CSC 20 Chapter 9: Inheritance

Inheritance

Code reuse is a major goal of object-oriented programming. One way to exploit code reuse is through inheritance.

Mechanically, inheritance is not very complicated. If we write

```
public class B extends A {
    ...
}
```

we are saying that if we have `B b = new B()` object method access follows this process.

Method dispatch:
~ `b.foo()` looks in class B to see if `foo` is defined there. If so, it is executed.
~ If not, it looks in A (because B extends A), and executes it if it is there.
~ Whichever `foo` executes, if it make a method call, the same process occurs (search in A only if it's not in B).

Here are some examples.

```
public class A {
    public foo() {
        System.out.println("foo");
    }
}

public class B extends A {

}

public class Main {
    public static void main(String[] args) {
        B b = new B();
        b.foo();          // "foo" gets printed
    }
}
```

In the following example, `` `B:bar `` is output, even though it is A's `foo` that is being executed.

```
class A {
    public void foo() {
        bar();
    }
    public void bar() {
        System.out.println("A:bar");
    }
}

class B extends A {
    public void bar() {
        System.out.println("B:bar");
    }
}
```

```
class Untitled {
    public static void main(String[] args) {
        B a = new B();
        a.foo();
    }
}
```

Examples

#1

```
import java.util.*;

class StringList {

    private String list[];
    private int numElements;

    public StringList() {
        numElements = 0;
        list = new String[100];
    }

    public void add(String s) {
        ... code to add s to the end of the list ...
    }

    public indexOf(String s) {
        ... code to report the first occurrence of s ...
    }

    public String toString() {
        ... code to create a String representation of the list ...
    }

    ... etc ...
}
```

Now, if we wanted to sometimes keep our Strings in sorted order, the code would not be much different. Rather than recode a whole new class, we can inherit all of the functionality and specify only what differs.

```
class SortedStringList extends StringList {

    public void add(String s) {
        ... code to add s into its sorted position ...
    }

}
```

Since this is the only difference between the operation of the two classes, only this one method needed writing.

Some notes:
~ list and numElements needs to be declared "protected" instead of "private" to allow subclasses to access them

directly.
~ The superclass constructor is called automatically when the subclass is created.
~ If you want to call a different superclass constructor, use "super(argument list)".
~ Declaring a method in a subclass with a name used in a superclass is allowed and is called "method overriding".
~ If you want to call a superclass's overridden method, use super.methodname().

#2 Here's a simple program that reads from a file and echoes what's read to `System.out`.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ProcessLines {

    public void process() {
        Scanner file = null;
        Scanner in = new Scanner(System.in);
        System.out.print("File: ");
        String name = in.nextLine();
        try {
            file = new Scanner(new File(name));
        }
        catch (FileNotFoundException e) {
            System.err.println("File not found");
            return;
        }
        while (file.hasNextLine()) {
            System.out.println(processLine(file.nextLine()));
        }
    }

    public String processLine(String s) {
        return s;
    }

    public static void main(String[] args) {
        ProcessLines x = new ProcessLines();
        x.process();
    }
}
```

We can now create other programs that process lines using inheritance and just a few lines of code.

```
public class ToUpper extends ProcessLines {

    public String processLine(String s) {
        return s.toUpperCase();
    }

    public static void main(String[] args) {
        ToUpper x = new ToUpper();
        x.process();
    }
}
```

#3

Chapter 9 Table 9.10 shows what would be printed if various methods were invoked.

```
class E extends F {
    public void method2() {
        System.out.print("E 2 ");
        method1();
    }
}

class F extends G {
    public String toString() {
        return ("F");
    }
    public void method2() {
        System.out.print("F 2 ");
        super.method2();
    }
}

class G {
    public String toString() {
        return ("G");
    }
    public void method1() {
        System.out.print("G 1 ");
    }
    public void method2() {
        System.out.print("G 2 ");
    }
}

class H extends E {
    public void method1() {
        System.out.print("H 1 ");
    }
}

public class Main {
    public static void main(String[] args) {
        G o = new G();
        System.out.println(o);
        o.method1();
        System.out.println();
        o.method2();
    }
}
```

Change `G o = new G();` to each of the different class names and try to predict what will print.

Polymorphism

When a variable can refer to different types, and the variable's behavior depends on the actual type of the object it refers to.

```
class Animal {
    public void says() {
        System.out.println("oops!");
    }
}

class Dog extends Animal {
    public void says() {
        System.out.println("woof!");
    }
}

class Cat extends Animal {
    public void says() {
        System.out.println("meow!");
    }
}

class Pig extends Animal {
    public void says() {
        System.out.println("oink!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal[] alist = new Animal[3];
        alist[0] = new Dog();
        alist[1] = new Cat();
        alist[2] = new Pig();
        for (int i=0; i<alist.length; i++) {
            x[i].says();
        }
    }
}
```

Here's a nicer main(). It uses an array initializer and a for-each loop

```
    public static void main(String[] args) {
        Animal[] alist = {new Dog(), new Cat(), new Pig()};
        for (Animal x : alist) {
            x.says();
        }
    }
```

Compile and Runtime errors

Errors involving method dispatch and/or assignment with inheritance and polymorphism are summarized here.

Checks done at compile time

1. If an object of type B is assigned to a variable, the variable must be declared type B or a type inherited by B.

```
// Let's say B extends A (ie, B inherits from A)
A w;
B y;
w=new A(); // OK - same type
w=new B(); // OK - assign to type above in inheritance hierarchy
y=new A(); // NO - cannot assign to type below in inheritance hierarchy
```

2. If a variable is declared type A, only methods defined or inherited in class A can be called.

```
A x = ...
x.foo();   // OK if and only if A defines or inherits a method called foo
```

3. If a variable is cast to type A, only methods defined or inherited in class A can be called.

```
((A)x).foo();   // OK if and only if A defines or inherits a method called foo
```

Checks done at runtime

1. If a variable x holds a reference to an A object, then `x.foo()` invokes the foo defined or inherited in A regardless of the declared or cast type of x. If no foo exists, a runtime error occurs.

```
// Let's say B and C both extend A
B w = new B();
A x = w;       // OK - assign to type above in inheritance hierarchy
w.foo();       // Invokes the foo defined in B because w is a reference to a B
x.foo();       // Invokes the foo defined in B because x is a reference to a B
((C)x).foo();  // Invokes the foo defined in B because x is a reference to a B
```

2. Casting variable x to type A is allowed only if x is referring to an object of type A or a type inherited by A. Otherwise a `ClassCastException` occurs.