Programming using Java

Java: Inheritance

Objectives:

- Get an introduction to Inheritance
- Get a general idea of how a hierarchy of classes is put together

Inheritance: Definition

- **inheritance**: a parent-child relationship between classes

- allows sharing of the behavior of the parent class into its child classes
  - one of the major benefits of object-oriented programming (OOP) is this code sharing between classes through inheritance

- child class can add new behavior or override existing behavior from parent

Inheritance terms

- **superclass**, **base class**, **parent class**: terms to describe the parent in the relationship, which shares its functionality
- **subclass**, **derived class**, **child class**: terms to describe the child in the relationship, which accepts functionality from its parent
- **extend**, **inherit**, **derive**: become a subclass of another class

Inheritance in Java

- in Java, you specify another class as your parent by using the keyword `extends`
  - `public class CheckingAccount`
    `extends BankAccount {`

  - the objects of your class will now receive all of the state (fields) and behavior (methods) of the parent class
  - constructors and static methods/fields are not inherited
  - by default, a class's parent is `Object`

Inheritance in Java

- in Java, you specify another class as your parent by using the keyword `extends`
  - `public class CheckingAccount`
    `extends BankAccount {`

- Java forces a class to have exactly one parent ("single inheritance")
  - other languages (C++) allow multiple inheritance

Inheritance Example

```
class BankAccount {
  private double myBal;
  public BankAccount() { myBal = 0; }
  public double getBalance() { return myBal; }
}

class CheckingAccount extends BankAccount {
  private double myInterest;
  public CheckingAccount(double interest) {  }
  public double getInterest() { return myInterest; }
  public void applyInterest() {  }
}
```

- `CheckingAccount` objects have myBal and myInterest fields, and getBalance(), getInterest(), and applyInterest() methods

Multiple layers of inheritance

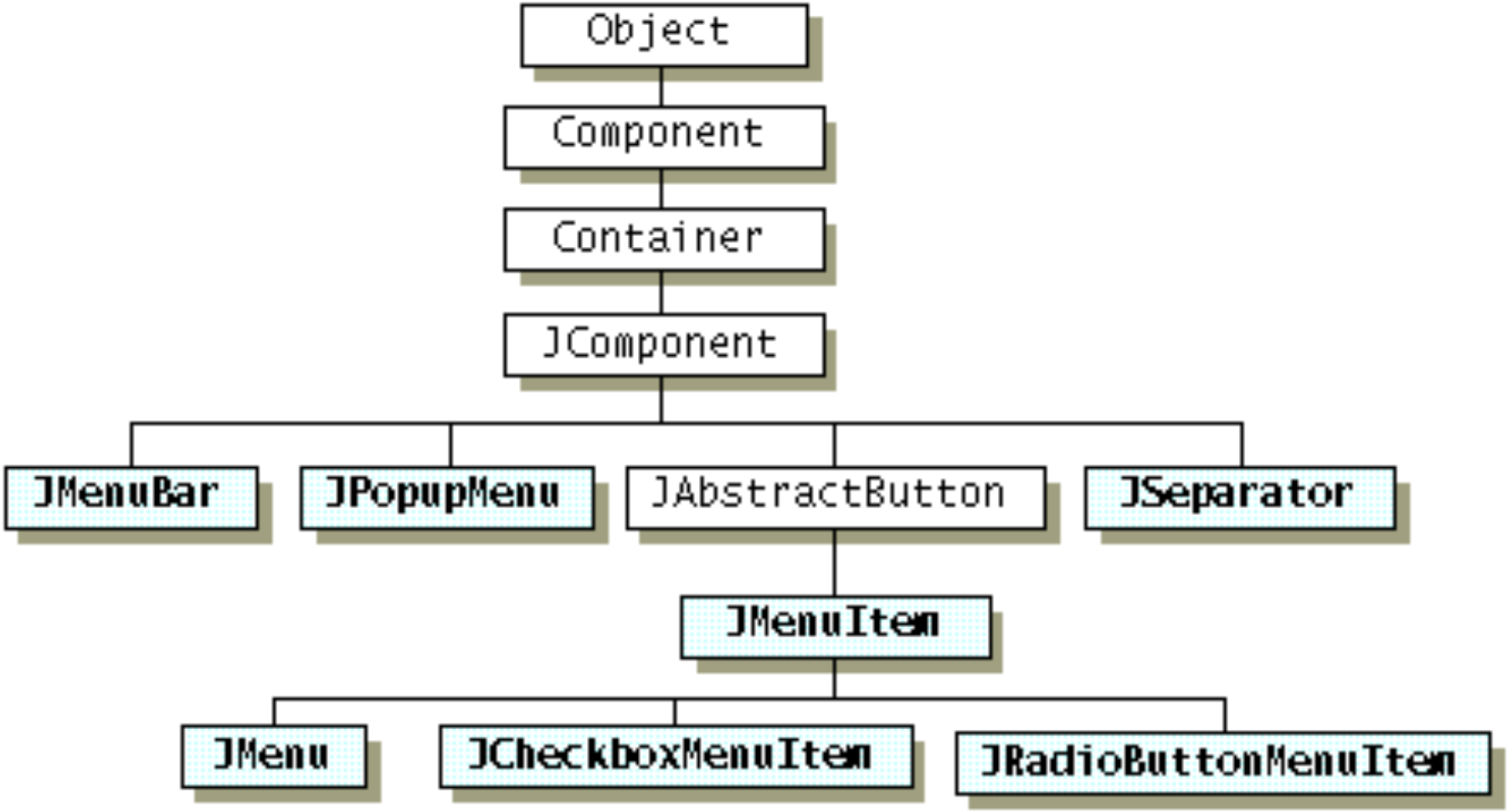
- it is possible to extend a class that itself is a child class; inheritance chains like this can be arbitrarily deep

```
public class TransactionFeeCheckingAccount
    extends CheckingAccount {
  private static final double FEE = 2.00;

  public void chargeFee() {
    withdraw(FEE);
  }
}
```

