Exercise

/* A lockbox can be open or closed. If closed, only a valid password will open the box. Once the box is open, the contents can be retrieved.
*/

```
interface LockBox {
   boolean openBox(String password); // attempt to open, return true if successful
   boolean isOpen(); // check if box is open
   Object getStuff(); // get contents if open, or null if closed
   void destroy (); // discard contents
}
```

→Implement a LockBox.

→Implement it again, but this time print a warning if the password is wrong. Only warn the user once for each lock box.

/* A volatile lock box self-destructs after 3 invalid password attempts. That is, the contents should be destroyed. The user can also ask how many attempts are left. */

→Design the interface, and then implement it.

Moral

- We have encapsulation:
  - Data & Methods together
  - Data & Implementation hidden – details don’t matter to user
- But we still have a lot copy-and-paste
  - Implementations of lockbox and volatile lock box nearly identical
- Also cannot change/extend behavior without having the source code, or rewriting from scratch.
  - Calling Microsoft to add a new method is not an option
  - Asking for source code is less of an option
  - we could rewrite is from scratch

OO-Programming Again

- OO-Programming = Encapsulation + Extensibility
- Encapsulation hides details of class implementation
- Extensibility: want to modify (extend, change) behavior of classes
  - Without lots of code duplication
  - Without involving class implementer
  - Without even having the class source code

Wrappers / Adapters

- Have a LockBox class, but want VolatileLockBox
- Make a new class to wrap the old one:
  public class VBox implements VolatileLockBox { … }
- Adapter:
  - Takes object of one type, adapts it to a new type
  - e.g., BufferedReader same as a Reader, but with a readLine( ) method. All other functions defer to the Reader that was passed to the constructor.
- Good for certain purposes, but still clumsy:
  - A lot of drudge work writing “wrapper” methods

Inheritance

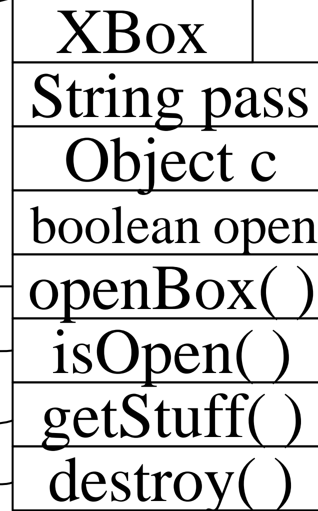
Please read for details → Weiss ch. 4

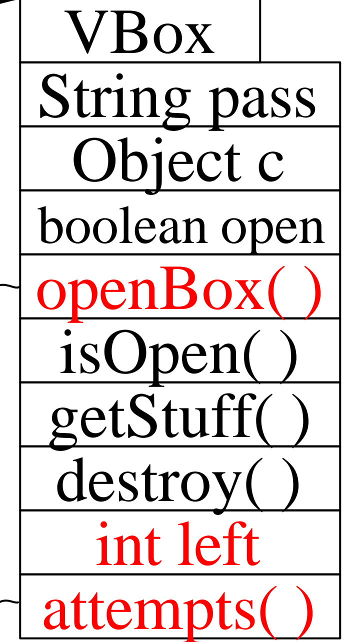
Goal

- Given class A from your supplier, want to create a new class B that extends A in some way.
  - Changes behavior of some existing methods
  - Inherits all the remaining methods
  - Creates some additional methods, fields, etc.
  - Implements same interfaces, and maybe some new ones
  - And can be used wherever A can be used
    - e.g., any code that accepted an A should also accept a B

Sub-classes

```
class XBox implements LockBox {
  private String pass;
  private Object c;
  private boolean open;

  public XBox(Object o, String p) {
    pass = p; c = o;
  }

  public boolean openBox(String guess) {
    if (open) return true;
    else return (open = pass.equals(guess));
  }

  public boolean isOpen() {
    return open;
  }

  public Object getStuff() {
    return open ? c : null;
  }

  public void destroy() {
    c = null;
  }
}
```






```
class VBox extends XBox
implements VolatileLockBox {
  ???
}
```

