University of Tokyo Java Class
September 22-26, 2003
Common Java Programming Bugs

Marc Hamilton

Director of Technology

Global Education and Research

Sun Microsystems, Inc

Overall Presentation Goal

Point out some common problems in Java™ technology design and coding

Especially the kind that show up late in a development process

Surprising Java™ Technology Code

- Is valid Java™ technology code
- Has run-time bugs
- Has future problems extending classes
- Uses Bug patterns

Today's Surprises Will Cover

10 Code Examples

- What does it do?
- Corrections
- Some examples are advanced

Lessons to learn

- Practices to avoid
- Practices to follow

References

- Web sites
- Published books

1. Watch Those Nulls

In this first example, we write code defensively in case a reference is null…

1. Watch Those Nulls

```
1: class Fun {
2:   String sVal = "Whee!";

3:   public static void main(String[ ] args) {
4:     Fun myFun = null;
5:           // . . .
6:     if (myFun!=null & myFun.sVal.length( )>0)
7:       System.out.println(myFun.sVal);
8:   }
9: }
```

1. Prints:

a) Whee!
b) Nothing
c) NullPointerException
d) Something else

```
1: class Fun {
2:   String sVal = "Whee!";

3:   public static void main(String[ ] args) {
4:     Fun myFun = null;
5:         // . . .
6:     if (myFun!=null & myFun.sVal.length( )>0)
7:       System.out.println(myFun.sVal);
8:   }
9: }
```

1. Prints:

a) Whee!

b) Nothing

c) NullPointerException

d) Something else

```
1: class Fun {
2:   String sVal = "Whee!";

3:   public static void main(String[ ] args) {
4:     Fun myFun = null;
5:         // . . .
6:     if (myFun!=null & myFun.sVal.length( )>0)
7:       System.out.println(myFun.sVal);
8:   }
9: }
```

1. What's Wrong?

- The null pointer test failed:

```
if (myFun!=null & myFun.sVal.length( )>0)
```

  Why?

- Since both expressions are boolean, it is a logical AND, not a bitwise AND
- But & doesn't short-circuit: && does

1. Watch Those Nulls: Fixed

```
1: class Fun {
2:   String sVal = "Whee!";

3:   public static void main(String[ ] args) {
4:     Fun myFun = null;
5:        // . . .
6:     if (myFun!=null && myFun.sVal.length( )>0)
7:       System.out.println(myFun.sVal);
8:   }
9: }
```

1. Lessons on Nulls

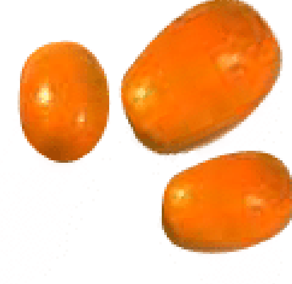
Use Logical operators correctly

Other Null tips:

- Assign all fields in constructors
  - avoids run-on constructors
  - avoids need for null checks
- Avoid returning nulls from methods
  - throw an exception at point of the problem
  - return empty object, such as: new array[0]
  - avoids need for null checks

2. What's for Lunch?

In this next example, a simple allocation of an object doesn't go as planned…



2. What’s for Lunch?

```
 1: public class Lunch {
 2:    int kumquats;
 3:    public void Lunch( ) {
 4:       kumquats = 5;
 5:    }
 6: }
 7: public static void main(String[ ] args) {
 8:    Lunch x = new Lunch( );
 9:    System.out.print( x.kumquats+" kumquats");
10: }
```

2. Prints:

- a) 0 kumquats
- b) 5 kumquats
- c) NullPointerException
- d) Something else

```
 1: public class Lunch {
 2:    int kumquats;
 3:    public void Lunch( ) {
 4:      kumquats = 5;
 5:    }
 6: }
 7: public static void main(String[ ] args) {
 8:    Lunch x = new Lunch( );
 9:    System.out.print( x.kumquats+" kumquats");
10: }
```

2. Prints:

a) <u>0 kumquats</u>

b) 5 kumquats

c) NullPointerException

d) Something else

```
 1: public class Lunch {
 2:    int kumquats;
 3:    public void Lunch( ) {
 4:       kumquats = 5;
 5:    }
 6: }
 7: public static void main(String[ ] args) {
 8:    Lunch x = new Lunch( );
 9:    System.out.print( x.kumquats+" kumquats");
10: }
```

2. What's Wrong?

Our constructor didn't fire

```
3:    public void Lunch( ) {
4:      kumquats = 5;
5:    }
```

Why?

