JavaOne℠

Sun's 2002 Worldwide Java Developer Conference

PROGRAMMING PUZZLERS

Joshua Bloch, Sr. Staff Engineer
Neal Gafter, Staff Engineer
Sun Microsystems, Java Software

Session 2500

Introduction

- Ten Java™ programming language puzzles
  - Short program with curious behavior
  - What does it print? (multiple choice)
  - The mystery revealed
  - How to fix the problem
  - The moral
- Covers language and core libraries

1. “All I Get Is Static”

```
class Dog {
    public static void bark() {
        System.out.print("woof ");
    }
}
class Basenji extends Dog {
    public static void bark() { }
}
public class Bark {
    public static void main(String args[]) {
        Dog woofer = new Dog();
        Dog nipper = new Basenji();
        woofer.bark();
        nipper.bark();
    }
}
```

What Does It Print?

(a) `woof`

(b) `woof woof`

(c) It varies

What Does It Print?

(a) `woof`

(b) `woof woof`

(c) It varies

No dynamic dispatch on static methods

Another Look

```
class Dog {
    public static void bark() {
        System.out.print("woof ");
    }
}
class Basenji extends Dog {
    public static void bark() { }
}
public class Bark {
    public static void main(String args[]) {
        Dog woofer = new Dog();
        Dog nipper = new Basenji();
        woofer.bark();
        nipper.bark();
    }
}
```

How Do You Fix It?

Remove `static` from the bark method

The Moral

- Static methods can’t be overridden
  - They can only be *hidden*
- Don’t hide static methods
- Never invoke static methods on instances
  - Not **`instance.staticMethod()`**
  - But **`Class.staticMethod()`**

2. “What’s in a Name?”

```
public class Name {
    private String first, last;
    public Name(String first, String last) {
        this.first = first;
        this.last  = last;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name)o;
        return n.first.equals(first) && n.last.equals(last);
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Donald", "Duck"));
        System.out.println(
          s.contains(new Name("Donald", "Duck")));
    }
}
```

What Does It Print?

(a) `true`

(b) `false`

(c) It varies

What Does It Print?

(a) `true`

(b) `false`

(c) It varies

Donald is in the set, but the set can’t find him

The `Name` class violates the `hashCode` contract

Another Look

```
public class Name {
    private String first, last;
    public Name(String first, String last) {
        this.first = first;
        this.last  = last;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Name)) return false;
        Name n = (Name)o;
        return n.first.equals(first) &&
  n.last.equals(last);
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Donald", "Duck"));
        System.out.println(
          s.contains(new Name("Donald", "Duck")));
    }
}
```

How Do You Fix It?

Add a **`hashCode`** method:

```
public int hashCode() {
    return 31 * first.hashCode() + last.hashCode();
}
```

The Moral

Override **`hashCode`** when overriding **`equals`**

Obey general contracts when overriding

See *Effective Java*^{TM}, Chapter 3

3. “Indecision”

```
class Indecisive {
    public static void main(String[] args) {
        System.out.println(waffle());
    }

    static boolean waffle() {
        try {
            return true;
        } finally {
            return false;
        }
    }
}
```

What Does It Print?

(a) `true`

(b) `false`

(c) None of the above

What Does It Print?

(a) `true`

(b) `false`

(c) None of the above

The `finally` is processed after the `try`.

Another Look

```
class Indecisive {
    public static void main(String[] args) {
        System.out.println(waffle());
    }

    static boolean waffle() {
        try {
            return true;
        } finally {
            return false;
        }
    }
}
```

The Moral

- Avoid abrupt completion of **`finally`** blocks
  - Wrap unpredictable actions with nested trys
  - Don’t return or throw exceptions

4. “The Saga of the Sordid Sort”

```
public class SordidSort {
    public static void main(String args[]) {
        Integer big   = new Integer( 2000000000);
        Integer small = new Integer(-2000000000);
        Integer zero  = new Integer(0);
        Integer[] arr = new Integer[] {big, small, zero};

        Arrays.sort(arr, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Integer)o2).intValue() -
                       ((Integer)o1).intValue();
            }
        });
        System.out.println(Arrays.asList(arr));
    }
}
```

What Does It Print?