Some Terminology

- Sub-class VBox:
  - inherits instance variables *pass*, *c*, and *open* from super-class XBox
  - inherits instance methods *isOpen*( ), *getStuff*( ), and *destroy*( ) from super-class XBox
  - has its own instance variable *tries* and instance method *attempts*
  - overrides instance method *openBox*( ) in super-class XBox
- Overriding:
  - must have same signature: return type, name, parameters, static or not
    - BUT may be less restricted in access: private can become public

Writing a Sub-class

```
class XBox implements LockBox {
  private String pass;
  private Object c;
  private boolean open;
  public XBox(Object o, String p) {
    pass = p; c = o;
  }
  public boolean openBox(String guess) {
    if (open) return true;
    else return (open = pass.equals(guess));
  }
  public boolean isOpen() {
    return open;
  }
  public Object getStuff() {
    return open ? c : null;
  }
  public void destroy() {
    c = null;
  }
}
```

```
class VBox extends XBox
implements VolatileLockBox {
  private int left;
  public VBox(Object o, String p) {
    pass = p; c = o;
    left = 3;
  }
  public boolean openBox(String guess) {
    if (left = = 0) {
      destroy();
      return false;
    } else {
      boolean good;
      if (open) good = true;
      else good = (open = pass.equals(guess));
      if (!good) tries--;
      return good;
    }
  public int attempts() {
    return left;
  }
}
```

Super-class Privates Inaccessible

- Problem:
  - Fields of super-class (XBox) might be private
  - But sub-class (VBox) needs to manipulate them…
  - Same goes for private helper methods in super-class
- Confused?
  - Sub-class VBox *has* the private fields and methods (it inherits them)
  - But only the code defined in XBox can use them
- Not a solution:
  - Make everything public instead

One Solution

- Make everything protected instead
- private = accessible only to the class that declares it
- public = accessible to any class at all
- protected = accessible to the class and its sub-classes
- default = accessible to classes in same “package”

Using Protected

```
class XBox implements LockBox {
  protected String pass;
  protected Object c;
  protected boolean open;

  public XBox(Object o, String p) {
    pass = p; c = o;
  }

  public boolean openBox(String guess) {
    if (open) return true;
    else return (open = pass.equals(guess));
  }

  public boolean isOpen() {
    return open;
  }

  public Object getStuff() {
    return open ? c : null;
  }

  public void destroy() {
    c = null;
  }
}
```

```
class VBox extends XBox implements VolatileLockBox {
  protected int left;

  public VBox(Object o, String p) {
    pass = p; c = o;
    left = 3;
  }

  public boolean openBox(String guess) {
    if (left = = 0) {
      destroy();
      return false;
    } else {
      boolean good;
      if (open) good = true;
      else good = (open = pass.equals(guess));
      if (!good) tries--;
      return good;
    }
  }

  public int attempts() {
    return left;
  }
}
```

Critique

- Use protected instead of private?
  - Decide: will a sub-class ever need to access directly?
  - But: Exposes some of the "internal state" to the outside world – e.g., to other programmers, hackers, etc.
    - What if they mess up? What if they are evil?
- Analogy: Car Mods & User-Serviceable Parts
  - Car has "public" components – paint color, steering wheel, blinkers, radio, etc. Anyone can access these (by design).
  - What of the rest?
    spark plugs ?
    muffler ?
    ECC (engine control computer)?
    shocks?
    airbags?

Another Solution

```
class XBox implements LockBox {
  private String pass;
  private Object c;
  private boolean open;
  public XBox(Object o, String p) {
    pass = p; c = o;
  }
  public boolean openBox(String guess) {
    if (open) return true;
    else return (open = pass.equals(guess));
  }
  public boolean isOpen() {
    return open;
  }
  public Object getStuff() {
    return open ? c : null;
  }
  public void destroy() {
    c = null;
  }
}
```

```
class VBox extends XBox implements VolatileLockBox {
  private int left;
  public VBox(Object o, String p) {
    pass = p; c = o;
    left = 3;
  }
  public boolean openBox(String guess) {
    if (left = = 0) {
      destroy();
      return false;
    } else {
      boolean good;
      if (open) good = true;
      else good = (open = pass.equals(guess));
      if (!good) tries--;
      return good;
    }
  }
  public int attempts() {
    return left;
  }
}
```

