In OOP languages, new classes can be derived from an existing class.

Why?
- organizes related classes
- reduces code redundancy
- increases code reuse
- enables polymorphic references

Chapter 13 - Inheritance

Goals
- To learn about inheritance
- To understand how to inherit and override superclass methods
- To be able to invoke superclass constructors
- To learn about protected and package access control
- To understand the common superclass Object and to override its toString and equals methods

13.1 Introduction to Inheritance

Inheritance: extend classes by adding methods and fields

Example: Savings account is a bank account with interest

class SavingsAccount extends BankAccount {
  
  new methods
  new instance fields
}

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

Original/base class is known as the superclass (BankAccount)

extending class is the subclass (SavingsAccount)
Every class extends the `Object` class either directly or indirectly.

Inheritance vs Interface
- Inheriting from class IS NOT the same as implementing interface
  - Subclass inherits behavior and state
  - Interfaces have no state or defined behavior (only names of shared methods)

Code Reuse
- One advantage of inheritance is code reuse
- Not “reinventing the wheel”
  - Already have a class that does some base functions, why not just build up on it?
  - Deposit, withdraw, getBalance common among all accounts

In subclass, specify added instance fields, added methods, and changed or overridden methods
- Inheritance takes care of what is common, you define what is different

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

Encapsulation
- Why do we call `getBalance`?
  ```java
double interest = getBalance() * interestRate / 100;
```
- Encapsulation: `addInterest` calls `getBalance` because `balance` field of the superclass is `private`
- Cannot access private members of another class (even a subclass!)

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

Figure 1 An Inheritance Diagram

Figure 2 Layout of a Subclass Object
Note that `addInterest` calls `getBalance` without specifying an implicit parameter (the calls apply to the same object).

Means the call to `getBalance` is applied to the same object as the object that called `addInterest`.

### 13.2 Inheritance Hierarchies

- Inheritance is a way to categorize.
- In real world, categories often use hierarchies.
  - Generic items yield more specific items:
    - Bird → Robin, Blue Jay, Cardinal, etc.
  - Sets of classes can form complex inheritance hierarchies.

### Syntax

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

### Example

- 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
Inheritance Hierarchy

Behaviors
- 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`

13.3 Inheriting Instance Fields and Methods
- A subclass can define additional instance fields and methods
- With existing methods
  - They can override definitions from the superclass
  - They can inherit them as is

Overriding methods
- 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 (but may not be visible)
- Add field: Supply a new field that doesn't exist in the superclass

Inheriting Methods

- 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
  - Another instance of shadowing

CheckingAccount Class

- Overrides deposit and withdraw to increment the transaction count:

```java
public class CheckingAccount extends BankAccount {
    private int transactionCount; // new instance field
    public void deposit(double amount) {
        transactionCount++;
        // now add amount to balance
        ...
    }
    public void withdraw(double amount) {
        ...
    }
    public void deductFees() { ... } // new method
}
```

- Each CheckingAccount object has two instance fields:
  - balance (inherited from BankAccount)
  - transactionCount (new to CheckingAccount)

Inheriting Private fields

- 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)

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

- Consider deposit method of CheckingAccount

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

Will this work?

- 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

Inheriting Private fields

- Consider deposit method of CheckingAccount

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

Will this work?

- Can't just call `deposit(amount)` in deposit method of CheckingAccount
- That is the same as `this.deposit(amount)` //Checking account!
- Calls the same method (infinite recursion)

Invoking a Superclass Method

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

Solution: super

- Java allows you to specify calling a method of the super class with the keyword `super`
- Invoke superclass method

```java
super.deposit(amount)
```

- Now calls deposit method of BankAccount class
public class CheckingAccount extends BankAccount {
    private static final int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 2.0;
    ...
    public void withdraw(double amount) {
        transactionCount++;
        // Now subtract amount from balance
        super.withdraw(amount);
    }
}

public void deductFees() {
    if (transactionCount > FREE_TRANSACTIONS) {
        double fees = TRANSACTION_FEE * (transactionCount - FREE_TRANSACTIONS);
        super.withdraw(fees);
    }
    transactionCount = 0;
}

Object Class

- A class extends Object by default when no extends clause is used, e.g:

```java
class Thing {
    ...
}
```

```java
class Thing extends Object {
    ...
}
```

Override (redefine)

We can override public (and protected) methods of any superclass

Use the same signature to override an inherited method.

13.4 Inheritance and Constructors

- Unlike members and methods of a superclass, constructors of a superclass are not inherited by its subclasses.
- You must define a constructor for a subclass or use the default constructor added by the compiler.
- How do you initialize superclass fields though?
  - In SavingsAccount, how do we initialize balance?

13.4 Inheritance and Constructors

```java
super();
```

- Calls the default constructor of the superclass
  - Analogous to this()

- Every constructor of a subclass must make a call to the superclass constructor.
  - If you don’t compiler will add in

- A call to super() MUST be the first line of code in the constructor
**Checking Account**

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

---

**13.4 Inheritance and Constructors**

- If a class has a superclass that is not the `Object` class, then a constructor of the class should make an explicit call to a constructor of the superclass.