It's not really a constructor

2. What's for Lunch: Fixed

```
 1: public class Lunch {
 2:    int kumquats;
 3:    public Lunch( ) {
 4:       kumquats = 5;
 5:    }
 6: }
 7: public static void main(String[ ] args) {
 8:    Lunch x = new Lunch( );
 9:    System.out.print( x.kumquats+" kumquats");
10: }
```

2. Lessons on Method Types

Follow naming conventions

- Capital letter for Class/Constructor names
- Small letter for Method names
- Helps multiple developers in single class

Some IDEs point this problem out

3. Catching Errors

Catching Java Errors can be tricky

3. Catching Errors

```
1: class MyError extends Error {
2:   MyError( String s ) { super(s); }
3: }
4: public class MyClass {
5:   MyClass( ) {
6:      throw new MyError("X"); }
7:   public static void main(String[ ] args) {
8:      try { MyClass m = new MyClass( );
9:            System.out.print("MyClass is OK"); }
10:     catch (Exception e) {
11:           System.out.print("Caught you!");}
12: } }
```

3. Prints:

a) MyClass is OK
b) Caught you!
c) Won't compile
d) Something else

```
1: class MyError extends Error {
2:    MyError( String s ) { super(s); }
3: }
4: public class MyClass {
5:    MyClass( ) {
6:       throw new MyError("X"); }
7:    public static void main(String[ ] args) {
8:       try { MyClass m = new MyClass( );
9:             System.out.print("MyClass is OK"); }
10:      catch (Exception e) {
11:            System.out.print("Caught you!");}
12: } }
```

3. Prints:

a) MyClass is OK

b) Caught you!

c) Won't compile

d) Something else

```
1:  class MyError extends Error {
2:    MyError( String s ) { super(s); }
3:  }
4:  public class MyClass {
5:    MyClass( ) {
6:      throw new MyError("X"); }
7:    public static void main(String[ ] args) {
8:      try { MyClass m = new MyClass( );
9:            System.out.print("MyClass is OK"); }
10:     catch (Exception e) {
11:           System.out.print("Caught you!");}
12: } }
```

3. What's Wrong?

- Exception is not caught because it's of type Error instead of Exception

```
1: class MyError extends Error {
2:    MyError( String s ) { super(s); }
3: }
```

- It DOES compile, even without "throws" clause

```
4: class MyClass {
5:    MyClass( ) throws MyError {
6:       throw new MyError("X"); }
```

- Errors are meant to be ignored (not caught)

3. Catching Errors: Fixed

```
 1: class MyError extends Exception {
 2:    MyError( String s ) { super(s); }
 3: }
 4: public class MyClass {
 5:    MyClass( ) throws MyError {
 6:       throw new MyError("X"); }
 7:    public static void main(String[ ] args) {
 8:       try { MyClass m = new MyClass( );
 9:             System.out.print("MyClass is OK"); }
10:       catch (Exception e) {
11:             System.out.print("Caught you!");}
12: } }
```

3. Lessons in Errors

Extend Exception, not Error

- Compiler warns you about checked exceptions
- Extend RuntimeException to avoid checks
- In general, ignore Error—you can't normally recover from it
- Don't extend Throwable

Consider: Do you need to wrap the error?

If you must wrap an error

- Include context information in the message
- Include the original exception

3A. Wrapping Exceptions

```
1: class MyException extends Exception {
2:    MyException( ) { super( ); }
3: }
4: class MyBlackBox {
5:    MyBlackBox( ) throws MyException {
6:       String a[ ] = new String[1];
7:       try
8:          { a[-1] = "X"; /* more lines of code */ }
9:       catch (Exception e)
10:         { throw new MyException( ); }
11:} }
```

3A. Shows:

MyException
  at MyBlackBox.<init>(...:10)
  at . . .

```
1: class MyException extends Exception {
2:    MyException( ) { super( ); }
3: }
4: class MyBlackBox {
5:    MyBlackBox( ) throws MyException {
6:       String a[ ] = new String[1];
7:       try
8:          { a[-1] = "X"; /* more
9:       catch (Exception e)
10:         { throw new MyException( ); }
11:} }
```

3A. Problems

There's no indication of the real problem (or error location) for the programmer

There's no helpful text for user or programmer

Let's look at an improvement

3A. Improved Wrapper

```
1: class MyException extends Exception {
2:   MyException(String s, Throwable e) {super(s,e);}
3: }
4: class MyBlackBox {
5:   MyBlackBox( ) throws MyException {
6:     String a[ ] = new String[1];
7:     try
8:       { a[-1] = "X"; /* many more lines of code */ }
9:     catch (Exception e)
10:      { throw new MyException("Ouch", e); }
11:} }
```

3A. Shows:

MyException: Ouch

at MyBlackBox.<init>(...:10)

at . . .

Caused by:

java.lang.ArrayIndexOutOfBoundsException

at MyBlackBox.<init>(...:8) Original Error!

at . . .

```
1: class MyException exte
2:    MyException(String s
3: }
4: class MyBlackBox {
5:    MyBlackBox( ) throws MyException {
6:       String a[ ] = new String[1];
7:       try
8:         { a[-1] = "X"; /* many more lines of code */ }
9:       catch (Exception e)
10:        { throw new MyException("Ouch", e); }
11:} }
```

3A. Improved Wrapper

New methods in 1.4 Throwable (Exception) class:

```
Exception(String message, Throwable cause)
initCause(Throwable cause)
```

You can simulate this for pre-JDK 1.4 environments with ChainedException class

See Brian Goetz's article to download the ChainedException code (for pre-JDK 1.4):

http://developer.java.sun.com/developer/technicalArticles/Programming/exceptions2

4. Roll Call

Our next example uses Enumeration to visit all entries in an Object, such as a Vector

What dangers lurk here?

