Reusing Classes

Introduction

- One of the most compelling features about Java is code reuse
- You reuse code by creating new classes, but instead of creating them from scratch
- The trick is to use the classes without soiling the existing code
- Two ways to accomplish this:
- ***Composition***
  - Simply create objects of your existing class inside the new class, and simply reuse the functionality of the code, not its form
- ***Inheritance***
  - It creates a new class as *a type of* an existing class
  - Take the form of the existing class and add code to it without modifying the existing class

Composition syntax

- **Simply place object references inside new classes**

```
1  class WaterSource {
2    private String s;
3    WaterSource() {
4      System.out.println("WaterSource()");
5      s = "Constructed";
6    }
7    public String toString() { return s; }
8  }
9
10 public class SprinklerSystem {
11   private String valve1, valve2, valve3, valve4;
12   private WaterSource source = new WaterSource();
13   private int i;
14   private float f;
15   public String toString() {
16     return
17       "valve1 = " + valve1 + " " +
18       "valve2 = " + valve2 + " " +
19       "valve3 = " + valve3 + " " +
20       "valve4 = " + valve4 + "\n" +
21       "i = " + i + " " + "f = " + f + " " +
22       "source = " + source;
23   }
24   public static void main(String[] args) {
25     SprinklerSystem sprinklers = new SprinklerSystem();
26     System.out.println(sprinklers);
27   }
28 }
```

Composition syntax (Cont.)

```
1  import static net.mindview.util.Print.*;
2
3  class Soap {
4    private String s;
5    Soap() {
6      print("Soap()");
7      s = "Constructed";
8    }
9    public String toString() { return s; }
10 }
11
12 public class Bath {
13   private String // Initializing at point of definition:
14     s1 = "Happy",
15     s2 = "Happy",
16     s3, s4;
17   private Soap castille;
18   private int i;
19   private float toy;
20   public Bath() {
21     print("Inside Bath()");
22     s3 = "Joy";
23     toy = 3.14f;
24     castille = new Soap();
25   }
26   // Instance initialization:
27   { i = 47; }
28   public String toString() {
29     if(s4 == null) // Delayed initialization:
30       s4 = "Joy";
31     return
32       "s1 = " + s1 + "\n" +
33       "s2 = " + s2 + "\n" +
34       "s3 = " + s3 + "\n" +
35       "s4 = " + s4 + "\n" +
36       "i = " + i + "\n" +
37       "toy = " + toy + "\n" +
38       "castille = " + castille;
39   }
40   public static void main(String[] args) {
41     Bath b = new Bath();
42     print(b);
43   }
44 }
```

- If you want the references initialized, you can do it:
- At the point the objects are defined
- In the constructor for that class
- Right before you actually need to use the object
  - Lazy initialization
  - It can reduce overhead in situations where object creation is expensive and the object doesn't need to be created every time
- Using instance initialization

Inheritance syntax

- Inheritance is an integral part of Java (and all OOP languages)
- You are always doing inheritance when you create a class
  - Because unless you explicitly inherit from some other class, you implicitly inherit from Java's standard root class Object
- When you inherit, you say "This new class is like that old class."
- Use the keyword ***extends*** followed by the name of the *base class*
  - When you do this, you automatically get all the fields and methods in the base class

Inheritance syntax (Cont.)

```
1  import static net.mindview.util.Print.*;
2
3  class Cleanser {
4    private String s = "Cleanser";
5    public void append(String a) { s += a; }
6    public void dilute() { append(" dilute()"); }
7    public void apply() { append(" apply()"); }
8    public void scrub() { append(" scrub()"); }
9    public String toString() { return s; }
10   public static void main(String[] args) {
11     Cleanser x = new Cleanser();
12     x.dilute(); x.apply(); x.scrub();
13     print(x);
14   }
15 }
16
17 public class Detergent extends Cleanser {
18   // Change a method:
19   public void scrub() {
20     append(" Detergent.scrub()");
21     super.scrub(); // Call base-class version
22   }
23   // Add methods to the interface:
24   public void foam() { append(" foam()"); }
25   // Test the new class:
26   public static void main(String[] args) {
27     Detergent x = new Detergent();
28     x.dilute();
29     x.apply();
30     x.scrub();
31     x.foam();
32     print(x);
33     print("Testing base class:");
34     Cleanser.main(args);
35   }
36 }
```