Keyword super

- In a sub-class, keyword super gives access to members of the super-class that were not inherited

super.m(…) for methods

super(…) for constructors

```
class XBox implements LockBox {
  private String pass;
  private Object c;
  private boolean open;

  public XBox(Object o, String p) {
    pass = p; c = o;
  }

  public boolean openBox(String guess) {
    if (open) return true;
    else return (open = pass.equals(guess));
  }

  …
}
```

```
class VBox extends XBox implements VolatileLockBox {
  private int left;

  public VBox(Object o, String p) {
    super(o, p);
    left = 3;
  }

  public boolean openBox(String guess) {
    if (left = = 0) {
      destroy();
      return false;
    } else {
      boolean good = super.openBox(guess);
      if (!good) tries--;
      return good;
    }
  }

  …
}
```

Keyword super

- Analogous to keyword this
- Cannot compose – only allowed access to direct super-class:

```
super.super.m(…); // not allowed
```

- Uses *static binding*:
  - Compiler binds "super.openBox" in VBox directly to XBox.openBox. No dynamic lookup done at run-time.

Inheritance as Sub-types

- Every VBox instance
  - “is-a” VolatileLockBox (b/c VBox implements it directly)
  - “is-a” LockBox (b/c VolatileLockBox is sub-type of LockBox)

  → “is-a” XBox (b/c VBox extends it directly, and so has all the needed fields, methods, etc.)

- Type Checking?

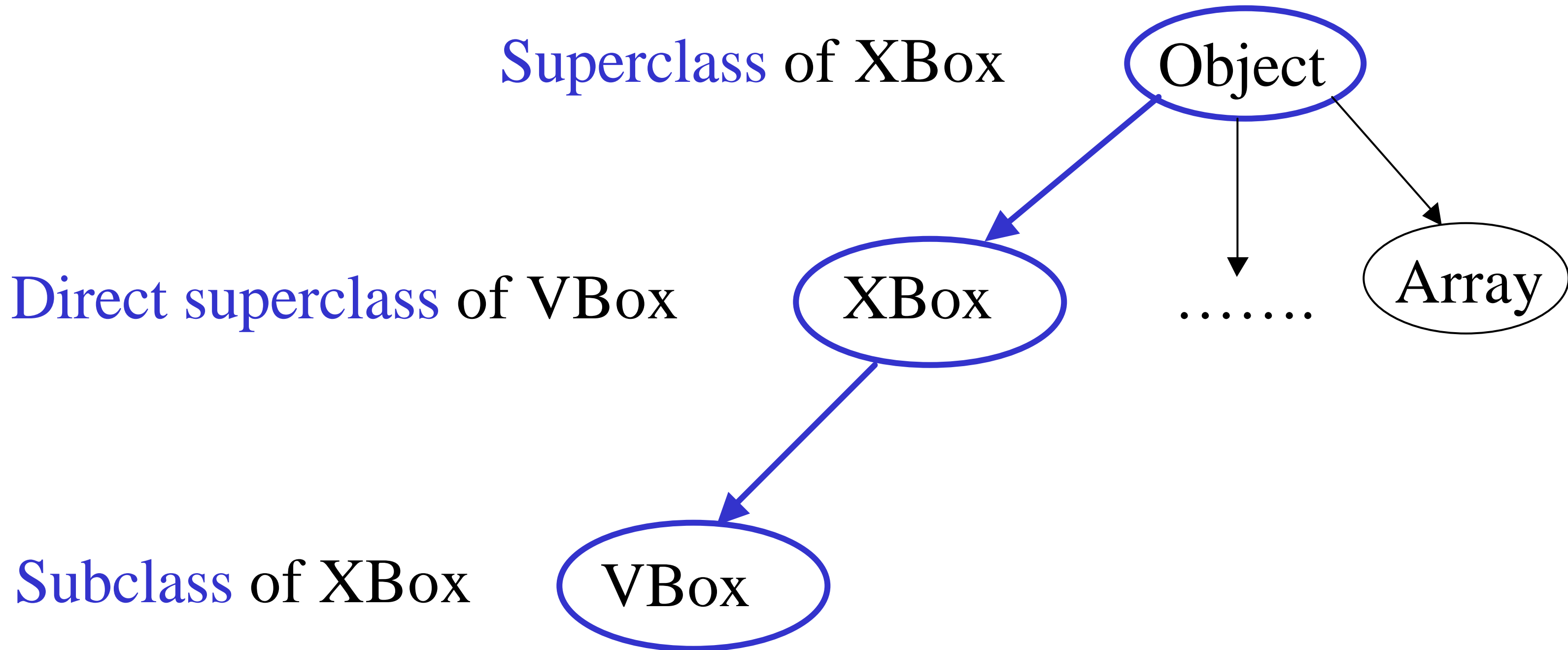
```
LockBox b = new VBox(“47”, “pw”);  // ?
VolatileLockBox v = b; // ?
XBox x = b; // ?
XBox x2 = new VBox(“47”, “guess-me”); // ?
```

