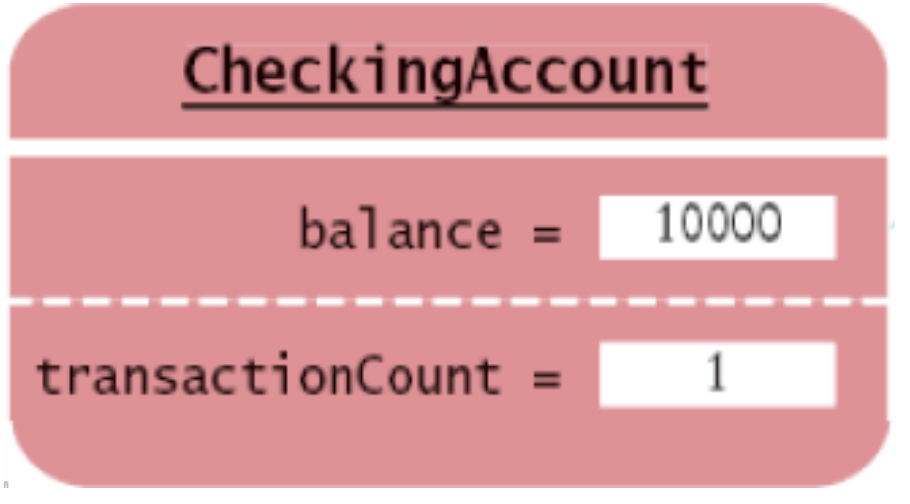
```
 public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Purpose:

To call a method of the superclass instead of the method of the current class

Implementing Remaining Methods

```
public class CheckingAccount extends BankAccount
{
   . . .
   public void withdraw(double amount)
   {
      transactionCount++;
      // Now subtract amount from balance
      super.withdraw(amount);
   }
```



Inheriting Instance Fields

- Can't override fields?
- Inherit field: All fields from the superclass are automatically inherited
- Add 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
  - Fields can hold different values
  - Legal but extremely undesirable

Common Error

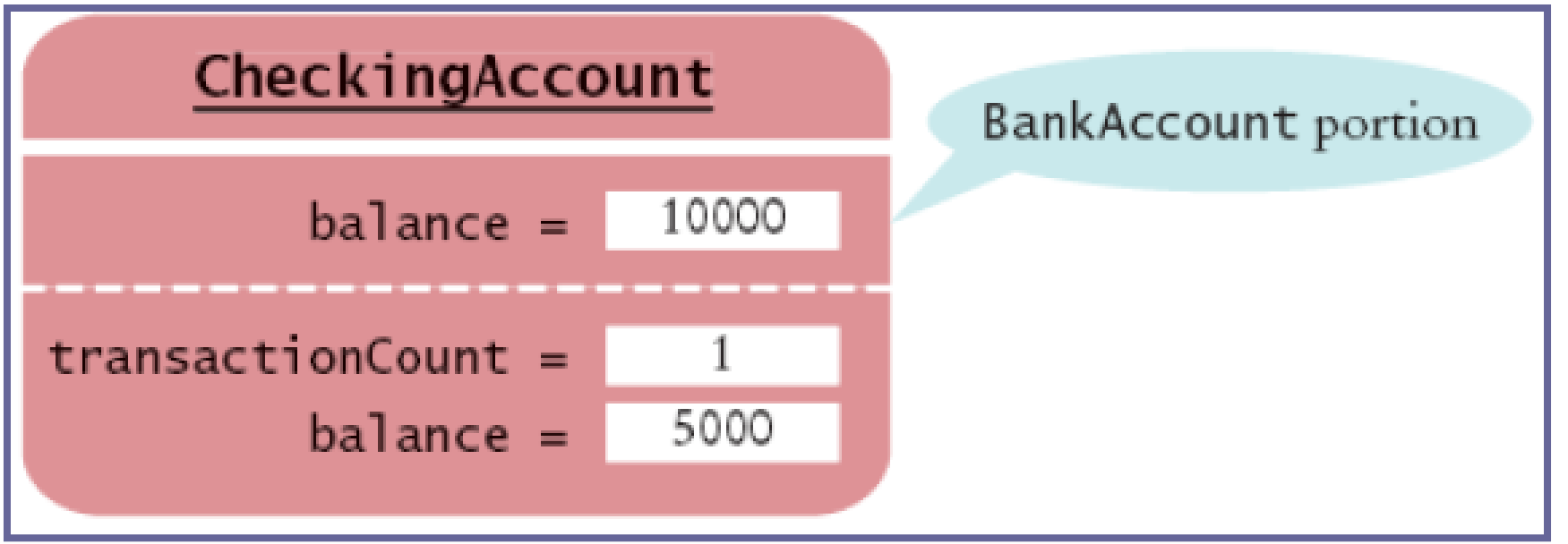
Shadowing Instance Fields

- A subclass has no access to the private instance fields of the superclass
- Beginner's error: "solve" this problem by adding another instance field with same name:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      balance = balance + amount;
   }
   . . .

   private double balance; // Don't
}
```