(a) `[-2000000000, 0, 2000000000]`

(b) `[2000000000, 0, -2000000000]`

(c) `[-2000000000, 2000000000, 0]`

(d) It varies

What Does It Print?

(a) `[-2000000000, 0, 2000000000]`

(b) `[2000000000, 0, -2000000000]`

(c) `[-2000000000, 2000000000, 0]`

(d) It varies (behavior is undefined)

The comparator is broken!

- It relies on `int` subtraction
- `int` too small to hold difference of 2 arbitrary `ints`

Another Look

```
public class SordidSort {
    public static void main(String args[]) {
        Integer big   = new Integer( 2000000000);
        Integer small = new Integer(-2000000000);
        Integer zero  = new Integer(0);
        Integer[] arr = new Integer[] {big, small, zero};

        Arrays.sort(arr, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Integer)o2).intValue() -
                       ((Integer)o1).intValue();
            }
        });
        System.out.println(Arrays.asList(arr));
    }
}
```

How Do You Fix It?

Replace comparator with one that works

```
public int compare(Object o1, Object o2) {
    int i1 = ((Integer)o1).intValue();
    int i2 = ((Integer)o2).intValue();
    return (i2 < i1 ? -1 : (i2 == i1 ? 0 : 1));
}
```

The Moral

- `ints` aren't integers!
- Think about overflow
- This particular comparison technique
  - OK only if `max - min <= Integer.MAX_VALUE`
  - For example: all values positive
- Don't write overly clever code

5. “You’re Such a Character”

```
public class Trivial {
    public static void main(String args[]) {
        System.out.print("H" + "a");
        System.out.print('H' + 'a');
    }
}
```

JavaOne

What Does It Print?

(a) **`HaHa`**

(b) **`Ha`**

(c) None of the above

What Does It Print?

(a) `HaHa`

(b) `Ha`

(c) None of the above: It prints `Ha169`

`'H' + 'a'` evaluated as `int`, then converted to `String`. Ouch.

The Moral

- Use string concatenation (**+**) with care
  - At least one operand must be a **`String`**
  - If it isn't, cast or convert
- Be glad operator overloading isn't supported

6. “The Case of the Confusing Constructor”

```
public class Confusing {
    public Confusing(Object o) {
        System.out.println("Object");
    }
    public Confusing(double[] dArray) {
        System.out.println("double array");
    }
    public static void main(String args[]) {
        new Confusing(null);
    }
}
```

What Does It Print?

(a) `Object`

(b) `double array`

(c) None of the above

What Does It Print?

(a) `Object`

(b) `double array`

(c) None of the above

When multiple overloadings apply,
the most specific wins

Another Look

```
public class Confusing {
    public Confusing(Object o) {
        System.out.println("Object");
    }
    public Confusing(double[] dArray) {
        System.out.println("double array");
    }
    public static void main(String args[]) {
        new Confusing(null);
    }
}
```

How Do You Fix It?

There may be no problem

If there is, use a cast:

```
new Confusing((Object)null);
```

The Moral

Avoid overloading

If you overload, avoid ambiguity

If you do have ambiguous overloadings, make their behavior identical

If you are using a “broken” class, make your intentions clear with a cast

7. “A Big Delight in Every Byte”

```
class ByteMe {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
            b < Byte.MAX_VALUE; b++) {
            if (b == 0x90)
                System.out.print("Byte me! ");
        }
    }
}
```

What Does It Print?

(a) (nothing)

(b) `Byte me!`

(c) `Byte me! Byte me!`

What Does It Print?

(a) (nothing)

(b) **`Byte me!`**

(c) **`Byte me! Byte me!`**

Program compares a **`byte`** with an **`int`**

`byte` is *promoted* with surprising results

Another Look

```
class ByteMe {
    public static void main(String[] args) {
         for (byte b = Byte.MIN_VALUE;
                 b < Byte.MAX_VALUE; b++) {
                 if (b == 0x90)  // (b == 144)
                     System.out.print("Byte me! ");
         }
    }
}

// But (byte)0x90 == -112
```

How Do You Fix It?

