Chapter 13

Inheritance

An Introduction to Inheritance

- Inheritance: extend classes by adding methods and fields

```
class SubclassName extends SuperclassName
{
   methods
   instance fields
}
```

- Example: Savings account = bank account with interest

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

An Introduction to Inheritance

- `SavingsAccount` automatically inherits all methods and instance fields of BankAccount

```
SavingsAccount collegeFund = new SavingsAccount(10);
// Savings account with 10% interest
collegeFund.deposit(500);
// OK to use BankAccount method with SavingsAccount object
```

- Extended class = *superclass* (`BankAccount`), extending class = *subclass* (`Savings`)

Continued...

An Inheritance Diagram

- Inheriting from class ≠ implementing interface: subclass inherits behavior and state
- One advantage of inheritance is code reuse
- Every class extends the Object class either directly or indirectly
- In subclass, specify added instance fields, added methods, and changed or overridden methods

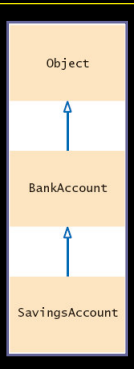


Figure 1:
An Inheritance Diagram

File `BankAccount.java`

```
public class BankAccount
{
   public BankAccount() //Constructor
   {
      balance = 0;
   }

   public BankAccount(double initialBalance) //Parameter
   {                                         //Constructor
      balance = initialBalance;
   }

   public void deposit(double amount) // Method
   {
      balance = balance + amount;
   }

   public void withdraw(double amount) // Method
   {
      balance = balance - amount;
   }
```

Continued...

File `BankAccount.java`

```
public double getBalance() // Method
   {
      return balance;
   }
public void transfer(double amount, BankAccount other) //Method
   {
      withdraw(amount);
      other.deposit(amount);
   }

   private double balance; // private instance field
}
```

A Simpler Hierarchy: 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
- **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`**

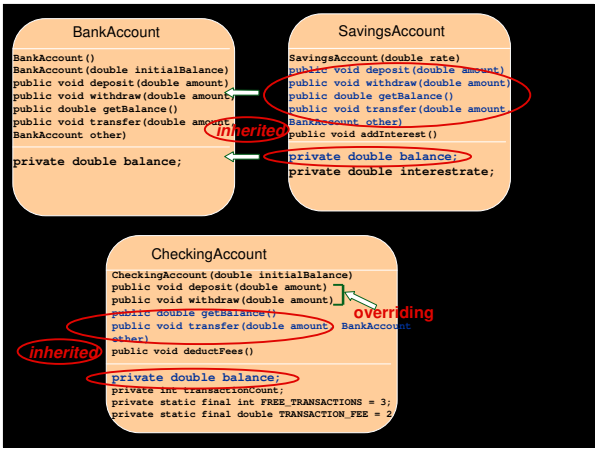
Inheriting Methods

- **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

Continued...

Inheriting Methods

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



Savings Account

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

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

   private double interestRate; // Added instance field
}
```

File `CheckingAccount.java`

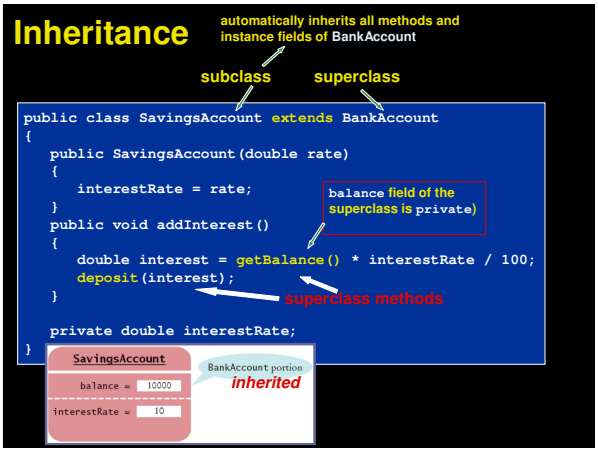
```
public class CheckingAccount extends BankAccount
 {
    public CheckingAccount(double initialBalance) //Constructor
    {
       super(initialBalance);
       transactionCount = 0;
    }
 public void deposit(double amount) //Overriden Method
    {
       transactionCount++;
       super.deposit(amount);
    }

    public void withdraw(double amount) //Overriden Method
    {
       transactionCount++;
       super.withdraw(amount);
    }
```

Continued...

File CheckingAccount.java