- To allow for inheritance, as a general rule make all fields *private* and all methods *public*
  - In particular cases you must make adjustments, but this is a useful guideline
  - Think of inheritance as reusing the class
- You might want to call the method from the base class inside the new version
- Java has the keyword *super* that refers to the “superclass” that the current class inherits

Initializing the base class

- Two classes involved: the base class and the derived class
- Create an object of the derived class, it contains within it a *subobject* of the base class
  - This subobject is the same as if you had created an object of the base class by itself
  - the *subobject* of the base class is wrapped within the derived-class object
- Perform the initialization in the constructor by calling the base-class constructor

```
1  import static net.mindview.util.Print.*;
2
3  class Art {
4    Art() { print("Art constructor"); }
5  }
6
7  class Drawing extends Art {
8    Drawing() { print("Drawing constructor"); }
9  }
10
11 public class Cartoon extends Drawing {
12   public Cartoon() { print("Cartoon constructor"); }
13   public static void main(String[] args) {
14     Cartoon x = new Cartoon();
15   }
16 }
```

```
/* Output:
Art constructor
Drawing constructor
Cartoon constructor
*///:~
```

Constructors with arguments

- Call a base-class constructor that has an argument
- Explicitly write the calls to the base-class constructor using the *super* keyword and the appropriate argument list
- The call to the base-class constructor must be the first thing you do in the derived-class constructor

```
1  import static net.mindview.util.Print.*;
2
3  class Game {
4    Game(int i) {
5      print("Game constructor");
6    }
7  }
8
9  class BoardGame extends Game {
10   BoardGame(int i) {
11     super(i);
12     print("BoardGame constructor");
13   }
14 }
15
16 public class Chess extends BoardGame {
17   Chess() {
18     super(11);
19     print("Chess constructor");
20   }
21   public static void main(String[] args) {
22     Chess x = new Chess();
23   }
24 }
```

```
/* Output:
Game constructor
BoardGame
constructor
Chess constructor
*///:~
```

Delegation

- This is midway between inheritance and composition
  - Place a member object in the class you're building (like composition)
  - Expose all the methods from the member object in your new class (like inheritance)
- E.g., a spaceship needs a control module

```
1  public class SpaceShipControls {
2    void up(int velocity) {}
3    void down(int velocity) {}
4    void left(int velocity) {}
5    void right(int velocity) {}
6    void forward(int velocity) {}
7    void back(int velocity) {}
8    void turboBoost() {}
9  }
```

- One way to build a spaceship is to use inheritance

```
1  public class SpaceShip extends SpaceShipControls {
2    private String name;
3    public SpaceShip(String name) { this.name = name; }
4    public String toString() { return name; }
5    public static void main(String[] args) {
6      SpaceShip protector = new SpaceShip("NSEA Protector");
7      protector.forward(100);
8    }
9  }
```

Delegation

```
1  public class SpaceShipDelegation {
2    private String name;
3    private SpaceShipControls controls =
4      new SpaceShipControls();
5    public SpaceShipDelegation(String name) {
6      this.name = name;
7    }
8    // Delegated methods:
9    public void back(int velocity) {
10     controls.back(velocity);
11   }
12   public void down(int velocity) {
13     controls.down(velocity);
14   }
15   public void forward(int velocity) {
16     controls.forward(velocity);
17   }
18   public void left(int velocity) {
19     controls.left(velocity);
20   }
21   public void right(int velocity) {
22     controls.right(velocity);
23   }
24   public void turboBoost() {
25     controls.turboBoost();
26   }
27   public void up(int velocity) {
28     controls.up(velocity);
29   }
30   public static void main(String[] args) {
31     SpaceShipDelegation protector =
32       new SpaceShipDelegation("NSEA Protector");
33     protector.forward(100);
34   }
35 }
```