Cast `int` to `byte`

```
 if (b == (byte)0x90)
     System.out.println("Byte me!");
```

Or convert `byte` to `int`, suppressing sign extension with mask

```
 if ((b & 0xff) == 0x90)
     System.out.println("Byte me!");
```

The Moral

- `bytes` aren't `ints`
- Be careful when mixing primitive types
- Compare like-typed expressions
  - Cast or convert one operand as necessary

8. “Time for a Change”

If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```
public class Change {
     public static void main(String args[]) {
         System.out.println(2.00 - 1.10);
     }
 }
```

What Does It Print?

(a) `0.9`

(b) `0.90`

(c) It varies

(d) None of the above

What Does It Print?

(a) `0.9`

(b) `0.90`

(c) It varies

(d) None of the above: 0.8999999999999999

Decimal values can’t be represented exactly by **`float`** or **`double`**

How Do You Fix It?

```
import java.math.BigDecimal;
public class Change2 {
    public static void main(String args[]) {
        System.out.println(
            new BigDecimal("2.00").subtract(
                new BigDecimal("1.10")));
    }
}

public class Change {
    public static void main(String args[]) {
        System.out.println(200 - 110);
    }
}
```

JavaOne

The Moral

Avoid **`float`** and **`double`** where exact answers are required

Use **`BigDecimal`**, **`int`**, or **`long`** instead

9. “A Private Matter”

```
class Base {
    public String name = "Base";
}
class Derived extends Base {
    private String name = "Derived";
}
public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(new Derived().name);
    }
}
```

What Does It Print?

(a) `Derived`

(b) `Base`

(c) Compiler error in class `Base`:
Can’t assign weaker access to `name`

(d) None of the above

What Does It Print?

(a) **`Derived`**

(b) **`Base`**

(c) Compiler error in class **`Base`**:
Can’t assign weaker access privileges to **`k`**

(d) None of the above: Compiler error in class **`PrivateMatter`**: Can’t access **`name`**

Private method can’t override public,
but private field can hide public

Another Look

```
class Base {
    public String name = "Base";
}
class Derived extends Base {
    private String name = "Derived";
}
public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(new Derived().name);
    }
}
```

How Do You Fix It?

```
class Base {
    public String getName() { return "Base"; }
}
class Derived extends Base {
    public String getName() { return "Derived"; }
}
public class PrivateMatter {
    public static void main(String[] args) {
        System.out.println(new Derived().getName());
    }
}
```

The Moral

- Avoid hiding
    - Violates *subsumption*
- Avoid public fields
    - Use accessor methods instead

10. “Loopy Behavior”

```
class Loopy {
    public static void main(String[] args) {
        final int start = Integer.MAX_VALUE - 100;
        final int end   = Integer.MAX_VALUE;
        int count = 0;
        for (int i = start; i <= end; i++)
            count++;
        System.out.println(count);
    }
}
```

What Does It Print?

(a) `100`

(b) `101`

(c) (nothing)

What Does It Print?

(a) `100`

(b) `101`

(c) (nothing)

The loop test is broken—infinite loop!

Another Look

```
class Loopy {
    public static void main(String[] args) {
        final int start = Integer.MAX_VALUE - 100;
        final int end   = Integer.MAX_VALUE;
        int count = 0;
        for (int i = start; i <= end; i++)
            count++;
        System.out.println(count);
    }
}
```

How Do You Fix It?

Change loop variable from `int` to `long`

