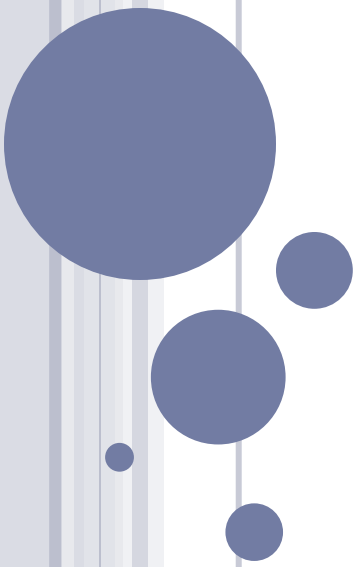


Inheritance

Chapter 10

Inheritance

- Mechanism for enhancing existing classes
- You need to implement a new class
- You have an existing class that represents a more general concept is already available.
- New class can inherit from the existing class.
- Example
  - BankAccount
  - SavingsAccount
  - Most of the methods of bank account apply to savings account
  - You need additional methods.
  - In savings account you only specify new methods.

Inheritance

- More generic form is super class.
- Class that inherits from super class is subclass.
- One advantage – code reuse

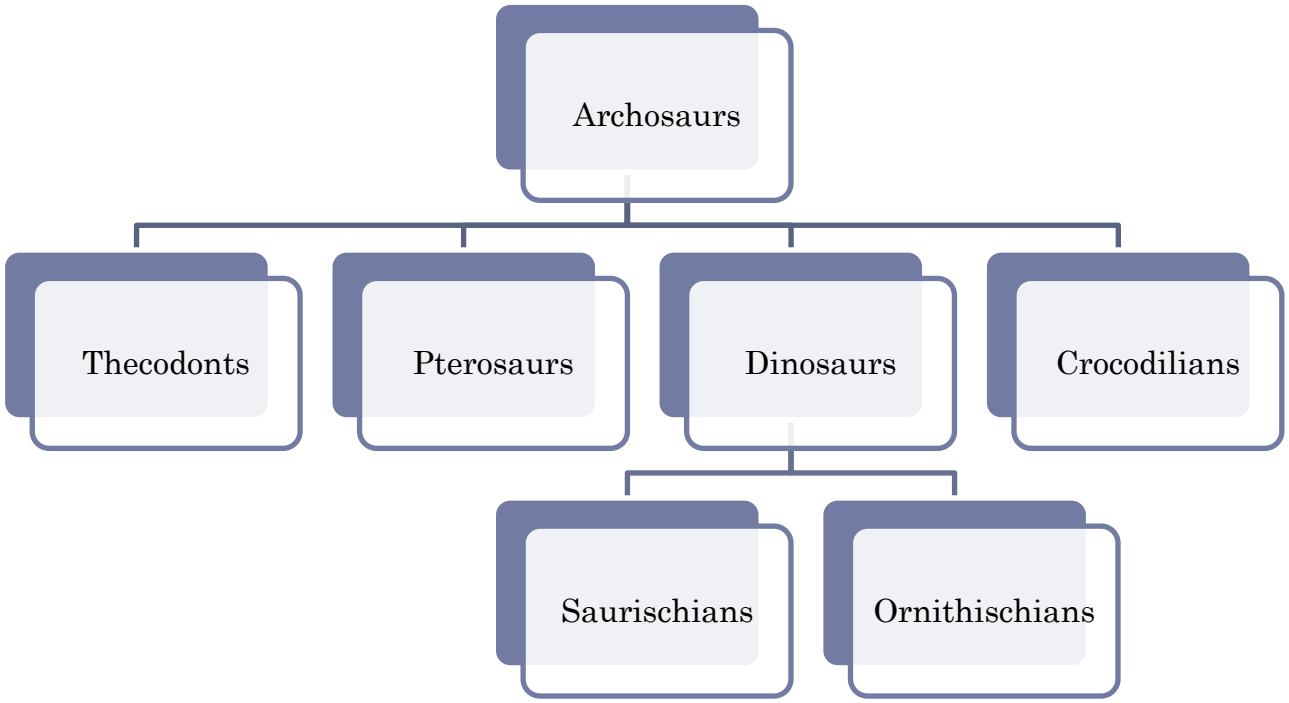
public class SavingsAccount extends BankAccount
{