- **Although the Java language doesn't support delegation, development tools often do**

Combining composition and inheritance

- **Common to use composition and inheritance together**

```
1  import static net.mindview.util.Print.*;
2
3  class Plate {
4    Plate(int i) {
5      print("Plate constructor");
6    }
7  }
8
9  class DinnerPlate extends Plate {
10   DinnerPlate(int i) {
11     super(i);
12     print("DinnerPlate constructor");
13   }
14 }
15
16 class Utensil {
17   Utensil(int i) {
18     print("Utensil constructor");
19   }
20 }
21
22 class Spoon extends Utensil {
23   Spoon(int i) {
24     super(i);
25     print("Spoon constructor");
26   }
27 }
28
29 class Fork extends Utensil {
30   Fork(int i) {
31     super(i);
32     print("Fork constructor");
33   }
34 }
35
```

```
36 class Knife extends Utensil {
37   Knife(int i) {
38     super(i);
39     print("Knife constructor");
40   }
41 }
42
43 // A cultural way of doing something:
44 class Custom {
45   Custom(int i) {
46     print("Custom constructor");
47   }
48 }
49
50 public class PlaceSetting extends Custom {
51   private Spoon sp;
52   private Fork frk;
53   private Knife kn;
54   private DinnerPlate pl;
55   public PlaceSetting(int i) {
56     super(i + 1);
57     sp = new Spoon(i + 2);
58     frk = new Fork(i + 3);
59     kn = new Knife(i + 4);
60     pl = new DinnerPlate(i + 5);
61     print("PlaceSetting constructor");
62   }
63   public static void main(String[] args) {
64     PlaceSetting x = new PlaceSetting(9);
65   }
66 }
```

Guaranteeing proper cleanup

- **Java doesn't have the C++ concept of a *destructor*, a method that is automatically called when an object is destroyed**
  - **Allow the garbage collector to reclaim the memory as necessary**
- **There are times when your class might perform some activities during its lifetime that require cleanup**
  - **Cannot know when the garbage collector will be called, or if it will be called**
- **To clean up something for a class, you must explicitly write a special method to do it**
- **You must guard against an exception by putting such cleanup in a *finally* clause**
- **Pay attention to the calling order for the base-class and member-object cleanup methods in case one subobject depends on another**

Guaranteeing proper cleanup (Cont.)

```
1  package reusing;
2  import static net.mindview.util.Print.*;
3
4  class Shape {
5    Shape(int i) { print("Shape constructor"); }
6    void dispose() { print("Shape dispose"); }
7  }
8
9  class Circle extends Shape {
10   Circle(int i) {
11     super(i);
12     print("Drawing Circle");
13   }
14   void dispose() {
15     print("Erasing Circle");
16     super.dispose();
17   }
18 }
19
20 class Triangle extends Shape {
21   Triangle(int i) {
22     super(i);
23     print("Drawing Triangle");
24   }
25   void dispose() {
26     print("Erasing Triangle");
27     super.dispose();
28   }
29 }
30
31 class Line extends Shape {
32   private int start, end;
33   Line(int start, int end) {
34     super(start);
35     this.start = start;
36     this.end = end;
37     print("Drawing Line: " + start + ", " + end);
38   }
39   void dispose() {
40     print("Erasing Line: " + start + ", " + end);
41     super.dispose();
42   }
43 }
```

```
44
45 public class CADSystem extends Shape {
46   private Circle c;
47   private Triangle t;
48   private Line[] lines = new Line[3];
49   public CADSystem(int i) {
50     super(i + 1);
51     for(int j = 0; j < lines.length; j++)
52       lines[j] = new Line(j, j*j);
53     c = new Circle(1);
54     t = new Triangle(1);
55     print("Combined constructor");
56   }
57   public void dispose() {
58     print("CADSystem.dispose()");
59     // The order of cleanup is the reverse
60     // of the order of initialization:
61     t.dispose();
62     c.dispose();
63     for(int i = lines.length - 1; i >= 0; i--)
64       lines[i].dispose();
65     super.dispose();
66   }
67   public static void main(String[] args) {
68     CADSystem x = new CADSystem(47);
69     try {
70       // Code and exception handling...
71     } finally {
72       x.dispose();
73     }
74   }
75 }
```

Guaranteeing proper cleanup (Cont.)