```
for (long i = start; i <= end; i++)
    count++;
```

The Moral

`ints` aren’t integers!

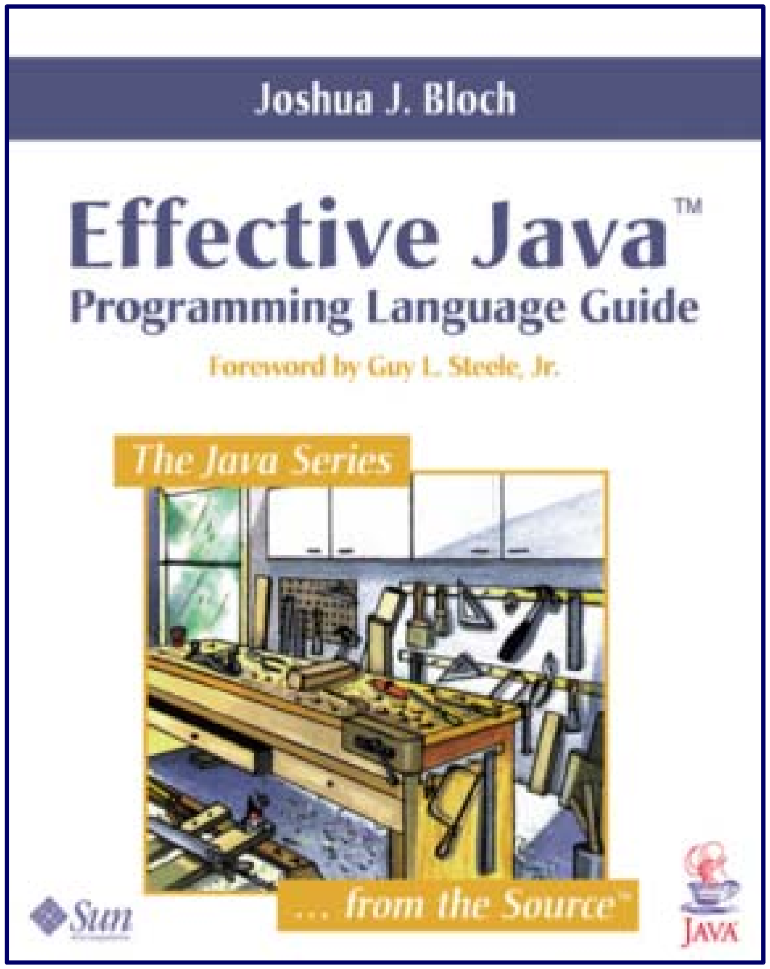
Think about overflow

Use larger type if necessary

Conclusion

- Java™ platform is simple and elegant
  - But it has a few sharp corners—avoid them!
- Keep programs simple
  - Avoid name reuse: overloading, hiding, shadowing
- If you aren't sure what a program does, it probably doesn't do what you want it to

Shameless Commerce Division



Send Us Your Puzzlers!

If you have a puzzler for us, send it to:

javapuzzlers@sun.com

JavaOne℠

Sun's 2002 Worldwide Java Developer Conference™

BEYOND BOUNDARIES

Session 2500

1. “Random Behavior”

```
public class RandomSet {
    public static void main(String[] args) {
        Set s = new HashSet();
        for (int i = 0; i < 100; i++)
            s.add(randomInteger());
        System.out.println(s.size());
    }

    private static Integer randomInteger() {
        return new Integer(new Random().nextInt());
    }
}
```

What Does It Print?

(a) A number close to 1

(b) A number close to 50

(c) A number close to 100

(d) None of the above

What Does It Print?

(a) A number close to 1

(b) A number close to 50

(c) A number close to 100

(d) None of the above

A new random number generator is created each iteration and the seed changes rarely
if at all.

Another Look

```
public class RandomSet {
    public static void main(String[] args) {
        Set s = new HashSet();
        for (int i=0; i<100; i++)
            s.add(randomInteger());
        System.out.println(s.size());
    }

    private static Integer randomInteger() {
        return new Integer(new Random().nextInt());
    }
}
```

How Do You Fix It?

```
public class RandomSet {
    public static void main(String[] args) {
        Set s = new HashSet();
        for (int i=0; i<100; i++)
            s.add(randomInteger());
        System.out.println(s.size());
    }

    private static Random rnd = new Random();

    private static Integer randomInteger() {
        return new Integer(rnd.nextInt());
    }
}
```

The Moral

- Use one **`Random`** instance for each sequence
- In most programs, one is all you need
- In multithreaded programs, you *may* want multiple instances for increased concurrency
  - Seed explicitly or risk identical sequences
  - Generally ok to use one instance to seed others

2.“Making a Hash of It”

```
public class Name {
    private String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    public boolean equals(Name o) {
        return first.equals(o.first) && last.equals(o.last);
    }
    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
            s.contains(new Name("Mickey", "Mouse")));
    }
}
```

What Does It Print?

(a) `true`

(b) `false`

(c) It varies

What Does It Print?

(a) `true`

(b) `false`

(c) It varies

`Name` overrides `hashCode` but not `equals`.
The two `Name` instances are unequal.

Another Look