```
public void deductFees() //Added Method
{
   if (transactionCount > FREE_TRANSACTIONS)
   {
    double fees = TRANSACTION_FEE *
                (transactionCount - FREE_TRANSACTIONS);
          super.withdraw(fees);
       }
       transactionCount = 0;
    }
              // Added instance fields
    private int transactionCount;
    private static final int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 2.0;
 }
```

Inheritance



An Introduction to Inheritance

- 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)

```
double interest = getBalance() * interestRate/100;
deposit(interest);
//Could write
double interest = super.getBalance() * interestRate/100;
super.deposit(interest);
```

- `super` indicates a call to the superclass method

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

Continued...

Inheriting Instance Fields

- 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

Implementing the CheckingAccount Class

- 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)

Inherited Fields Are Private

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

- `balance` is a *private* field of the superclass
- Subclass must use public interface

```
transactionCount++;
deposit(amount)    ?
```

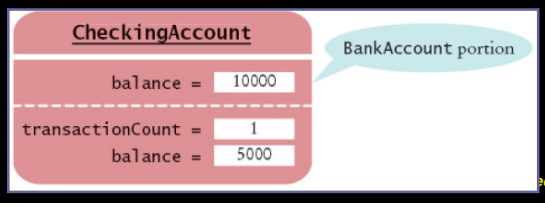
- Equivalent to `this.deposit(amount)` //infinite recursion

```
transactionCount++;
super.deposit(amount)
```

- Instead, invoke *superclass method* `super.deposit(amount)`
- Similar for `withdraw()` method `super.withdraw(amount)`

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:
  - Below, may update balance of CheckingAccount but not of BankAccount



ed...

Subclass Construction

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

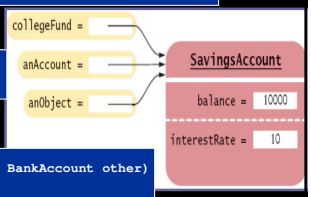
- call to the superclass constructor
- Must be the *first* statement in subclass constructor

- 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

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

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




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

- Code Reuse
  - 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

Continued...

Converting Between Subclass and Superclass Types

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
. . .
Measurable max = coinData.getMaximum(); // Get the largest coin
```

```
String name = max.getName(); // ERROR
```

max isn't a coin, so the compiler throws an exception

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

Converting Between Subclass and Superclass Types

- Solution: use the `instanceof` operator
- `instanceof`: tests whether an object belongs to a particular type

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

Syntax 13.4: 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

Polymorphism

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

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

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

- Compiler needs to check that only legal methods are invoked

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

Polymorphism and Interfaces

- Interface variable holds reference to object of a class that implements the interface
  `Measurable x;`

```
x = new BankAccount(10000);
x = new Coin(0.1, "dime");
```

Note that the object to which `x` refers doesn't have type `Measurable`; the type of the object is some class that implements the Measurable interface

Polymorphism

- You can call any of the interface methods:

```
double m = x.getMeasure();
```

- If `x` refers to a bank account, calls `BankAccount.getMeasure`
- If `x` refers to a coin, calls `Coin.getMeasure`
- Polymorphism (many shapes): Behavior can vary depending on the actual type of an object

Polymorphism

- **Called *late binding*: resolved at runtime**
- **Different from overloading; overloading is resolved by the compiler (*early binding*)**

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
    - See Advanced Topic 13.3

Continued...

Recommended Access Levels

- **Instance and static fields: Always `private`. Exceptions:**
  - `public static` final constants are useful and safe
  - 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); inner classes are usually better

Continued...

Recommended Access Levels

- **Methods: `public` or `private`**
- **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`)
- **Beware of accidental package access (forgetting `public` or `private`)**