Common Error: Shadowing Instance Fields

- Now the deposit method compiles, but it doesn't update the correct balance!



Now we have two instance fields for balance. Which one is the correct one?

Subclass Construction

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

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

Must be the ***first*** statement in subclass constructor

Subclass Construction

If subclass constructor doesn't call superclass constructor, default superclass constructor is used

- Default constructor: constructor with no parameters

Syntax : Calling a Superclass Constructor

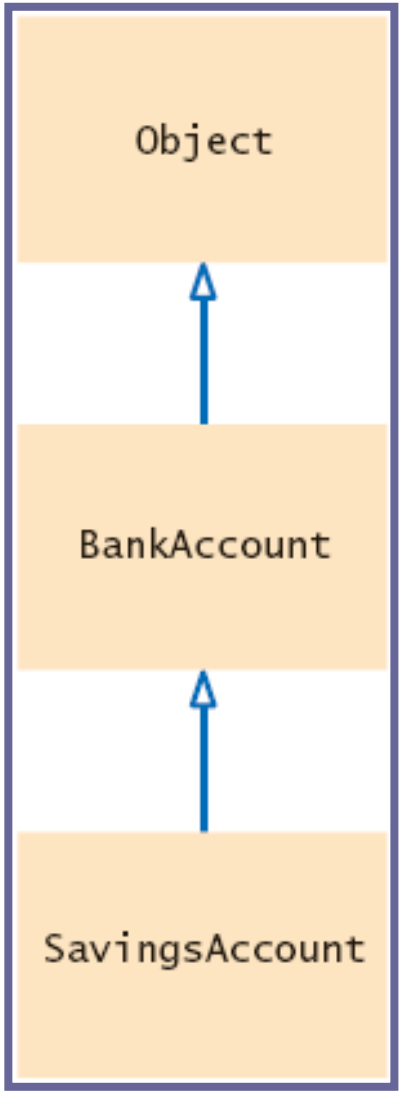
```
ClassName(parameters)
{
   super(parameters);
   . . .
}
```

Example:

```
public CheckingAccount(double initialBalance)
{
      super(initialBalance); // Call super class constructor
      transactionCount = 0;
}
```

Purpose:

To invoke a constructor of the superclass. Note that this statement must be the first statement of the subclass constructor.