4. Roll Call

```
 1: Vector v = new Vector( );
 2: v.add("A"); v.add("B"); v.add("C");
 3: Enumeration enum = v.elements( );
 4: while (enum.hasMoreElements( )) {
 5:   String s = (String) enum.nextElement( );
 6:   if (s.equals("A")) {
 7:     System.out.println("remove: A");
 8:     v.remove("A");
 9:   } else {
10:     System.out.println("visit: " + s);
11:   }
12: }
```

4. Prints:

```
 1: Vector v = new Vector( );
 2: v.add("A"); v.add("B"); v.add("C"
 3: Enumeration enum = v.elements( );
 4: while (enum.hasMoreElements( )) {
 5:   String s = (String) enum.nextElement( );
 6:   if (s.equals("A")) {
 7:     System.out.println("remove: A");
 8:     v.remove("A");
 9:   } else {
10:     System.out.println("visit: " + s);
11:   }
12: }
```

a) remove: A
visit: B
visit: C

b) remove: A
Exception

c) Exception

d) something else

4. Prints:

a) remove: A
visit: B
visit: C

b) remove: A
Exception

c) Exception

d) something else
remove: A
visit: C

```
1: Vector v = new Vector( );
2: v.add("A"); v.add("B"); v.add("C"
3: Enumeration enum = v.elements( );
4: while (enum.hasMoreElements( )) {
5:    String s = (String) enum.nextElement( );
6:    if (s.equals("A")) {
7:       System.out.println("remove: A");
8:       v.remove("A");
9:    } else {
10:      System.out.println("visit: " + s);
11:   }
12: }
```

4. What's Wrong?

Enumeration and Vector are fighting over the same data structure

```
4: while (enum.hasMoreElements( )) {
5:   String s = (String)enum.nextElement( );
6:   if (s.equals("A")) {
7:     System.out.println("remove: A");
8:     v.remove("A");
```

4. Roll Call: Fixed

```
 1: Vector v = new Vector( );
 2: v.add("A"); v.add("B"); v.add("C");
 3: Iterator iter = v.iterator( );
 4: while (iter.hasNext( )) {
 5:   String s = (String) iter.next( );
 6:   if (s.equals("A")) {
 7:     System.out.println("remove: A");
 8:     iter.remove("A");
 9:   } else {
10:     System.out.println("visit: " + s);
11:   }
12: }
```

4. Lessons on Iterators

Use Iterators instead of Enumeration when possible (available since 1.2)

Instead of this:	Use this:
e.hasMoreElements( )	i.hasNext( )
e.nextElement( )	i.next( )
collection.remove( )	i.remove( )

5. Where Does it Go?

Hashtables are great for storing things

As long as we don't lose the key…

5. Where Does it Go?

```
 1: Hashtable hash = new Hashtable( );
 2: StringBuffer key = new StringBuffer("S");
 3: hash.put(key, "Salt" );
 4: key.replace(0, key.length( ), "P");
 5: hash.put(key, "Pepper" );
 6:
 7: Iterator iter = hash.keySet( ).iterator( );
 8: while (iter.hasNext( )) {
 9:    StringBuffer sb=(StringBuffer)iter.next( );
10:    System.out.println(sb+" - "+hash.get(sb));
11: }
```

5. Prints:

a) S – Salt
P – Pepper

b) P – Pepper

c) P – Pepper
P – Pepper

```
1: Hashtable hash = new Hashtable( );
2: StringBuffer key = new StringBuffer("S");
3: hash.put(key, "Salt" );
4: key.replace(0, key.length( ), "P");
5: hash.put(key, "Pepper" );
6:
7: Iterator iter = hash.keySet( ).iterator( );
8: while (iter.hasNext( )) {
9:    StringBuffer sb=(StringBuffer)iter.next( );
10:   System.out.println(sb+" - "+hash.get(sb));
11: }
```

5. Prints:

a) S – Salt
P – Pepper

b) P – Pepper

c) P – Pepper
P – Pepper

```
 1: Hashtable hash = new Hashtable( );
 2: StringBuffer key = new StringBuffer("S");
 3: hash.put(key, "Salt" );
 4: key.replace(0, key.length( ), "P");
 5: hash.put(key, "Pepper" );
 6:
 7: Iterator iter = hash.keySet( ).iterator( );
 8: while (iter.hasNext( )) {
 9:    StringBuffer sb=(StringBuffer)iter.next( );
10:    System.out.println(sb+" - "+hash.get(sb));
11: }
```

5. What's Wrong?

Hashtables call your key's equals( )

StringBuffer doesn't implement equals( )

- So it uses Object.equals( )
- And both put( ) operations think they have the same key

So the second put( ) overwrites the first

5. Where Does it Go: Fixed

```
 1: Hashtable hash = new Hashtable( );
 2: StringBuffer key = new StringBuffer("S");
 3: hash.put(key.toString( ), "Salt" );
 4: key.replace(0, key.length( ), "P");
 5: hash.put(key.toString( ), "Pepper" );
 6:
 7: Iterator iter = hash.keySet( ).iterator( );
 8: while (iter.hasNext( )) {
 9:   String sb = (String)iter.next( );
10:   System.out.println(sb+" - "+hash.get(sb));
11: }
```

5. Lessons in Hashing

Hash lists are optimized—they store:

- References to keys and values
- HashCodes for keys

Your key objects must implement:

- **`equals( Object )`**
- **`hashCode( )`**

Don’t change the value of any object used as a key!