- Pay attention to the calling order for the base-class and member-object cleanup methods in case one subobject depends on another
- In general, you should follow the same form that is imposed by a C++ compiler on its destructors
  - Perform all of the cleanup work specific to your class, in the reverse order of creation
  - Call the base-class cleanup method
- You can't rely on garbage collection for anything but memory reclamation
- If you want cleanup to take place, make your own cleanup methods and don't use on *finalize( )*

Name hiding

- Overloading works regardless of whether the method was defined at this level or in a base class (unlike C++)
- Java SE5 has added the *@Override* annotation

```
1  import static net.mindview.util.Print.*;
2
3  class Homer {
4    char doh(char c) {
5      print("doh(char)");
6      return 'd';
7    }
8    float doh(float f) {
9      print("doh(float)");
10     return 1.0f;
11   }
12 }
13
14 class Milhouse {}
15
16 class Bart extends Homer {
17   void doh(Milhouse m) {
18     print("doh(Milhouse)");
19   }
20 }
21
22 public class Hide {
23   public static void main(String[] args) {
24     Bart b = new Bart();
25     b.doh(1);
26     b.doh('x');
27     b.doh(1.0f);
28     b.doh(new Milhouse());
29   }
30 }
```

```
1  class Lisa extends Homer {
2    @Override void doh(Milhouse m) {
3      System.out.println("doh(Milhouse)");
4    }
5  }
```

Choosing composition vs. inheritance

- Both composition and inheritance allow you to place subobjects inside your new class
  - Composition explicitly does this—with inheritance it's implicit
- Composition is generally used when you want the features of an existing class inside your new class, but not its interface
  - Embed an object so that you can use it to implement features in your new class
  - the user of your new class sees the interface you've defined for the new class rather than the interface from the embedded object
- When you inherit, you take an existing class and make a special version of it
  - Take a general-purpose class and specializing it for a particular need

Choosing composition vs. inheritance

```
1  class Engine {
2    public void start() {}
3    public void rev() {}
4    public void stop() {}
5  }
6
7  class Wheel {
8    public void inflate(int psi) {}
9  }
10
11 class Window {
12   public void rollup() {}
13   public void rolldown() {}
14 }
15
16 class Door {
17   public Window window = new Window();
18   public void open() {}
19   public void close() {}
20 }
21
22 public class Car {
23   public Engine engine = new Engine();
24   public Wheel[] wheel = new Wheel[4];
25   public Door
26     left = new Door(),
27     right = new Door(); // 2-door
28   public Car() {
29     for(int i = 0; i < 4; i++)
30       wheel[i] = new Wheel();
31   }
32   public static void main(String[] args) {
33     Car car = new Car();
34     car.left.window.rollup();
35     car.wheel[0].inflate(72);
36   }
37 }
```

- **The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition**

protected

- In an ideal world, the *private* keyword would be enough
  - In real projects, there are times when you want to make something hidden from the world at large and yet allow access for members of derived classes
- The *protected* keyword is a nod to pragmatism
  - This is *private* as far as the class user is concerned, but available to anyone who inherits from this class or anyone else in the same *package*
- Although it's possible to create *protected* fields, the best approach is to leave the fields *private*
- You should always preserve your right to change the underlying implementation

protected

```
1  import static net.mindview.util.Print.*;
2
3  class Villain {
4    private String name;
5    protected void set(String nm) { name = nm; }
6    public Villain(String name) { this.name = name; }
7    public String toString() {
8      return "I'm a Villain and my name is " + name;
9    }
10 }
11
12 public class Orc extends Villain {
13   private int orcNumber;
14   public Orc(String name, int orcNumber) {
15     super(name);
16     this.orcNumber = orcNumber;
17   }
18   public void change(String name, int orcNumber) {
19     set(name); // Available because it's protected
20     this.orcNumber = orcNumber;
21   }
22   public String toString() {
23     return "Orc " + orcNumber + ": " + super.toString();
24   }
25   public static void main(String[] args) {
26     Orc orc = new Orc("Limburger", 12);
27     print(orc);
28     orc.change("Bob", 19);
29     print(orc);
30   }
31 }
```