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

Inheritance

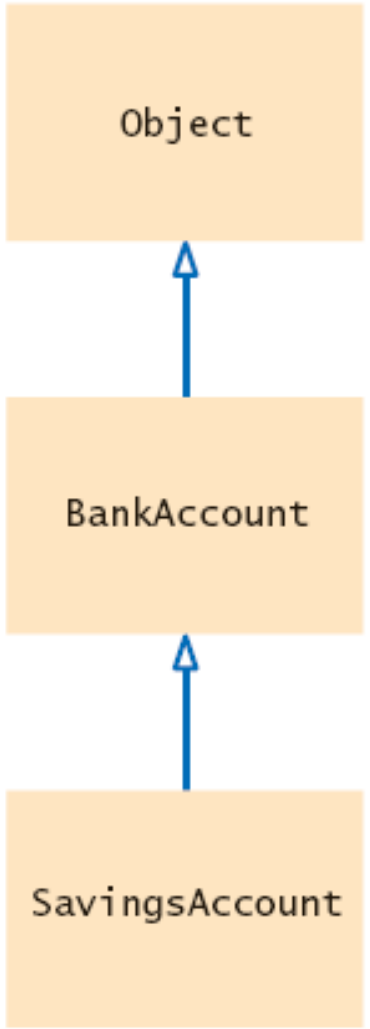
- BankAccount will have the methods
  - deposit( )
  - withdraw( )
  - getBalance( )
- SavingsAccount will have the methods
  - deposit( )
  - withdraw( )
  - getBalance( )
  - addInterest( )

Hierarchies



An Inheritance Diagram

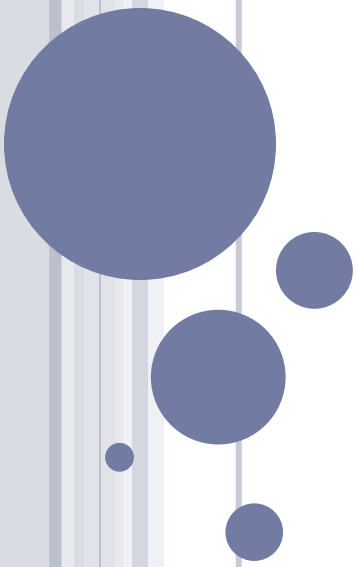
Every class extends the `Object` class either directly or indirectly



Relook at SavingsAccount

public class SavingsAccount extends BankAccount

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



- Encapsulation: `addInterest` calls `getBalance` rather than updating the `balance` field of the superclass (field 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`:

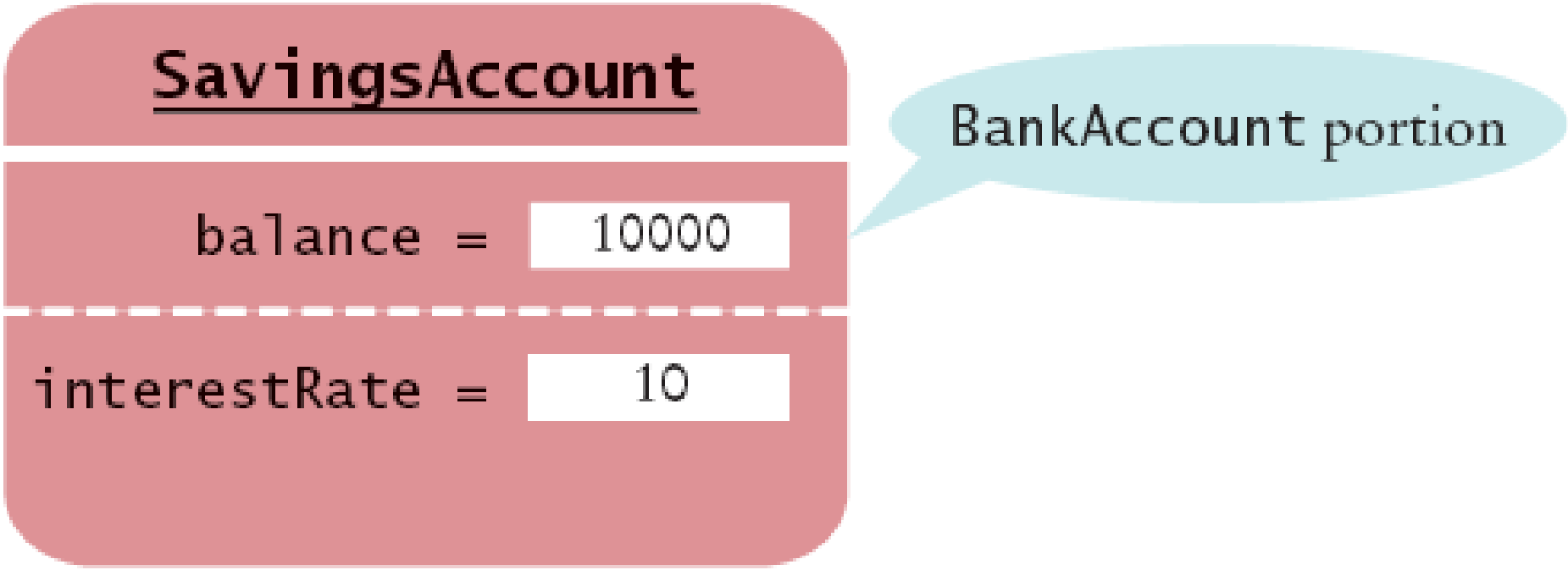


Figure 2 Layout of a Subclass Object

Check

- If the class `Manager` extends the class `Employee`, which class is the `superclass` and which is the `subclass`?

Hierarchy of Bank Accounts

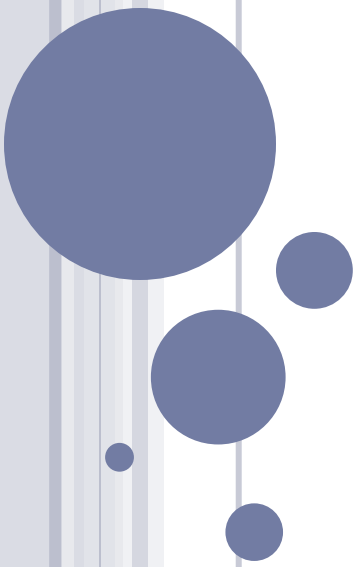
- Consider a bank that offers its customers the following account types:
  1. *Checking account: no interest; small number of free transactions per month, additional transactions are charged a small fee*
  2. *Savings account: earns interest that compounds monthly*
- Inheritance hierarchy:
- All bank accounts support the `getBalance` method
- All bank accounts support the `deposit` and `withdraw` methods, but the implementations differ
- Checking account needs a method `deductFees`; savings account needs a method `addInterest`

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*
- Inherit method:
  - *Don't supply a new implementation of a method that exists in* `superclass`
  - `Superclass` *method can be applied to the subclass objects*
- Add method:
  - *Supply a new method that doesn't exist in the* `superclass`
  - *New method can be applied only to* `subclass` *objects*

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*



Inherited Fields are Private

- Consider `deposit` method of `CheckingAccount`

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

- Can't just add `amount` to `balance`
- `balance` is a *private* field of the superclass
- A subclass has no access to private fields of its superclass
- Subclass must use public interface

Invoking a Superclass Method

- Can't just call

  `deposit(amount)`

  in `deposit` method of `CheckingAccount`

- That is the same as

  `this.deposit(amount)`

- Calls the same method

- Instead, invoke *superclass method*

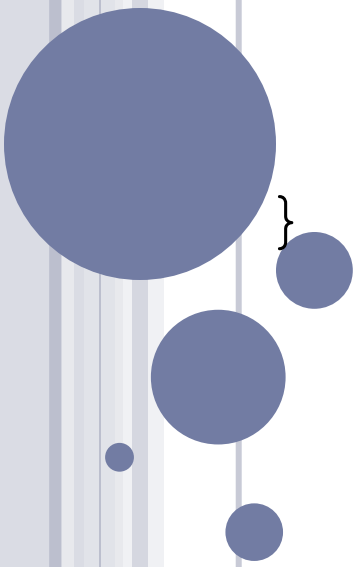
  `super.deposit(amount)`

- Now calls `deposit` method of `BankAccount` class

Invoking a Superclass Method

- Complete method:

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

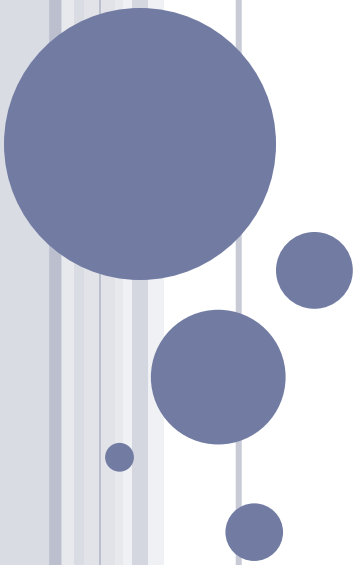


Subclass Constructor

- You write a constructor in the subclass
  - Call the super class constructor
  - Use the word super
  - Must be the first statement of the subclass constructor

Subclass Construction

- If subclass constructor doesn't call superclass constructor, default superclass constructor is used
  - *Default constructor: constructor with no parameters*
  - *If all constructors of the superclass require parameters, then the compiler reports an error*