Strings work great as keys

6. Ambiguity Abounds

Isn’t Overloading great?

6. Ambiguity Abounds

```
1: class BaseNum {
2:  public void myMethod( Number n ) {
3:    System.out.println("Base!");}}
4: class IntNum extends BaseNum {
5:  public void myMethod( Integer i ) {
6:    System.out.println("Integers Only");}}
7: public static void main(String[ ] args) {
8:  Number   n   = new Integer(1);
9:  Integer i    = new Integer(2);
10: BaseNum bn = new IntNum( );
11: bn.myMethod( n );   //prints?
12: bn.myMethod( i );   //prints?
13:}
```

6. Ambiguity Abounds

```
1: class BaseNum {
2:  public void myMethod( Number n )
3:   System.out.println("Base!");}}
4: class IntNum extends BaseNum {
5:  public void myMethod( Integer i ) {
6:   System.out.println("Integers Only");}}
7: public static void main(String[ ] args) {
8:  Number  n  = new Integer(1);
9:  Integer i  = new Integer(2);
10: BaseNum bn = new IntNum( );
11: bn.myMethod( n );   //prints?
12: bn.myMethod( i );   //prints?
13:}
```

a) Integers Only
   Integers Only

b) Base!
   Base!

c) Base!
   Integers Only

d) Something else

6. Ambiguity Abounds

```
1: class BaseNum {
2:  public void myMethod( Number n ) {
3:   System.out.println("Base!");}}
4: class IntNum extends BaseNum {
5:  public void myMethod( Integer i ) {
6:   System.out.println("Integers Only");}}
7: public static void main(String[ ] args) {
8:  Number  n  = new Integer(1);
9:  Integer i  = new Integer(2);
10: BaseNum bn = new IntNum( );
11: bn.myMethod( n );   //prints?
12: bn.myMethod( i );   //prints?
13:}
```

a) Integers Only
Integers Only

b) Base!
Base!

c) Base!
Integers Only

d) Something else

6. What's Wrong?

We don't seem to be paying any attention to the parameter type

These two methods are ambiguous

```
BaseNum: myMethod( Number n )
IntNum:  myMethod( Integer i )
```

- Because Integer extends Number

The methods are overloaded, not overridden

- Choice of which overloaded method to call is made at compile time

6. Second Look

```
1: class BaseNum {
2:  public void myMethod( Number n ) {
3:   System.out.println("Base!");}}
4: class IntNum extends BaseNum {
5:  public void myMethod( Integer i ) {
6:   System.out.println(
7: public static void m
8:  Number  n  = new Intege
9:  Integer i  = new Integer(2);
10: BaseNum bn = new IntNum( );
11: bn.myMethod( n );
12: bn.myMethod( i );
13:}
```

6. Ambiguity Abounds

```
1: class BaseNum {
2:  public void myMethod( Number n ) {
3:    System.out.println("Base!");}}
4: class IntNum extends BaseNum {
5:  public void myMethod( Integer i ) {
6:    System.out.println(
7: public static void m
8:  Number   n   = new Intege
9:  Integer i    = new Integer(2);
10: BaseNum bn = new IntNum( );
11: bn.myMethod( n );
12: bn.myMethod( i );
13:}
```

BaseNum.myMethod(Number)

6. Ambiguity: Fixes 1 and 2

1. Rename one method—to be distinct

```
BaseNum: myMethod( Number n )
IntNum:  myIntMethod( Integer i )
```

   3. This makes it obvious which method you're calling

2. Call **`myMethod`** using an **`IntNum`** object, and cast the parameter as an (**`Integer`**)
   1. This requires users of your class to change their code
   2. The compiler won't help you enforce it

6. Ambiguity Fix #2

for Users of Ambiguous Methods

```
1: class BaseNum {
2:  public void myMethod( Number n ) {
3:    System.out.println("Base!");}}
4: class IntNum extends BaseNum {
5:  public void myMethod( Integer i ) {
6:    System.out.println("Integers Only");}}
7: public static void main(String[ ] args) {
8:  Number  n = new Integer(1);
9:  Integer i = new Integer(2);
10: IntNum in = new IntNum( );
11: in.myMethod( (Integer)n );
12: in.myMethod( i );
13:}
```

6. Ambiguity Fix #3

for Class Designers

1. Use overriding, not overloading

   a. Replace IntNum: `myMethod(Integer)`
      with IntNum: `myMethod(Number)`

   b. Dispatch with `instanceof` inside myMethod
      1. Based on the type of object passed in

   c. The user of the class sees pure overriding
      1. Your classes are more user-proof