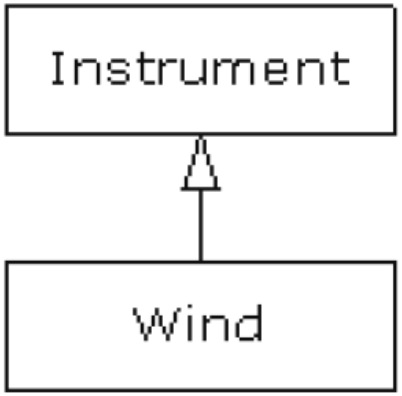
Upcasting

- This relationship between the new class and the base class can be summarized by saying
  - "The new class is *a type of* the existing class"
- Inheritance means that all of the methods in the base class are also available in the derived class

```
1  class Instrument {
2    public void play() {}
3    static void tune(Instrument i) {
4      // ...
5      i.play();
6    }
7  }
8
9  // Wind objects are instruments
10 // because they have the same interface:
11 public class Wind extends Instrument {
12   public static void main(String[] args) {
13     Wind flute = new Wind();
14     Instrument.tune(flute); // Upcasting
15   }
16 }
```

Why “upcasting”?

- Casting from a derived type to a base type moves up on the inheritance diagram, so it’s commonly referred to as *upcasting*
- Upcasting is always **safe** because you’re going from a more specific type to a more general type
- The only thing that can occur to the class interface during the upcast is that it can lose methods, not gain them



Composition vs. inheritance revisited

- Although inheritance gets a lot of emphasis while learning OOP, it doesn't mean that you should use it everywhere you possibly can
- On the contrary, you should use it sparingly, only when it's clear that inheritance is useful
- How to determine whether you should use composition or inheritance
  - Ask whether you'll ever need to upcast from your new class to the base class
  - If you remember to ask "Do I need to upcast?" you'll have a good tool for deciding between composition and inheritance

The *final* keyword

- Java's *final* keyword has slightly different meanings depending on the context
  - "This cannot be changed."
- Prevent changes for two reasons: design or efficiency
  - It's possible to misuse the final keyword
- Three places where *final* can be used: for data, methods, and classes

final data

- Many programming languages have a way to tell the compiler that a piece of data is “constant”
- A constant is useful for two reasons:
  - It can be a compile-time constant that won’t ever change
  - It can be a value initialized at run time that you don’t want changed
- In Java, constants must be primitives and are expressed with the *final* keyword
  - A value must be given at the time of definition of such a constant
- A field that is both *static* and *final* has only one piece of storage that cannot be changed

final data

- With a primitive, *final* makes the value a constant
- With an object reference, *final* makes the reference a constant
  - It can never be changed to point to another object
  - By convention, fields that are both *static* and *final* (that is, compile-time constants) are capitalized and use underscores to separate

```
1  import java.util.*;
2  import static net.mindview.util.Print.*;
3
4  class Value {
5    int i; // Package access
6    public Value(int i) { this.i = i; }
7  }
8
9  public class FinalData {
10   private static Random rand = new Random(47);
11   private String id;
12   public FinalData(String id) { this.id = id; }
13   // Can be compile-time constants:
14   private final int valueOne = 9;
15   private static final int VALUE_TWO = 99;
16   // Typical public constant:
17   public static final int VALUE_THREE = 39;
18   // Cannot be compile-time constants:
19   private final int i4 = rand.nextInt(20);
20   static final int INT_5 = rand.nextInt(20);
21   private Value v1 = new Value(11);
22   private final Value v2 = new Value(22);
23   private static final Value VAL_3 = new Value(33);
24   // Arrays:
25   private final int[] a = { 1, 2, 3, 4, 5, 6 };
26   public String toString() {
27     return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
28   }
29   public static void main(String[] args) {
30     FinalData fd1 = new FinalData("fd1");
31     //! fd1.valueOne++; // Error: can't change value
32     fd1.v2.i++; // Object isn't constant!
33     fd1.v1 = new Value(9); // OK -- not final
34     for(int i = 0; i < fd1.a.length; i++)
35       fd1.a[i]++; // Object isn't constant!
36     //! fd1.v2 = new Value(0); // Error: Can't
37     //! fd1.VAL_3 = new Value(1); // change reference
38     //! fd1.a = new int[3];
39     print(fd1);
40     print("Creating new FinalData");
41     FinalData fd2 = new FinalData("fd2");
42     print(fd1);
43     print(fd2);
44   }
45 }
```