```
public class Name {
    private String first, last;
    public Name(String first, String last) {
        if (first == null || last == null)
            throw new NullPointerException();
        this.first = first; this.last = last;
    }
    public boolean equals(Name o) { // Accidental overloading
        return first.equals(o.first) && last.equals(o.last);
    }
    public int hashCode() {            // Overriding
        return 31 * first.hashCode() + last.hashCode();
    }
    public static void main(String[] args) {
        Set s = new HashSet();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(
          s.contains(new Name("Mickey", "Mouse")));
    }
}
```

How Do You Fix It?

- Replace the overloaded **`equals`** method with an overriding **`equals`** method

```
 public boolean equals(Object o) {
     if (!(o instanceof Name))
         return false;
     Name n = (Name)o;
     return n.first.equals(first) && n.last.equals
(last);
 }
```

The Moral

- If you want to override a method:
  - Make sure signatures match
  - The compiler doesn't check for you
  - *Do* copy-and-paste declarations!

6. “All Strung Out”

```
public class Puzzling {
    public static void main(String[] args) {
        String s = new String("blah");
        System.out.println(s);
    }
}
class String {
    java.lang.String s;

    public String(java.lang.String s) {
        this.s = s;
    }
    public java.lang.String toString() {
        return s;
    }
}
```

What Does It Print?

(a) Won’t compile

(b) **`blah`**

(c) Throws an exception at runtime

(d) Other

What Does It Print?

(a) Won’t compile

(b) **`blah`**

(c) Throws an exception at runtime

(d) Other

`NoSuchMethodError` is thrown because the **`Puzzling`** class is missing a **`main`** method.

Another Look

```java
public class Puzzling {
    public static void main(String[] args) {
        String s = new String("blah");
        System.out.println(s);
    }
}
class String {
    java.lang.String s;

    public String(java.lang.String s) {
        this.s = s;
    }
    public java.lang.String toString() {
        return s;
    }
}
```

How Do You Fix It?

```
public class Puzzling {
    public static void main(String[] args) {
        MyString s = new MyString("blah");
        System.out.println(s);
    }
}
class MyString {
    String s;

    public MyString(String s) {
        this.s = s;
    }
    public String toString() {
        return s;
    }
}
```

The Moral

- Avoid name reuse in all its guises
  - hiding, shadowing, overloading
- Don’t even think about reusing platform class names!

2. “No Pain, No Gain”

```
public class Rhymes {
   private static Random rnd = new Random();
   public static void main(String[] args) {
      StringBuffer word = null;
      switch(rnd.nextInt(2)) {
         case 1:  word = new StringBuffer('P');
         case 2:  word = new StringBuffer('G');
         default: word = new StringBuffer('M');
      }
      word.append('a');
      word.append('i');
      word.append('n');
      System.out.println(word);
   }
}
```

Thanks to madbot (also known as Mike McCloskey)

What Does It Print?

(a) **`Pain`**, **`Gain`**, or **`Main`** (varies at random)

(b) **`Pain`** or **`Main`** (varies at random)

(c) **`Main`** (always)

(d) None of the above

What Does It Print?

(a) **`Pain`**, **`Gain`**, or **`Main`** (varies at random)

(b) **`Pain`** or **`Main`** (varies at random)

(c) **`Main`** (always)

(d) None of the above: **`ain`** (always)

The program has three separate bugs.
One of them is quite subtle.

Another Look

```
public class Rhymes {
   private static Random rnd = new Random();
   public static void main(String[] args) {
      StringBuffer word = null;
      switch(rnd.nextInt(2)) {  // No breaks!
         case 1:  word = new StringBuffer('P');
         case 2:  word = new StringBuffer('G');
         default: word = new StringBuffer('M');
      }
      word.append('a');
      word.append('i');
      word.append('n');
      System.out.println(word);
   }
}
```

How Do You Fix It?

```
public class Rhymes {
   private static Random rnd = new Random();
   public static void main(String[] args) {
      StringBuffer word = null;
      switch(rnd.nextInt(3)) {
         case 1:  word = new StringBuffer("P"); break;
         case 2:  word = new StringBuffer("G"); break;
         default: word = new StringBuffer("M"); break;
      }
      word.append('a');
      word.append('i');
      word.append('n');
      System.out.println(word);
   }
}
```

The Moral

- Use common idioms
  - If you must stray, consult the documentation
- Chars are not strings; they’re more like ints
- Always remember breaks in `switch` statement
- Watch out for fence-post errors
- Watch out for sneaky puzzlers