6. Ambiguity Fix #3

for Class Designers

```
1: class BaseNum {
2:  public void myMethod( Number n ) {
3:    System.out.println("Base!");}}
4: class IntNum extends BaseNum {
5:  public void myMethod( Number n ) {

5A:   if (n instanceof Integer)
5B:     System.out.println("Integers Only");
5C:   else
5D:     super.myMethod( n );
5E: }
```

6. Lessons in Overloading

Compiler may not choose what you expect

Beware of castable arguments:

- `myMethod( Number )`
- `myMethod( Integer )`

OK if parameters can’t be cast to each other

- `myMethod( Object )`
- `myMethod( int )`

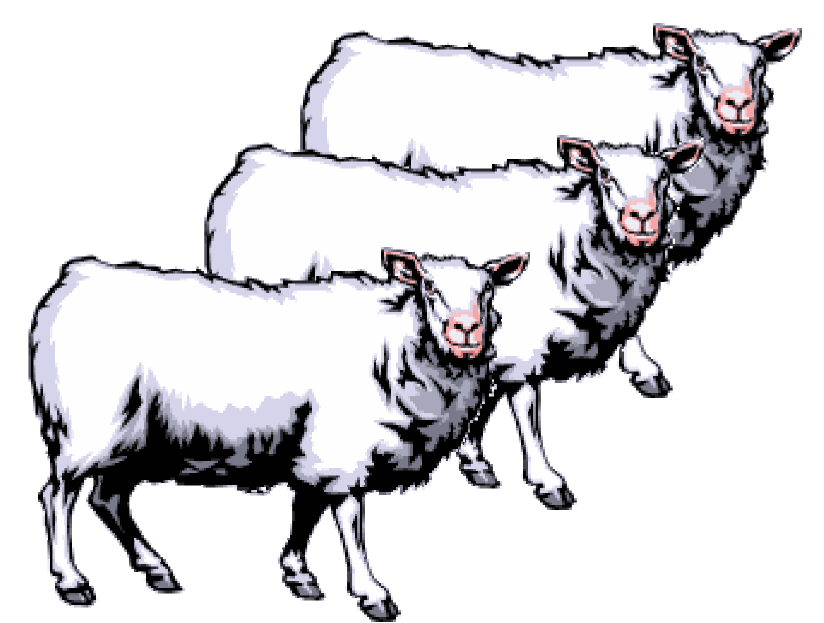
OK if other parameters are different

- `myMethod( Number, StringBuffer )`
- `myMethod( Integer, String )`

7. Who Do You Want to Be?

Cloning is simple.

Right.



7. Who Do You Want to Be?

This next example uses Cloning

Cloning bypasses all constructors to make a copy of the object

Java has a Cloneable interface—if you implement it, you tell Java that you support cloning via the **`clone()`** method

7. Who Do You Want to Be?

```
 1: class Customer { String name;  /* plus other data */ }
 2: class Acct implements Cloneable {
 3:   public Customer c = new Customer( );
 4:   public long bal;
 5:   public Object clone( ) {
 6:     try {  return super.clone( ); }
 7:     catch (CloneNotSupportedException e) {
 8:             throw new UnsupportedOperationException( );} }
 9: public static void main(String[ ] args) {
10:    Acct one = new Acct( );          one.c.name="Pat";  one.bal=500;
11:    Acct two=(Acct)one.clone( ); two.c.name="Sam"; two.bal=200;
12:    System.out.println(one.c.name + " has $" + one.bal);
13:    System.out.println(two.c.name + " has $" + two.bal);  } }
```

7. Prints:

```
1: class Customer { String name;  /* plus other da
2: class Acct implements Cloneable {
3:   public Customer c = new Customer( );
4:   public long bal;
5:   public Object clone( ) {
6:      try {  return super.clone( ); }
7:      catch (CloneNotSupportedException e) {
8:                throw new UnsupportedOperationException( );} }
9: public static void main(String[ ] args) {
10:    Acct one = new Acct( );          one.c.name="Pat";  one.bal=500;
11:    Acct two=(Acct)one.clone( ); two.c.name="Sam"; two.bal=200;
12:    System.out.println(one.c.name + " has $" + one.bal);
13:    System.out.println(two.c.name + " has $" + two.bal);   } }
```

a) Pat has $500
Sam has $200

b) Pat has $500
Pat has $500

c) Sam has $500
Sam has $200

d) Something else

7. Prints:

a) Pat has $500
Sam has $200

b) Pat has $500
Pat has $500

c) <u>Sam has $500</u>
<u>Sam has $200</u>

d) Something else

```
1: class Customer { String name;  /* plus other da
2: class Acct implements Cloneable {
3:   public Customer c = new Customer( );
4:   public long bal;
5:   public Object clone( ) {
6:     try {  return super.clone( ); }
7:     catch (CloneNotSupportedException e) {
8:              throw new UnsupportedOperationException( );}
9: public static void main(String[ ] args) {
10:    Acct one = new Acct( );        one.c.name="Pat";  one.bal=500;
11:    Acct two=(Acct)one.clone( ); two.c.name="Sam"; two.bal=200;
12:    System.out.println(one.c.name + " has $" + one.bal);
13:    System.out.println(two.c.name + " has $" + two.bal);  } }
```