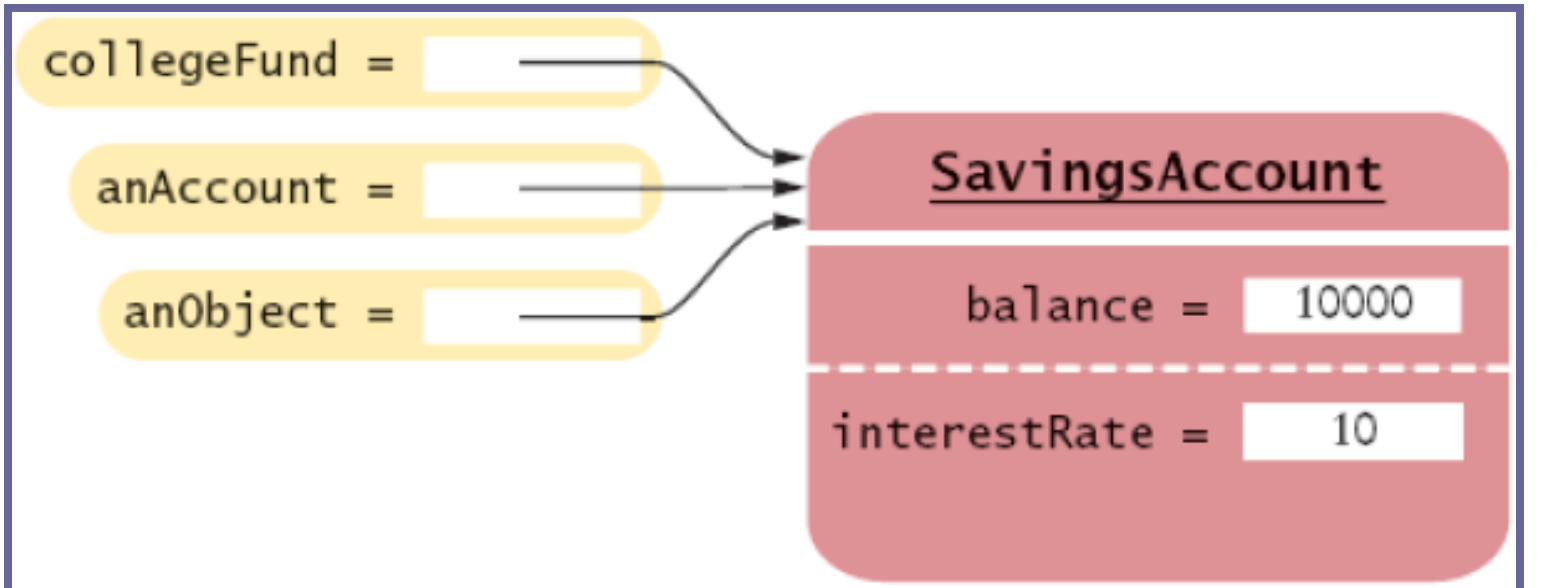

Object
BankAccount
SavingsAccount

Reference

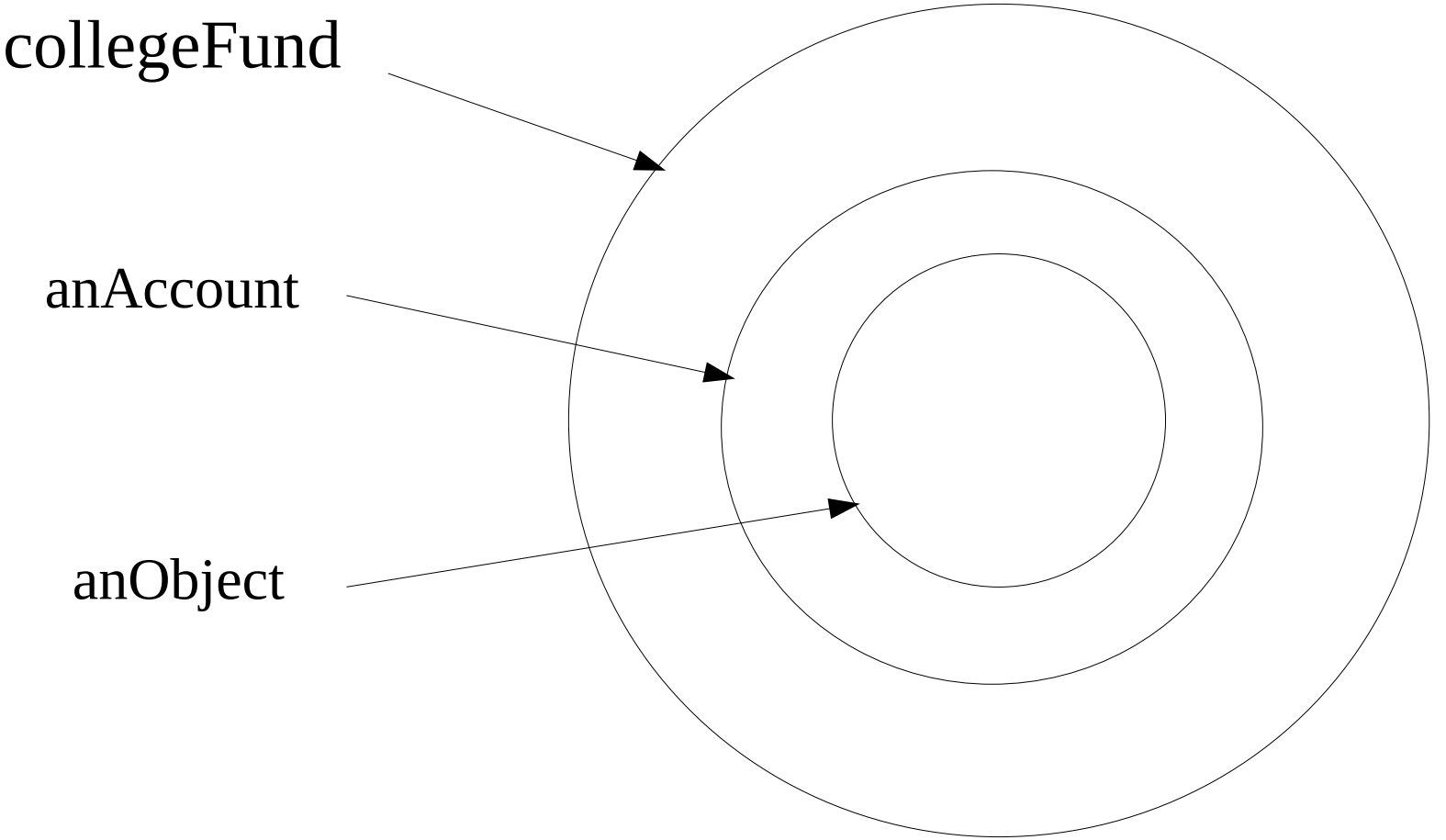
Converting Between Subclass and Superclass Types