Inheritance Hierarchies

- Deeper layered chain of classes, many children extending many layers of parents



"Has-a" Relationships

- "Has-a" relationship: when one object contains another as a field

```
public class BankAccountManager {
  private List myAccounts;
  // ...
}
```

- a `BankAccountManager` object "has-a" `List` inside it, and therefore can use it

"Is-a" relationships

- "Is-a" relationships represent sets of abilities; implemented through interfaces and inheritance

```
public class CheckingAccount
  extends BankAccount {
  // ...
}
```

- a `CheckingAccount` object "is-a" `BankAccount`
  - therefore, it can do anything an `BankAccount` can do
  - it can be substituted wherever a `BankAccount` is needed
  - a variable of type `BankAccount` may refer to a `CheckingAccount` object

Using the account classes

- `CheckingAccount` inherits `BankAccount`'s methods

```
CheckingAccount c = new CheckingAccount(0.10);
System.out.println(c.getBalance());
c.applyInterest();
```

-);

Using the account classes

- a `BankAccount` variable can refer to a `CheckingAccount` object

```
BankAccount b2 = new CheckingAccount(0.06);
System.out.println(b2.getBalance());
```

- an `Object` variable can point to either account type

```
Object o  = new BankAccount();
Object o2 = new CheckingAccount(0.09);
```

Some code that won't compile

- `CheckingAccount` variable can't refer to `BankAccount` (not every `BankAccount` "is-a" `CheckingAccount`)

```
CheckingAccount c = new BankAccount();
```

- cannot call a `CheckingAccount` method on a variable of type `BankAccount` (can only use `BankAccount` behavior)

```
BankAccount b = new CheckingAccount(0.10);
b.applyInterest();
```

- cannot use any account behavior on an `Object` variable

```
Object o = new CheckingAccount(0.06);
System.out.println(o.getBalance());
o.applyInterest();
```

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overriding behavior

- Child class can replace the behavior of its parent's methods by redefining them

```
public class BankAccount {
  private double myBalance;  // ....
  public String toString() {
    return getID() + " $" + getBalance();
} }

public class FeeAccount extends BankAccount {

  private static final double FEE = 2.00;
  public String toString() {  // overriding
    return getID() + " $" + getBalance()
      + " (Fee: $" + FEE + ")";
} }
```

Overriding behavior example

```
BankAccount b = new BankAccount("Ed",9.0);
FeeAccount f = new FeeAccount("Jen", 9.0);

System.out.println(b);
System.out.println(f);
```

- Output:

```
Ed $9.0
Jen $9.0 (Fee: $2.0)
```

Overloading vs. Overriding

- Don't confuse the concepts of overloading and overriding
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

Access modifiers

- `public`: visible to all other classes
```
public class BankAccount
```

- `private`: visible only to the current class, its methods, and every instance (object) of its class
  - a child class cannot refer to its parent's private members!
```
private String myID;
```

- `protected` (this one's new to us): visible to the current class, and all of its child classes
```
protected int myWidth;
```

- package (default access; no modifier): visible to all classes in the current "package" (seen later)
```
int myHeight;
```

Access modifier problem

```
public class Parent {
  private int field1;
  protected int field2;
  public int field3;
  private void method1() {}
  public void method2() {}
  protected void setField1(int value) {
    field1 = value;
  }
}
```

Access modifier problem

```
public class Child extends Parent {
  public int field4;

  public Child() {          // Which are legal?
    field4 = 0;             // ____________
    field1++;               // ____________
    field2++;               // ____________
    field3++;               // ____________
    method1();              // ____________
    method2();              // ____________
    setField1(field4);      // ____________
  }
}
```

Some code that won't compile

```
public class Point2D {
  private int x, y;
  public Point2D(int x, int y) {
    this.x = x;   this.y = y;
  }
}

public class Point3D extends Point2D {
  private int z;
  public Point3D(int x, int y, int z) {
    this.x = x;  this.y = y;  // can't do this!
    this.z = z;
  }
}
```