Blank *finals*

- Java allows the creation of blank ***final***s
  - Fields that are declared as ***final*** but are not given an initialization value
- Blank ***final***s provide much more flexibility in the use of the ***final*** keyword
- Perform assignments to finals either with an expression at the point of definition of the field or in every constructor

```
1  import java.util.*;
2  import static net.mindview.util.Print.*;
3
4  class Value {
5    int i; // Package access
6    public Value(int i) { this.i = i; }
7  }
8
9  public class FinalData {
10   private static Random rand = new Random(47);
11   private String id;
12   public FinalData(String id) { this.id = id; }
13   // Can be compile-time constants:
14   private final int valueOne = 9;
15   private static final int VALUE_TWO = 99;
16   // Typical public constant:
17   public static final int VALUE_THREE = 39;
18   // Cannot be compile-time constants:
19   private final int i4 = rand.nextInt(20);
20   static final int INT_5 = rand.nextInt(20);
21   private Value v1 = new Value(11);
22   private final Value v2 = new Value(22);
23   private static final Value VAL_3 = new Value(33);
24   // Arrays:
25   private final int[] a = { 1, 2, 3, 4, 5, 6 };
26   public String toString() {
27     return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
28   }
29   public static void main(String[] args) {
30     FinalData fd1 = new FinalData("fd1");
31     //! fd1.valueOne++; // Error: can't change value
32     fd1.v2.i++; // Object isn't constant!
33     fd1.v1 = new Value(9); // OK -- not final
34     for(int i = 0; i < fd1.a.length; i++)
35       fd1.a[i]++; // Object isn't constant!
36     //! fd1.v2 = new Value(0); // Error: Can't
37     //! fd1.VAL_3 = new Value(1); // change reference
38     //! fd1.a = new int[3];
39     print(fd1);
40     print("Creating new FinalData");
41     FinalData fd2 = new FinalData("fd2");
42     print(fd1);
43     print(fd2);
44   }
45 }
```

Blank *finals*

- Java allows the creation of blank ***final***s
  - Fields that are declared as ***final*** but are not given an initialization value
- Blank ***final***s provide much more flexibility in the use of the ***final*** keyword
- Perform assignments to finals either with an expression at the point of definition of the field or in every constructor

```
1  class Poppet {
2    private int i;
3    Poppet(int ii) { i = ii; }
4  }
5
6  public class BlankFinal {
7    private final int i = 0; // Initialized final
8    private final int j; // Blank final
9    private final Poppet p; // Blank final reference
10   // Blank finals MUST be initialized in the constructor:
11   public BlankFinal() {
12     j = 1; // Initialize blank final
13     p = new Poppet(1); // Initialize blank final reference
14   }
15   public BlankFinal(int x) {
16     j = x; // Initialize blank final
17     p = new Poppet(x); // Initialize blank final reference
18   }
19   public static void main(String[] args) {
20     new BlankFinal();
21     new BlankFinal(47);
22   }
23 }
```

final arguments

- Java allows you to make arguments *final* by declaring them as such in the argument list
  - You can read the argument, but you can't change it
  - This feature is primarily used to pass data to *anonymous inner classes*

```
1  class Gizmo {
2    public void spin() {}
3  }
4
5  public class FinalArguments {
6    void with(final Gizmo g) {
7      //! g = new Gizmo(); // Illegal -- g is final
8    }
9    void without(Gizmo g) {
10     g = new Gizmo(); // OK -- g not final
11     g.spin();
12   }
13   // void f(final int i) { i++; } // Can't change
14   // You can only read from a final primitive:
15   int g(final int i) { return i + 1; }
16   public static void main(String[] args) {
17     FinalArguments bf = new FinalArguments();
18     bf.without(null);
19     bf.with(null);
20   }
21 }
```

