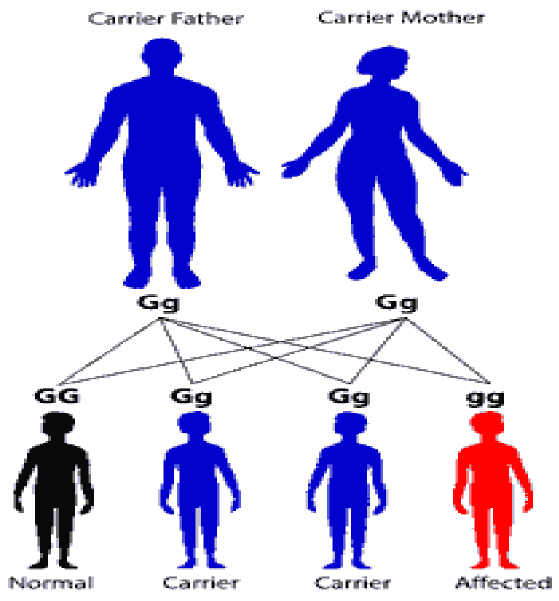
CECS 277, Spring 2018 Lecture Notes

Inheritance



`super` keyword

- used to refer to superclass (parent) of current class
- can be used to refer to parent class's methods, variables, constructors to call them
  - needed when there is a name conflict with current class
- useful when overriding and you want to keep the old behavior but add new behavior to it

- syntax:

```
super(args);                 // call parent’s constructor
super.fieldName              // access parent’s field
super.methodName(args);      //    or method
```

`super` example

```
public class BankAccount {
   private double myBalance;
   public void withdraw(double amount) {
      myBalance -= amount;
} }

public class FeeAccount
                              extends BankAccount {
   public void withdraw(double amount) {
      super.withdraw(amount);
      if (getBalance() < 100.00)
         withdraw(2.00); // charge $2 fee
} }
```

- didn't need to say `super.getBalance()` because the `FeeAccount` subclass doesn't override that method (it is unambiguous which version of the method we are talking about)

`super` and constructors

- if the superclass has a constructor that requires any arguments (not `()`), you *must* put a constructor in the subclass and have it call the super-constructor (call to super-constructor must be the first statement)

```
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) {
    super(x, y);   // calls Point2D constructor
    this.z = z;
  }
}
```

The `instanceof` keyword

- Performs run-time type check on the object referred to by a reference variable
- Usage: *object-reference* `instanceof` *type*
  (result is a boolean expression)
  - if *type* is a class, evaluates to true if the variable refers to an object of *type* or any subclass of it.
  - if *type* is an interface, evaluates to true if the variable refers to an object that implements that interface.
  - if *object-reference* is null, the result is false.
- Example:

```
Object o = myList.get(2);
if (o instanceof BankAccount)
    ((BankAccount)o).deposit(10.0);
```

Some `instanceof` problems

```
Object o = new BankAccount(...);
BankAccount c = new CheckingAccount(...);
BankAccount n = new NumberedAccount(...);
CheckingAccount c2 = null;
                                    T/F ???

o instanceof Object                 _______
o instanceof BankAccount            _______
o instanceof CheckingAccount        _______
c instanceof BankAccount            _______
c instanceof CheckingAccount        _______
n instanceof CheckingAccount        _______
n instanceof Comparable             _______
n instanceof NumberedAccount        _______
c2 instanceof Object                _______
```

Which method gets called?

```
BankAccount b = new FeeAccount("Ed", 9.00);
b.withdraw(5.00);
System.out.println(b.getBalance());
```

- Will it call the `withdraw` method in `BankAccount`, leaving Ed with $4?
- Will it call the `withdraw` method in `FeeAccount`, leaving Ed with $2 (after his $2 fee)?

The answer: dynamic binding

- The version of `withdraw` from `FeeAccount` will be called

- The version of an object's method that gets executed is always determined by that object's type, *not* by the type of the variable

- The variable should only be looked at to determine whether the code would compile; after that, all behavior is determined by the object itself

Static and Dynamic Binding

- **static binding**: methods and types that are hard-wired at compile time
  - static methods
  - referring to instance variables
  - the types of the reference variables you declare

- **dynamic binding**: methods and types that are determined and checked as the program is running
  - non-static (a.k.a virtual) methods that are called
  - types of objects that your variables refer to

A dynamic binding problem

```
class A {
  public void method1() { System.out.println("A1"); }
  public void method3() { System.out.println("A3"); }
}

class B extends A {
  public void method2() { System.out.println("B2"); }
  public void method3() { System.out.println("B3"); }
}

A var1 = new B();
Object var2 = new A();

var1.method1();
var1.method2();
var2.method1();
((A)var2).method3();
```

OUTPUT???

Type-casting and objects

- You cannot call a method on a reference unless the reference's type has that method

```
Object o = new BankAccount("Ed", 9.00);
o.withdraw(5.00);  // doesn't compile
```

- You can cast a reference to any subtype of its current type, and this will compile successfully

```
((BankAccount)o).withdraw(5.00);
```

Down-casting and runtime

- It is illegal to cast a reference variable into an unrelated type (example: casting a `String` variable into a `BankAccount`)

- It is legal to cast a reference to the wrong subtype; this will compile but crash when the program runs
  - Will crash even if the type you cast it to has the method in question

```
((String)o).toUpperCase();        // crashes
((FeeAccount)o).withdraw(5.00);  // crashes
```

Inner classes

- **Inner classes** are classes defined within other classes
  - The class that includes the inner class is called the **outer class**
  - There is no particular location where the definition of the inner class (or classes) must be place within the outer class
  - Placing it first or last, however, will guarantee that it is easy to find