Access Restrictions vs. Overriding

XBox x2 = new VBox("47", "guess-me"); // OK

- VBox is a sub-type of XBox
- So: VBox must do anything an XBox can do
  - Can VBox.openBox( ) be private or protected?
- Overriding (again):
  - Must have same signature: return type, name, parameters, static or not
  - BUT may be less restricted in access: private in XBox can become public in VBox, but not the reverse.
  - Compiler enforces this rule.

Class Hierarchy

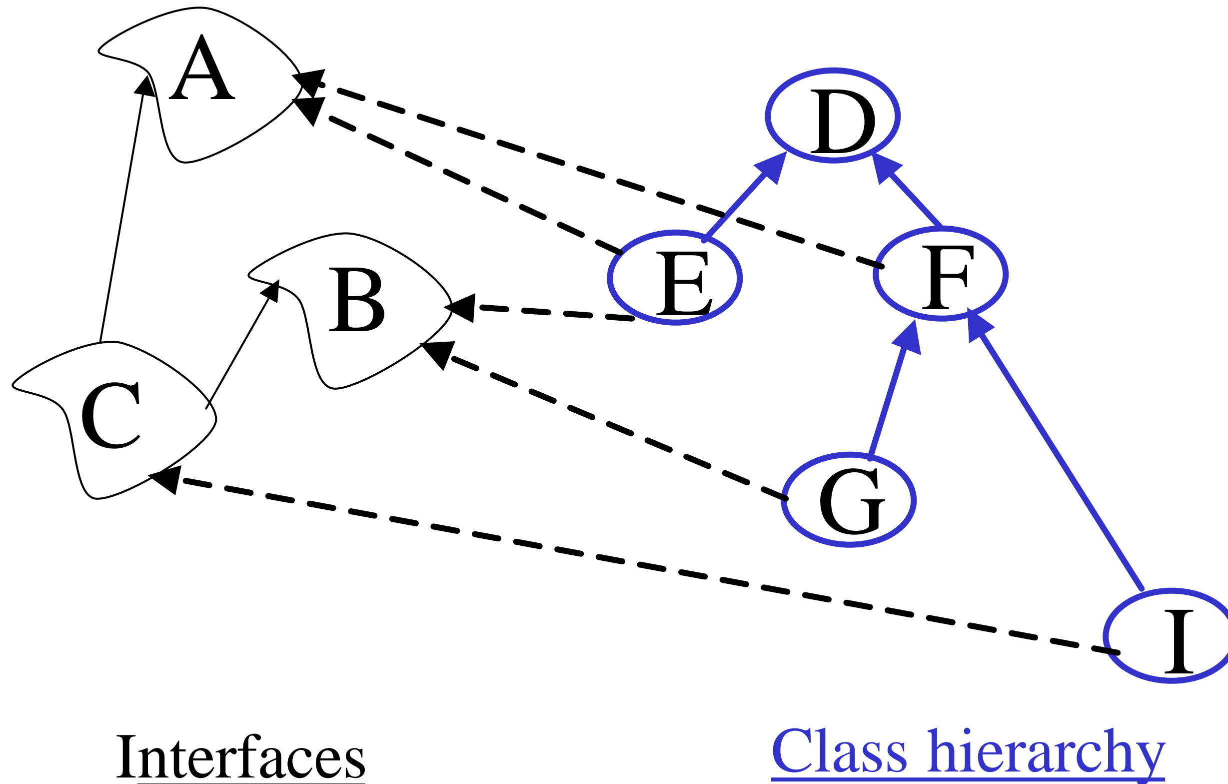


- If no super-class specified, uses java.lang.Object by default
- All classes derive (eventually) from Object
- Object has no parent class (super-class)

Java Uses Single Inheritance

- Each class picks ONE direct super-class, or uses the default Object as direct super-class
- Class hierarchy is a tree
- Each class can implement many interfaces
- C++ allows *multiple inheritance* also:
  - very complicated in practice
  - Java's single inheritance + multiple interfaces get most of the benefits, with far fewer complications

Single Inheritance + Multiple Interfaces




```
interface C extends A,B{
  …
}
```

```
class F extends D implements A{
…..}
class E extends D implements A,B{
….}
```

Details, Details

- **Instance methods** are inherited by sub-class
  - public ones can be overridden by sub-class
  - dynamic “run-time” lookup is used to find which class’ implementation of the method to use

    (We don’t call them “dynamic methods” for nothing)
  - sub-class can still get to old method using super.m( ), but no one else can.

Details, Details

- **Static "class" methods** are "inherited" by sub-class
- Or not (depending on who you talk to)
- They seem like they are inherited:

```
class XBox { public static int id( ) { return 1; } }
VBox.id( ); // OK: resolves to XBox.id( )
VBox v = null; v.id( ); // OK: same thing
XBox x = v; x.id( ); // OK: same thing
```

- But then again:

```
class VBox { public static int id( ) { return 2; } }
VBox.id( ); // OK: but now resolves to VBox.id( )
VBox v = null; v.id( ); // OK: now resolves to VBox.id()
XBox x = v; x.id(); // OK: still resolves to XBox.id()