7. What's Wrong?

Object Clone does shallow copy

- Customer c—the reference is copied!
- long bal—as a primitive, its value is copied
- So both Acct one and Acct two point to the same Customer object

Fixing this requires a deep copy

Deep copies copy values, not references

7. Second Look

```
 1: class Customer { String name;  /* plus other
 2: class Acct implements Cloneable {
 3:   public Customer c = new Customer( );
 4:   public long bal;
 5:   public Object clone( ) {
 6:     try {  return super.clone( ); }
 7:     catch (CloneNotSupportedException e) {
 8:              throw new UnsupportedOperationException( );} }
 9: public static void main(String[ ] args) {
10:    Acct one = new Acct( );         one.c.name="Pat";  one.bal=500;
11:    Acct two=(Acct)one.clone( ); two.c.name="Sam"; two.bal=200;
12:    System.out.println(one.c.name + " has $" + one.bal);
13:    System.out.println(two.c.name + " has $" + two.bal);  } }
```

Object.clone()

Customer c

long bal

7. Second Look

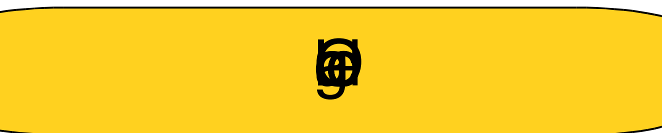
```
1: class Customer { String name;  /* plus other data */ }
2: class Acct implements Cloneable {
3:   public Customer c = new Customer( );
4:   public long bal;
5:   public Object clone( ) {
6:     try {  return super.clone( ); }
7:     catch (CloneNotSupportedException e) {
8:             throw new UnsupportedOperationException( );} }
9: public static void main(String[ ] args) {
10:    Acct one = new Acct( );          one.c.name="Pat";  one.bal=500;
11:    Acct two=(Acct)one.clone( ); two.c.name="Sam"; two.bal=200;
12:    System.out.println(one.c.name + " has $" + one.bal);
13:    System.out.println(two.c.name + " has $" + two.bal);  } }
```

one.c and two.c [illegible]

Line 11 overwrites line 10

7. Cloning: Fixed

Make a deep copy of the Customer object



```
5:  public Object clone( ) {
6:    try {
6A:     return super.clone( );
7:    } catch...
```

7. Cloning: Fixed

Make a deep copy of the Customer object

```
5:  public Object clone( ) {
6:    try {
6A:     Acct newAcct = (Acct)super.clone( );
6B:     return newAcct;
7:    } catch...
```

7. Cloning: Fixed

Make a deep copy of the Customer object

```
5:  public Object clone( ) {
6:    try {
6A:     Acct newAcct = (Acct)super.clone( );
6B:     newAcct.c = new Customer( newAcct.c );
6C:     return newAcct;
7:    } catch...
```

Note: Customer will need a copy constructor

7. Lessons in Cloning

Do you even need to support cloning?

Carefully consider when to make deep copies

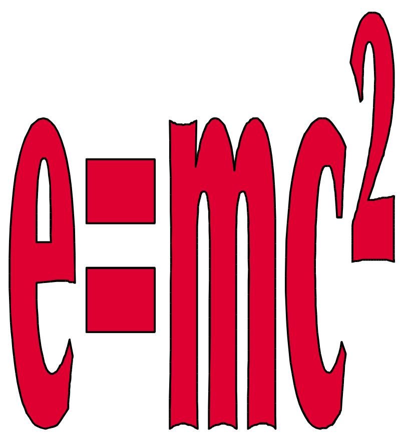
– Make copies of mutable objects

If you ever want to subclass

– Don’t call your constructor from clone( )
– Always call `super.clone()` to create the proper type of object

8. Apples and Oranges

This example shows how something as simple as **`equals()`** can trip you up…



8. Apples and Oranges

```
1: class Fruit {
 2:  int i = 1;
 3:  public boolean equals( Fruit f ) {
 4:    return (i = = f.i);
 5:  }
 6:  public static void main(String[ ] args) {
 7:    Object o = new Fruit( );
 8:    Fruit f = new Fruit( );
 9:  System.out.print( o.equals(f) ? "Obj=Fruit ":"Obj!=Fruit ");
10:  System.out.print( f.equals(o) ? "Fruit=Obj ":"Fruit!=Obj ");
11: }  }
```

8. Prints:

a)	Obj=Fruit	Fruit=Obj
b)	Obj!=Fruit	Fruit=Obj
c)	Obj=Fruit	Fruit!=Obj
d)	Obj!=Fruit	Fruit!=Obj

```
1: class Fruit {
2:  int i = 1;
3:  public boolean equals( Fruit f ) {
4:    return (i = = f.i);
5:  }
6:  public static void main(String[ ] args) {
7:    Object o = new Fruit( );
8:    Fruit f = new Fruit( );
9:  System.out.print( o.equals(f) ? "Obj=Fruit ":"Obj!=Fruit ");
10:  System.out.print( f.equals(o) ? "Fruit=Obj ":"Fruit!=Obj ");
11: }  }
```