Inner classes

- An inner class definition is a member of the outer class in the same way that the instance variables and methods of the outer class are members
  - An inner class is local to the outer class definition
  - The name of an inner class may be reused for something else outside the outer class definition
  - If the inner class is private, then the inner class cannot be accessed by name outside the definition of the outer class

Inner classes

```
public class Outer
{
          private class Inner
          {
                    // inner class instance variables
                    // inner class methods

          } // end of inner class definition

          // outer class instance variables
          // outer class methods
}
```

Inner classes

- There are two main advantages to inner classes
  - They can make the outer class more self-contained since they are defined inside a class
  - Both of their methods have access to each other's private methods and instance variables
- Using an inner class as a helping class is one of the most useful applications of inner classes
  - If used as a helping class, an inner class should be marked private

Inner classes

- Within the definition of a method of an inner class:
  - It is legal to reference a private instance variable of the outer class
  - It is legal to invoke a private method of the outer class
  - Essentially, the inner class has a hidden reference to the outer class
- Within the definition of a method of the outer class
  - It is legal to reference a private instance variable of the inner class on an object of the inner class
  - It is legal to invoke a (nonstatic) method of the inner class <u>as long as an object of the inner class is used as a calling object</u>
- Within the definition of the inner or outer classes, the modifiers **`public`** and **`private`** are equivalent

Inner classes

Display 13.9 Class with an Inner Class *(Part 1 of 2)*

```
public class BankAccount
{
    private class Money
    {
        private long dollars;
        private int cents;

        public Money(String stringAmount)
        {
            abortOnNull(stringAmount);
            int length = stringAmount.length();
            dollars = Long.parseLong(
                            stringAmount.substring(0, length − 3));
            cents = Integer.parseInt(
                            stringAmount.substring(length − 2, length));
        }

        public String getAmount()
        {
            if (cents > 9)
               return (dollars + "." + cents);
            else
               return (dollars + ".0" + cents);
        }
```

The modifier `private` in this line should not be changed to `public`. However, the modifiers `public` and `private` inside the inner class `Money` can be changed to anything else and it would have no effect on the class `BankAccount`.

Inner classes

Display 13.9 Class with an Inner Class *(Part 1 of 2)* (continued)

```
        public void addIn(Money secondAmount)
        {
            abortOnNull(secondAmount);
            int newCents = (cents + secondAmount.cents)%100;
            long carry = (cents + secondAmount.cents)/100;
            cents = newCents;
            dollars = dollars + secondAmount.dollars + carry;
        }

       private void abortOnNull(Object o)
       {
           if (o == null)
           {
               System.out.println("Unexpected null argument.");
               System.exit(0);
           }
        }
    }
```

The definition of the inner class ends here, but the definition of the outer class continues in Part 2 of this display.

Inner classes

Display 13.9 Class with an Inner Class *(Part 2 of 2)*

```
    private Money balance;

    public BankAccount()
    {
        balance = new Money("0.00");
    }

    public String getBalance()
    {
        return balance.getAmount();
    }

    public void makeDeposit(String depositAmount)
    {
        balance.addIn(new Money(depositAmount));
    }

    public void closeAccount()
    {
        balance.dollars = 0;
        balance.cents = 0;
    }
}
```

To invoke a nonstatic method of the inner class outside of the inner class, you need to create an object of the inner class.

This invocation of the inner class method `getAmount()` would be allowed even if the method `getAmount()` were marked as `private`.

Notice that the outer class has access to the private instance variables of the inner class.

This class would normally have more methods, but we have only included the methods we need to illustrate the points covered here.

Inner classes

- If an inner class is marked **`public`**, then it can be used outside of the outer class
- In the case of a nonstatic inner class, it must be created using an object of the outer class

```
BankAccount account = new BankAccount();
BankAccount.Money amount =
  account.new Money("41.99");
```

  - Note that the prefix ***`account.`*** must come before ***`new`***
  - The new object **`amount`** can now invoke methods from the inner class, but only from the inner class

Inner classes

- In the case of a static inner class, the procedure is similar to, but simpler than, that for nonstatic inner classes

```
OuterClass.InnerClass innerObject =
                    new OuterClass.InnerClass();
```

  - Note that all of the following are acceptable

```
innerObject.nonstaticMethod();
innerObject.staticMethod();
OuterClass.InnerClass.staticMethod();
```

Inner classes

If the Money inner class in the BankAccount example was defined as public, we can create and use objects of type Money outside the BankAccount class.

```
// this is okay in main( )
BankAccount account = new BankAccount( );
BankAccount.Money amt =    // note syntax
                account.new Money( "41.99" );
System.out.println( amt.getAmount( ) );
// but NOT this – why not??
System.out.println( amt.getBalance( ) );
```

Multiple Inner Classes

- A class can have as many inner classes as it needs.
- Inner classes have access to each other's private members as long as an object of the other inner class is used as the calling object.

The .class File for an Inner Class

- Compiling any class in Java produces a `.class` file named ***`ClassName.class`***

- Compiling a class with one (or more) inner classes causes both (or more) classes to be compiled, and produces two (or more) .class files

  - Such as ***`ClassName.class`*** **and** ***`ClassName$InnerClassName.class`***

Nesting Inner Classes

- It is legal to nest inner classes within inner classes
  - The rules are the same as before, but the names get longer
  - Given class **`A`**, which has public inner class **`B`**, which has public inner class **`C`**, then the following is valid:

```
A aObject = new A();
A.B bObject = aObject.new B();
A.B.C cObject = bObject.new C();
```

References

- Koffman/Wolfgang Ch. 3, pp. 125-149, 155-159; Appendix B.1, pp. 738-744
- *The Java Tutorial:* Implementing Nested Classes. http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html
- *The Java Tutorial:* Inheritance. http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html