final methods

- There are two reasons for *final* methods
  - The first is put a "lock" on the method to prevent any inheriting class from changing its meaning
  - The second reason for *final* methods is efficiency
  - Allowed the compiler to turn any calls to that method into *inline calls*

final and *private*

- Any ***private*** methods in a class are implicitly ***final***
  - Because you can't access a private method, you can't override it
  - You can add the ***final*** specifier to a private method, but it doesn't give that method any extra meaning

```
1  import static net.mindview.util.Print.*;
2
3  class WithFinals {
4    // Identical to "private" alone:
5    private final void f() { print("WithFinals.f()"); }
6    // Also automatically "final":
7    private void g() { print("WithFinals.g()"); }
8  }
9
10 class OverridingPrivate extends WithFinals {
11   private final void f() {
12     print("OverridingPrivate.f()");
13   }
14   private void g() {
15     print("OverridingPrivate.g()");
16   }
17 }
18
19 class OverridingPrivate2 extends OverridingPrivate {
20   public final void f() {
21     print("OverridingPrivate2.f()");
22   }
23   public void g() {
24     print("OverridingPrivate2.g()");
25   }
26 }
```

```
27
28 public class FinalOverridingIllusion {
29   public static void main(String[] args) {
30     OverridingPrivate2 op2 = new OverridingPrivate2();
31     op2.f();
32     op2.g();
33     // You can upcast:
34     OverridingPrivate op = op2;
35     // But you can't call the methods:
36     //! op.f();
37     //! op.g();
38     // Same here:
39     WithFinals wf = op2;
40     //! wf.f();
41     //! wf.g();
42   }
43 }
```

final classes

- When you say that an entire class is *final*, you state that you don't want to inherit from this class or allow anyone else to do so
  - There is never a need to make any changes, or for safety or security reasons you don't want subclassing

```
1  class SmallBrain {}
2
3  final class Dinosaur {
4    int i = 7;
5    int j = 1;
6    SmallBrain x = new SmallBrain();
7    void f() {}
8  }
9
10 //! class Further extends Dinosaur {}
11 // error: Cannot extend final class 'Dinosaur'
12
13 public class Jurassic {
14   public static void main(String[] args) {
15     Dinosaur n = new Dinosaur();
16     n.f();
17     n.i = 40;
18     n.j++;
19   }
20 }
```

Initialization and class loading

- In more traditional languages, programs are loaded all at once as part of the startup process
- This is followed by initialization, and then the program begins
- The process of initialization in these languages must be carefully controlled
  - The order of initialization of *static*s does cause trouble, e.g., in C++
- Java doesn't have this problem because it takes a different approach to loading
  - "class code is loaded at the point of first use."
  - This is usually when the first object of that class is constructed, but loading also occurs when a *static* field or *static* method is accessed

Initialization with inheritance

- It's helpful to look at the whole initialization process, including inheritance, to get a full picture of what happens

```
1  import static net.mindview.util.Print.*;
2
3  class Insect {
4    private int i = 9;
5    protected int j;
6    Insect() {
7      print("i = " + i + ", j = " + j);
8      j = 39;
9    }
10   private static int x1 =
11     printInit("static Insect.x1 initialized");
12   static int printInit(String s) {
13     print(s);
14     return 47;
15   }
16 }
17
18 public class Beetle extends Insect {
19   private int k = printInit("Beetle.k initialized");
20   public Beetle() {
21     print("k = " + k);
22     print("j = " + j);
23   }
24   private static int x2 =
25     printInit("static Beetle.x2 initialized");
26   public static void main(String[] args) {
27     print("Beetle constructor");
28     Beetle b = new Beetle();
29   }
30 }
```

谢谢

zhenling@seu.edu.cn