```

Inheriting Static Methods?

- We don’t call them “static” for nothing
  - Compiler binds all static method calls at compile-time
  - It will “walk up the class tree” to find the method, but you could do that yourself.
  - It will allow “x.id()”, but this is just shorthand for “XBox.id()”, regardless of actual type of x.
- Java calls this “hiding” instead of “overriding”.

Constructors

- Constructors are **not** inherited; so such thing as "constructor overriding"
  - But: a sub-class constructor can *defer* to a super-class constructor using keyword super on first line
- Object initialization can get tricky in sub-classes

Variables

- Java uses *static binding* for all variables
  - even “instance” variables?

```
class A {
   static String s = "A.s";
   String i = “a.i";
}

class B extends A {
   static String s = "B.s";
   String i = “b.i";
}

class C extends B {
   static String s = "C.s";
   String i = “c.i";
}
```

```
class Confused {
   public static void main(String args[]) {
      System.out.println("class.static");
      System.out.println(A.s);
      System.out.println(B.s);
      System.out.println(C.s);

      System.out.println(A.i);
      System.out.println("class.instance");
      System.out.println(B.i);
      System.out.println(C.i);

      System.out.println("null.static");
      A a = null; B b = null; C c = null;
      System.out.println(a.s);
      System.out.println(b.s);
      System.out.println(c.s);

      System.out.println("Object-of-type-C.instance");
      a = b = c = new C();
      System.out.println(a.i);
      System.out.println(b.i);
      System.out.println(c.i);
   }
}
```

Shadowing Variables

- static variables:
  - can't override
  - three copies stored: A.s, B.s, C.s
  - compiler decides which one at compile-time
- instance variables:
  - can't override – instead, we "shadow" super-class vars
    - similar to "hiding" of static methods
    - can use *super*.*field* from B or C
    - don't even have to be the same type!
  - three copies stored ***in every object***
  - compiler decides which one at compile-time