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
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
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()`
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
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:

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

- Override `hashCode` when overriding `equals`
- Obey general contracts when overriding
- See *Effective Java™*, Chapter 3
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.
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 try blocks
- Don’t return or throw exceptions
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 \texttt{int} subtraction
   \texttt{int} too small to hold difference of 2 arbitrary \texttt{ints}
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));
    }
}
Replace comparator with one that works

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

```java
new Confusing((Object)null);
```
The Moral

- Avoid overloading
- If you overload, avoid ambiguity
- If you do have ambiguous overloading, make their behavior identical
- If you are using a “broken” class, make your intentions clear with a cast
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
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

\[ \text{bytes aren't ints} \]
\[ \text{Be careful when mixing primitive types} \]
\[ \text{Compare like-typed expressions} \]
\[ \mu \text{Cast or convert one operand as necessary} \]
If you pay $2.00 for a gasket that costs $1.10, how much change do you get?

```java
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
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);
    }
}
The Moral

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

Use `BigDecimal`, `int`, or `long` instead.
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
  - μ Violates *subsumption*
- Avoid public fields
  - μ 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!
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`

```java
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
Send Us Your Puzzlers!

If you have a puzzler for us, send it to:
μ javapuzzlers@sun.com