J.28

SUBCLASSES and SUPERCLASSES

The Object class is the root of the Java class hierarchy.

The Object class declares methods that are implemented by all objects.

- public boolean equals(Object obj)
- public int hashCode()
- protected Object clone()
- public final Class getClass()
- protected void finalize() throws Throwable

Variables of type Object can refer to any object and can be used in lists.

J.29

Dessert Example

```
public class Dessert {
    private String name;
    private int calories;

  public Dessert(String nameOf) {
    name = nameOf;
  }

  public Dessert(String nameOf, int caloriesOf) {
    name = nameOf;
    calories = caloriesOf;
  }

  public String nameOf() { return name; }

  public int caloriesOf() { return calories; }
  }
```

J.30

```
public class Cake extends Dessert {
    private String cakeType;
    private String frostType;

  public Cake(String cakeNameOf, int cakeCalOf,
           String cakeTypeof, String frostTypeOf) {

    super(cakeNameOf, cakeCalOf);
    cakeType = cakeTypeof;
    frostType = frostTypeOf;
    }

  public Cake(String cakeNameOf, String cakeTypeOf) {
    super(cakeNameOf,0);
    cakeType = cakeTypeOf;
    frostType = "none";
    }

  public String cakeTypeOf() { return cakeType; }

  public String frostTypeOf() { return frostType; }


  }
```

J.31

Testing Desserts and Cakes

```
public class DessertTest {

 public static void main (String[] args) {

    Dessert general = new Dessert("General",500);
    System.out.println(general.nameOf() + ": " +
                          general.caloriesOf());

    Cake acake = new Cake("Mike's Cake",800,
                              "German Chocolate","Chocolate");
    System.out.println(acake.nameOf() + ": " + acake.cakeTypeOf()
      + " " + acake.caloriesOf() + " " + acake.frostTypeOf());

    Cake bcake = new Cake("Darrell's Cake","Carrot");
    System.out.println(bcake.nameOf() + ": " + bcake.cakeTypeOf()
      + " " + bcake.caloriesOf() + " " + bcake.frostTypeOf());

  }
}
```

General: 500
Mike's Cake: German Chocolate 800 Chocolate
Darrell's Cake: Carrot 0 none

J.32

Can a private field of the parent class be changed by a subclass?

```
public class Cake extends Dessert {
    private String cakeType;
    private String frostType;


  public Cake(String cakeNameOf) {
    super(cakeNameOf);
    calories = 250;
    ^
    }

  public String cakeTypeOf() { return cakeType; }

  public String frostTypeOf() { return frostType; }


  }
```

Variable calories in class Dessert not accessible from class Cake.

J.33

Protected Access Solves This Problem

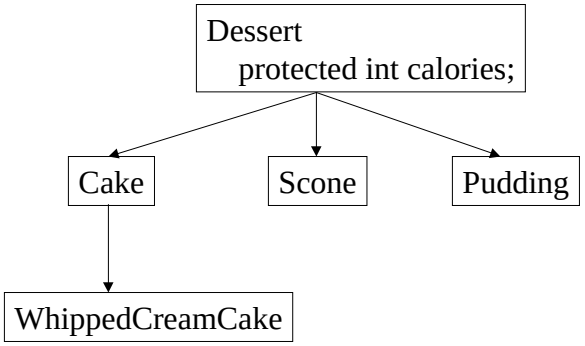
```
public class Dessert {
    private String name;
    protected int calories;

  public Dessert(String nameOf) {
    name = nameOf; }

  public Dessert(String nameOf, int caloriesOf) {
    name = nameOf;
    calories = caloriesOf; }

  public String nameOf() { return name; }

  public int caloriesOf() { return calories; }
  }
```

```
public class DessertTest2 {

 public static void main (String[] args) {

   Cake ccake = new Cake("Linda's Cake");

   System.out.println(ccake.nameOf() + ": " + ccake.cakeTypeOf()
     + " " + ccake.caloriesOf() + " " + ccake.frostTypeOf());

 }}
```

What will be printed?

J.34

What protected Really Means



Each class that extends Dessert inherits calories.

Code in the Cake class can access calories only through a reference to a type that is a Cake or a subclass of Cake.

Code in the Cake class cannot access the calories field of a Scone or a Pudding or a generic Dessert.

Note: protected static fields can be accessed in any extended class.

J.35

Overloading vs. Overriding

- Overloading a method means providing more than one method with the same name, but different signatures.

- Overriding a method means replacing the superclass's implementation of a method with one of your own, with an identical signature.

```
Public class Cake extends Dessert {
    •
    •
    •
public String NameOf() {
return "Cake  " + name;

}
```

J.36

Fields can be hidden, but not overridden

```
class SuperShow {
  public String str = "SuperStr";

  public void show() {
  System.out.println("Super.show: " + str);  }  }

class ExtendShow extends SuperShow {
  public String str = "ExtendStr";

  public void show() {
  System.out.println("Extend.show: " + str); }

  public static void main(String[] args) {
    ExtendShow ext = new ExtendShow();
    SuperShow sup = ext;
    sup.show();
    ext.show();
    System.out.println(" sup.str = " + sup.str);
    System.out.println(" ext.str = " + ext.str);  }}
```

What will the output be ?

J.37

More Terminology

- Final Classes and Methods:

A method declared as final cannot be overriden by any extended classes of its class.

A class marked final cannot be subclassed by any other class, and its methods are implicitely final, too.

What are final classes and methods needed for ?

J.38

- Abstract Classes and Methods:

A method declared as abstract needs only a signature. The definition is left for implementation in extended classes.

A class marked abstract is one that has at least one abstract method.

In what kinds of applications are abstract classes and methods useful?

J.39

INTERFACES

Interfaces provide a way to declare a type consisting only of abstract methods and constants, enabling any implementation to be written for those methods.

"An interface is an expression of pure design, where a class is a mix of design and implementation."

```
Interface Attributed {
   void add(String attrName, Attr newAttr);
   Attr find(String attrName);
   Attr remove(String attrName);
}
```

J.40

```
public class AVListNode {
   String Attribute;
   Attr Value;
   AVListNode Next;
   . . .   }
```

```
public class AVTable implements Attributed {
   private AVListNode  Head;

  public void find(String attrName) {
     AVListNode  Marker;

    Marker = Head;
    while (Marker != null &&
            Marker.Attribute != attrName)
      Marker = Marker.Next;

    if (Marker != null)  return Marker.Value;
   else return null;
   }   . . .
}
```

J.41

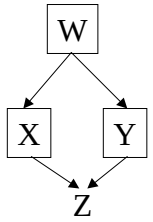
More About Interfaces

A class definition can both extend a class and implement one or more interfaces.

```
Class AttributedBody extends Body
        implements  Attributed {
. . .
}
```

Interfaces add multiple inheritance to Java.

```
interface W { }
interface X extends W { … }
interface Y extends W { . . . }
class Z implements X, Y { . . . }
```



J.42

Name Conflicts

What happens when a method of the same name appears in more than one interface, ie. in both X and Y?

- If methods in X and Y have the same name but different signatures, the Z class will have two overloaded methods.
- If the methods in X and Y have exactly the same signature, the Z class will have one method with that signature.
- If the signatures differ only in return type, you cannot implement both X and Y.
- If the two methods differ only in the types of exceptions they throw, there must be only one implementation that satisfies both throws clauses.