8. Prints:

a)	Obj=Fruit	Fruit=Obj
b)	Obj!=Fruit	Fruit=Obj
c)	Obj=Fruit	Fruit!=Obj
d)	Obj!=Fruit	Fruit!=Obj

```
1: class Fruit {
 2:  int i = 1;
 3:  public boolean equals( Fruit f ) {
 4:    return (i = = f.i);
 5:  }
 6:  public static void main(String[ ] args) {
 7:    Object o = new Fruit( );
 8:    Fruit f = new Fruit( );
 9:  System.out.print( o.equals(f) ? "Obj=Fruit ":"Obj!=Fruit ");
10:  System.out.print( f.equals(o) ? "Fruit=Obj ":"Fruit!=Obj ");
11: }  }
```

8. What's Wrong?

- Our **`equals()`** method was never called!
- Consider its signature
  - public boolean equals(Fruit f) {...}
  - Signature: Fruit.equals(Fruit)

8. What's Wrong?

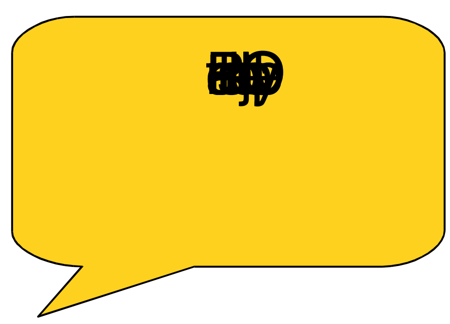
What equals( ) did we actually call?

- ...print(o.equals(f) ...);
  Signature: Object.equals(Fruit)
  compiles to: Object.equals(Object)

- ...print(f.equals(o) ...);
  Signature: Fruit.equals(Object)
  compiles to: Object.equals(Object)

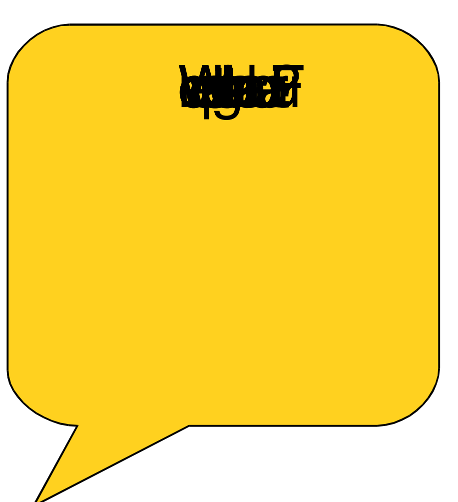
8. Apples and Oranges: Fix

```
3:  public boolean equals( Object o ) {
4:    return (i = = f.i);
5:  }
```



8. Apples and Oranges: Fix

```
3:  public boolean equals( Object o ) {
3A:   if (!(o instanceof Fruit))
3B:        return false;
4:    return (i = = f.i);
5:  }
```



8. Apples and Oranges: Fix

```
3:  public boolean equals( Object o ) {
3A:   if (!(o instanceof Fruit))
3B:         return false;
3C:   Fruit f = (Fruit)o;
4:    return (i = = f.i);
5:  }
```

8. How Does This Work?

What equals( ) will we call now?

- ...print(o.equals(f) ...);
  Signature: Object.equals(Fruit)
  Compiles to: Object.equals(Object)
  Fruit.equals(Object)

BUT overrides at runtime to Fruit

- ...print(f.equals(o) ...);
  Signature: Fruit.equals(Object)
  Compiles to: Fruit.equals(Object)

8. Lessons in Equality

Do you need to override equals?

Always override equals(Object)

Test for type with instanceof and cast:

```
if (!(o instanceof Fruit))
   return false;
Fruit f = (Fruit)o;
// use f in the code
```

If you override equals(), override hashCode() !

9. Are You Protected?

In this example, we look at access levels. If you came from a C++ background it's time to sit up.

9. Are You Protected?

```
1:   package testing;
2:   public class Super {
3:     public String reply = "S ";
4:   }
5:   public class Derived extends Super {
6:     protected String reply = "D ";
7:   }
8:   public class Flummox {
9:   public static void main(String[ ] args) {
10:    Derived d = new Derived( );
11:    System.out.print(d.reply + ((Super)d).reply);
12:} }
```

9. Prints:

a) D D

b) D S

c) S S

d) won't compile

```
1:   package testing;
2:   public class Super {
3:     public String reply = "S ";
4:   }
5:   public class Derived extends Super {
6:     protected String reply = "D ";
7:   }
8:   public class Flummox {
9:   public static void main(String[ ] args) {
10:    Derived d = new Derived( );
11:    System.out.print(d.reply + ((Super)d).reply);
12:} }
```

9. Prints:

a) D D

b) D S

c) S S

d) won't compile

```
1:   package testing;
2:   public class Super {
3:     public String reply = "S ";
4:   }
5:   public class Derived extends Super
6:     protected String reply = "D ";
7:   }
8:   public class Flummox {
9:   public static void main(String[ ] args) {
10:    Derived d = new Derived( );
11:    System.out.print(d.reply + ((Super)d).reply);
12:} }
```