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

The three object references all refer to the same object of type SavingsAccount



Reference



Reference (continued)

All three, collegeFund, anAccount, and anObject refer to the same object of type SavingsAccount.

The collegeFound knows all about the SavingAccount

anAccount only knows part of the story. For example we cannot invoke addInterest(...) method of the object. Since it is no idea about addInterest(...) method.

```
AnAccount.addinterest();   // Wrong
```

AnObject knows only a tiny bit of the story. For example we cannot invoke depost(...) method of the object.

```
AnObject.deposit(100);      // Wrong
```

Reference (continued)

Superclass references don't know the full story:

```
anAccount.deposit(1000); // OK
anAccount.addInterest(); // Wrong
// anAccoount is a BankAccount. It has no idea about
// addInterest() method of SavingAccount
```

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

Continued...

Reference (continued)

Superclass references don't know the full story:

```
anObject.deposit(1000); // Wrong
// anObject is an instance of Object. It has no idea
// about deposit() method of BankAccount
```

Continued...

Type Casting

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

Answer: reusability

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`

Typecasting

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

```
BankAccount anAccount = (BankAccount) anObject;
```

This cast is dangerous: if you are wrong, an exception is thrown

Solution: use the **`instanceof`** operator

`instanceof`: tests whether an object belongs to a particular type

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

Syntax : The `InstanceOf` Operator

```
 object instanceof TypeName
```

Example:

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

Purpose:

To return `true` if the *object* is an instance of *`TypeName`* (or one of its subtypes), and `false` otherwise

Next Lecture: Polymorphism

In Java, type of a variable doesn't completely determine type of object to which it refers

```
BankAccount aBankAccount = new SavingsAccount(1000);
// aBankAccount holds a reference to a SavingsAccount
```

Method calls are determined by type of actual object, not type of object reference

```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
   // Calls "deposit" from CheckingAccount
```

Next Lecture: Polymorphism

In Java, type of a variable doesn't completely determine type of object to which it refers

```
BankAccount aBankAccount = new SavingsAccount(1000);
// aBankAccount holds a reference to a SavingsAccount
```

Method calls are determined by type of actual object, not type of object reference

```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
   // Calls "deposit" from CheckingAccount
```

Next Lecture: Polymorphism

- Compiler needs to check that only legal methods are invoked

```
Object anObject = new BankAccount();
anObject.deposit(1000); // Wrong!
```

Summary

A subclass inherits data and behavior from a super class

You can always use a subclass object in place of a superclass object

A subclass inherits all methods that it does not override

A subclass can override a superclass method by providing a new implementation

Use word super to call a super class method.

Summary

Unless specified otherwise, the subclass constructor calls the superclass default constructor

To call a superclass constructor, use the super reserved word in the first statement of the subclass constructor

A subclass reference can be used when a superclass reference is expected