Converting Between Subclass and Superclass Types

- Ok to convert subclass reference to superclass reference

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

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

Converting Between Subclass and Superclass Types

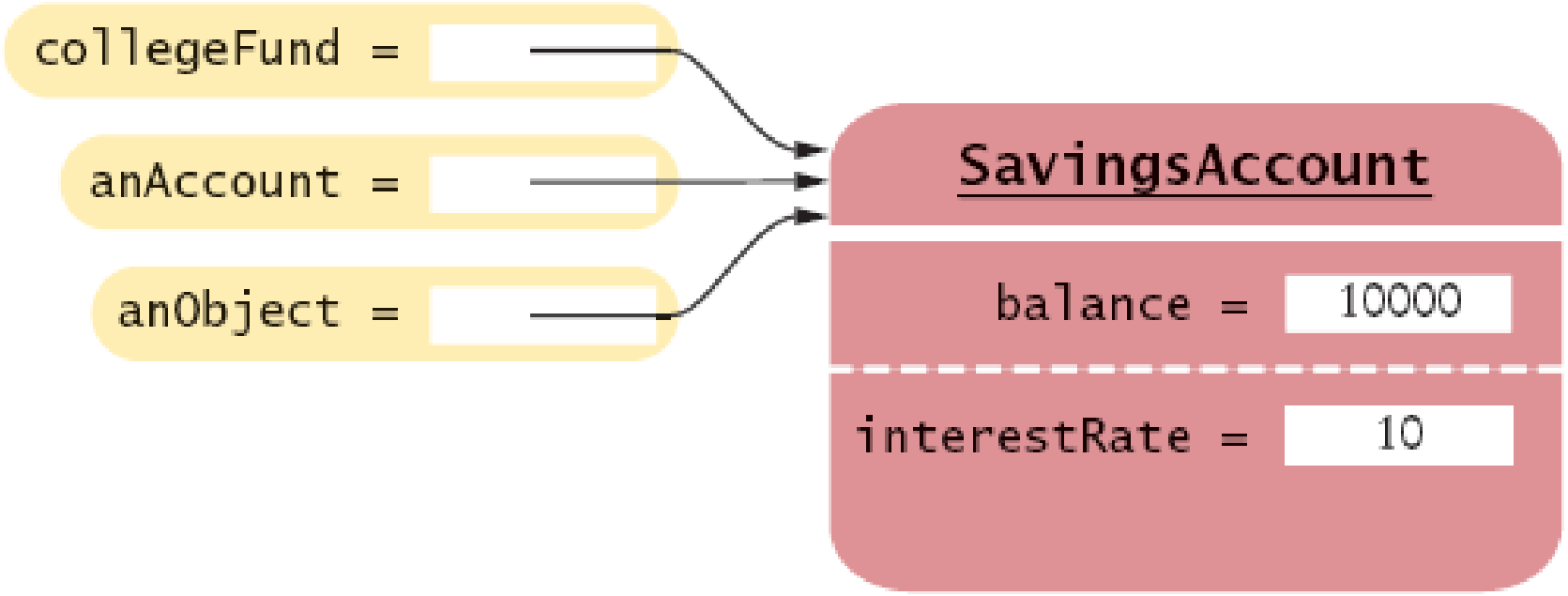


Figure 7 Variables of Different Types Refer to the Same Object

Converting Between Subclass and Superclass Types

- Superclass references don't know the full story:

```
anAccount.deposit(1000); // OK
anAccount.addInterest();
// No--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*

Converting Between Subclass and Superclass Types (cont.)

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

Converting Between Subclass and Superclass Types

- 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;
   . . .
}
```