9. Is Anything Wrong?

Java programmers say no

C++ programmers are surprised it compiles

9. Second Look

a) D D

b) D S

c) S S

d) won't compile

```
1:   package testing;
2:   public class Super {
3:     public String reply = "S ";
4:   }
5:   public class Derived extends Super {
6:     protected String reply = "D ";
7:   }
8:   public class Flummox {
9:   public static void main(String[ ] args) {
10:    Derived d = new Derived( );
11:    System.out.print(d.reply + ((Super)d).reply);
12:} }
```

Sun

a) D D

b) D S

c) S S

d) won't compile

```
1:   package testing;
2:   public class Super {
3:     public String reply = "S ";
4:   }
5:   public class Derived extends Super {
6:     protected String reply = "D ";
7:   }
8:   public class Flummox {
9:   public static void main(String[ ] args) {
10:    Derived d = new Derived( );
11:    System.out.print(d.reply + ((Super)d).reply);
12:} }
```

9. Lessons in Protection

protected allows greater access than package-private

Field or method is	Access in class	Access in package	Access in subclass	Access anywhere
public				
protected		!		–
package private			–	–
private		–	–	–

C++ has no packages

9. Lessons in Protection

protected allows greater access than package-private

Always use the most restrictive access for all methods and fields

- Preserves internal integrity

10. Where Are We Going?

One last surprise –
Overriding methods
can be painful

10. Where Are We Going?

```
 1: class Coaster {
 2:    String[ ] a = new String[1];
 3:    Coaster( ) { init( ); }
 4:    public void init( ) { a[0] = "I'm "; }
 5: }
 6: public class Roller extends Coaster {
 7:    String[ ] b = new String[1];
 8:    Roller( ) { init( ); }
 9:    public void init( ) { b[0] = "here "; }
10:   public static void main(String[ ] args) {
11:     Roller r = new Roller( );
12:     System.out.println( r.a[0] + r.b[0]);
13:} }
```

10. Prints:

a) I'm here
b) I'm null
c) null here
d) something else

```
 1: class Coaster {
 2:    String[ ] a = new String[1];
 3:    Coaster( ) { init( ); }
 4:    public void init( ) { a[0] = "I'm "; }
 5: }
 6: public class Roller extends Coaster {
 7:    String[ ] b = new String[1];
 8:    Roller( ) { init( ); }
 9:    public void init( ) { b[0] = "here "; }
10:   public static void main(String[ ] args) {
11:     Roller r = new Roller( );
12:     System.out.println( r.a[0] + r.b[0]);
13:} }
```

10. Prints:

a) I'm here
b) I'm null
c) null here
d) <u>something else</u>

```
1: class Coaster {
2:    String[ ] a = new String[1];
3:    Coaster( ) { init( ); }
4:    public void init( ) { a[0] = "I'm "; }
5: }
6: public class Roller extends Coaster {
7:    String[ ] b = new String[1];
8:    Roller( ) { init( ); }
9:    public void init( ) { b[0] = "here "; }
10:   public static void main(String[ ] args) {
11:      Roller r = new Roller( );
12:      System.out.println( r.a[0] + r.b[0]);
13:} }
```

10. What's Wrong?

- The original Coaster Constructor works
  - It fails when Coaster is subclassed!
- The secret: Be wary calling concrete overridable methods from public methods

```
1: class Coaster {
2:    String[ ] a = new String[1];
3:    Coaster( ) { init( ); }
4:    public void init( ) { a[0] = "I'm "; }
5: }
```

10. Second Look

```
1: class Coaster {
2:    String[ ] a = new String[1];
3:    Coaster( ) { init( ); }
4:    public void init( ) { a[0] = "I'm "; }
5: }
6: public class Roller extends Coaster {
7:    String[ ] b = new String[1];
8:    Roller( ) { init( ); }
9:    public void init( ) { b[0] = "here "; }
10:   public static void main(String[ ] args) {
11:     Roller r = new Roller( );
12:     System.out.println( r.a[0] + r.b[0]);
13:} }
```

10. Roller Coaster: Fix

Public methods call private non-overridable methods to do their work

```
1: class Coaster {
2:    String[ ] a = new String[1];
3:    Coaster( ) { privateInit( ); }
4:    public void init( ) { privateInit( );}
4A:   private void privateInit( ) {
4B:       a[0] = "I'm ";
4C:   }
5: }
```

10. Lessons in Overriding

Classes can fail in the future when a class gets subclassed

Avoid public methods that call overridable methods

- An override may corrupt your internal state

This is especially true of:

- constructors
- **`readObject( )`**
- **`clone( )`**

Summary

- Learn from the experts
  - Design Patterns/Practices to follow
- Avoid problematic constructs
  - Bug patterns/Practices to avoid
- Code Reviews are essential
- Be familiar with JDK classes

References

- Book: Java Pitfalls, by Daconta, et. al.
- Book: Effective Java, by Joshua Bloch
  http://java.sun.com/docs/books/effective/
- SUN site articles by Brian Goetz and others
  http://developer.java.sun.com/developer/technicalArticles/Programming

Marc Hamilton
marc.hamilton@sun.com
Sun Microsystems, Inc.