- Always provide a constructor for every class you define. Don’t rely on default constructors.

- Ok to convert subclass reference to superclass reference (think: BlueJay to Bird)

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

- Converting up to superclass leads to less
  information

- Why would we want this?
  - Reuse code that uses superclass

Super to Sub Conversion

- How do we convert down the chain
  - BankAccount object → CheckingAccount?

- Is this safe?

- We need a way to protect ourselves if we aren’t sure…

- Superclass references don’t know the full story:
  
  ```java
  anAccount.deposit(1000); // OK
  anAccount.addInterest();
  // No—not a method of the class to which
  // anAccount belongs
  ```

- Why is this?

- Conversions
  - Converting up to superclass leads to less information
  - Why would we want this?
    - Reuse code that uses superclass

- Super to Sub Conversion
  - How do we convert down the chain
    - BankAccount object → CheckingAccount?
  
  - Is this safe?

  - We need a way to protect ourselves if we aren’t sure…
instanceof

- Purpose: Check to see if an object is of a particular class
- Give: identifier and class
- Returns: boolean – true if it is that type, false otherwise

Convert from super to sub

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

13.6 Polymorphism

- In Ch.11, we learned that the type of the identifier (Measurable) does not have to match the type of the object (BankAccount, Coin)
- Inheritance demonstrates the same phenomenon
  - A BankAccount identifier can be referring to a BankAccount, CheckingAccount, or SavingsAccount

```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
```

Method calls are always determined on the type of the actual object being stored, not the type of the reference/identifier

This ability to refer to multiple types with varying behavior is called polymorphism

Limitation

- A limitation is that polymorphism only works if the reference type always has an implementation of that call
- Ex. Will the following work?
  ```java
  Measurable x = new BankAccount();
x.deposit(500);
  ```
Why?
- Previously, we called deposit on a BankAccount object.
- When compiling, Java needs to know that a deposit method is legal to call on that object, not which method will be called.
- Even though we didn’t know which version would be called, we can be guaranteed that any object stored with a BankAccount reference can handle deposit.

If the method is specific to only one subclass, then the compiler can’t guarantee legality.

Object anObject = new BankAccount();
anObject.deposit(1000); // Compiling Error

BankAccount ba = new CheckingAccount();
anObject.deductFees(); // Compiling Error

13.7 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
- Can be accessed by methods of this class and subclasses only
- See Advanced Topic 13.3

package access
- The default, when no access modifier is given
- Can be accessed by all classes in the same package

13.8 Object: The Cosmic Superclass
- Recall that everything inherits from **Object**.
- What comes in this class?
- Most useful methods:
  - `toString()`
  - `equals(Object otherObject)`
  - `clone()`
- Why to these matter?
  - Good idea to override these methods

package java.lang;
class Object
- belongs to `java.lang` package
- is the superclass of all other classes
- has several `generic` methods
  - `equals`
  - `toString`
  - `getClass`
  - `clone`
Object Class Methods

- `boolean equals (Object obj)`
  - returns true if (and only if) this object is alias of obj
  - default: compares addresses

- `String toString ()`
  - returns a String representing this object
  - default: `<class name>@<hashCode>`

13.8.1 toString()

- Going to concentrate on the toString method
- Returns a string representation of the object
- Useful for debugging:
  ```java
  Rectangle box = new Rectangle(5, 10, 20, 30);
  String s = box.toString(); // Sets s to
  // "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
  ```

Java insight

- Unlike other methods, toString() can actually be called implicitly
  - Concatenation
    ```java
    "box = " + box;
    ```
  - Calling `print()` or `println()`
    ```java
    System.out.println(box);
    ```

- How can the compiler know to do this?
  - Because every object has atoString method through Inheritance → polymorphism

What does toString return?

- The Object class definition returns the object and a hashcode (identifier)
  ```java
  BankAccount momsSavings = new BankAccount(5000);
  String s = momsSavings.toString();
  // Sets s to something like "BankAccount@d24606bf"
  ```
- Pretty boring…

Overriding toString()

- We can override the definition of any inherited method!
- Usually want to know what’s inside the object (fields, etc)
  - Print name of class, then the values of instance fields in brackets

```java
public String toString()
{
  return "BankAccount[balance=" + balance + "]";
}
```
More useful version

- Now this works better:

```java
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString(); // Sets s to "BankAccount[balance=5000]"
```

Adv 13.4

- `toString` is a little harder with subclasses
- How do we make `BankAccount toString` work for any subclasses?
- Use `getClass()` method

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

equals

- Seen before in the String class
- Tests that the data in the two objects are the same, not that both reference the same object
- Version in Object does the same thing as `==`
- Testing for equality might have different meanings in different classes (do all the data members need to match, or just certain ones?)
- So we need to override!

```java
public boolean equals(Object obj)
{
    BankAccount b = (BankAccount)obj;
    return b.balance == this.balance;
}
```

- Parameter must be of type Object, but we must cast it before we can test the data members
- It is pretty easy for `BankAccount`, with only 1 primitive data member, but gets trickier when data members include arrays and other objects