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
```

- Compiler needs to check that only legal methods are invoked

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

Polymorphism

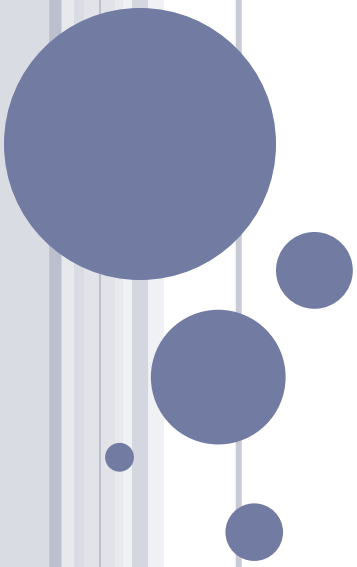
- Polymorphism: ability to refer to objects of multiple types with varying behavior
- Polymorphism at work:

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

- Depending on types of `amount` and `other`, different versions of `withdraw` and `deposit` are called

Access Control

- Java has four levels of controlling access to fields, methods, and classes:
  - *`public` access*
    - *Can be accessed by methods of all classes*
  - *`private` access*
    - *Can be accessed only by the methods of their own class*
  - *`protected` access*
  - *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*



The following table shows the access to members permitted by each modifier.

Access Levels				
Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html

Object: The Cosmic Superclass

- All classes defined without an explicit `extends` clause automatically extend `Object`

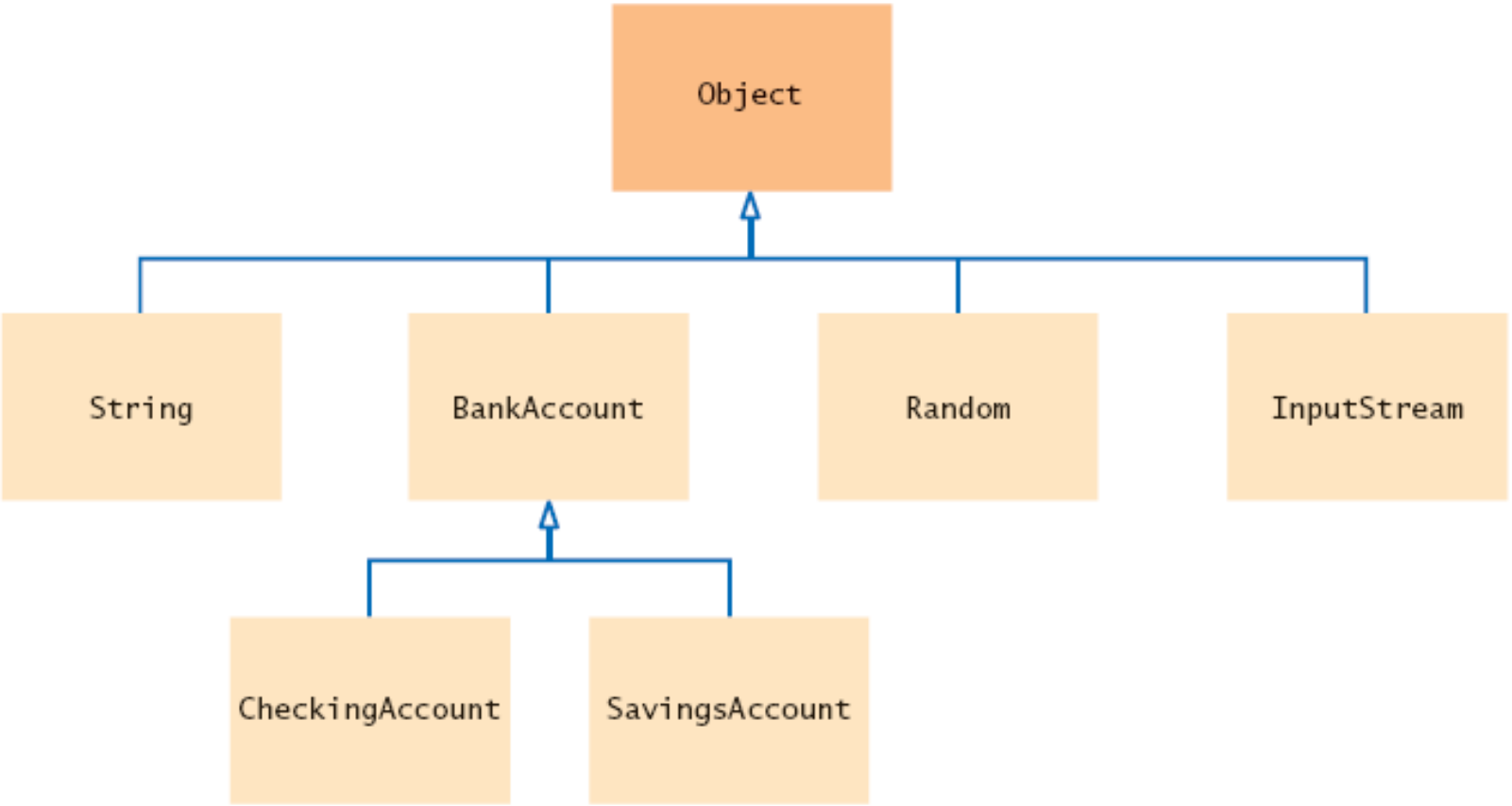


Figure 8 The Object Class Is the Superclass of Every Java Class