Effective Java Puzzlers

JSUG 9th Meeting
Vienna, 12.01.2009
Christoph Pickl
Agenda

i. The books
   - Effective Java & Java Puzzlers

ii. It’s your turn
   - Solve some puzzles at your own

iii. Solve’em together
   - Provide solutions and background infos

iv. Good advice
   - Some nuggets to be more effective

v. Summary
The Books
Effective Java

- by Joshua Bloch
- designed/implemented many Java platform libraries
- 57 items on 252 pages
- program from an API designer’s point of view

114 Reviews

<table>
<thead>
<tr>
<th>Rating</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 star</td>
<td>103</td>
</tr>
<tr>
<td>4 star</td>
<td>5</td>
</tr>
<tr>
<td>3 star</td>
<td>4</td>
</tr>
<tr>
<td>2 star</td>
<td>2</td>
</tr>
<tr>
<td>1 star</td>
<td>0</td>
</tr>
</tbody>
</table>
Java Puzzlers

- by Joshua Bloch and Neal Gafter
- 95 puzzles on 282 pages
- Covers different topics
  - Expressions, Strings, Loops, Exceptions, Classes, Threads, Java Library, Serialization

23 Reviews

- 5 star: (19)
- 4 star: (1)
- 3 star: (1)
- 2 star: (1)
- 1 star: (1)

amazon.com
Puzzle Alone
Puzzle alone

- Hand out questionnaires and pencils
  - 18 puzzles
- **45 minutes** time (max!)
- Providing name is **optionally**
  - Results will be **evaluated**
  - Best result will be placed in “hall of fame”
- Most of them are multiple choice
  - Make use of “Other...” option
- No talking, no cheating, no use of internet :)
Puzzle Together
### #1 Simple Subtraction

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

// solution #1: poor - still uses binary floating-point!
System.out.printf("%.2f%n", 2.00 - 1.10);

// solution #2: use integral types
System.out.println((200 - 110) + " cents");

// solution #3: use BigDecimal(String)
System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.10"))); 

---

Avoid `float` and `double` where exact answers are required; for monetary calculations, use `int`, `long` or `BigDecimal`
public class SimpleAddition {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
    }
}

List<String> l = new ArrayList<String>();
l.add("Foo");
System.out.println(l);

System.out.println(12345 + 5432L);

always use a capitel L in long literals, never a lowercase l
public class SimpleDivision {
    public static void main(String[] args) {
        final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}

// computation of constant overflows!
long MICROS_PER_DAY = ((int) (24 * 60 * 60 * 1000 * 1000));
// afterwards widening primitive conversion [JLS 5.1.2]
final long MICROS_PER_DAY = 24L * 60 * 60 * 1000 * 1000;
final long MILLIS_PER_DAY = 24L * 60 * 60 * 1000;
System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);

when working with large numbers, watch out for overflow - it’s a silent killer
#4 Compound Legal

```java
    ______ x = ______;
    ______ i = ______;
    x += i;    // first statement legal
    x = x + i; // second statement illegal
```

```java
short x = 0;
int i = 123456;
x += i;    // narrowing primitive conversion [JLS 5.1.3]
x = x + i; // won’t compile: “possible loss of precision”
```

// [JLS 15.26.2] says about compound assignment operator:
// E1 op= E2 <=> E1 = (T) ((E1) op (E2))

**do not use compound assignment operators on variables of type byte, short or char**
#5 Compound Illegal

```
  _______ x = _______;  
  _______ i = _______;  
  x += i;  // first statement illegal  
  x = x + i;  // second statement legal  
```

**Object**
```
x = "object string ";
```

**String**
```
i = "real string";
```

```
x += i;  // left-hand side object reference type != String  
x = x + i;  // is assignment compatible [JLS 5.2]  
           // string concatenation is performed [JLS 15.26.2]
```
do not use Unicode escapes to represent ASCII characters; avoid Unicode escapes except where they are truly necessary
public class Classifier {
    public static void main(String[] args) {
        System.out.println(classify('n') +
                            classify('+') + classify('2'));
    }
    public static String classify(char c) {
        if("0123456789".indexOf(c) >= 0)
            return "NUMERAL ";
        if("abcdefghijklmnopqrstuvwxyz".indexOf(c) >= 0)
            return "LETTER ";
        // TODO finish implementation of operator classification
        // if("+-*/&|!=".indexOf(c) >= 0)
        //    return "OPERATOR ";
        //
        return "UNKOWN ";
    }
}

comment out a section of code by make use of a sequence of single-line comments
public class InTheLoop {
    public static final int END = Integer.MAX_VALUE;
    public static final int START = END - 100;
    public static void main(String[] args) {
        int count = 0;
        for (int i = START; i <= END; i++)
            count++;
        System.out.println(count);
    }
}

for (long i = START; i <= END; i++)
count++;
#9 Never Ending Story

```java
int start = Integer.MAX_VALUE - 1;
for (int i = start; i <= start + 1; i++) {
}

double i = Double.POSITIVE_INFINITY; // see [IEEE-754]
while (i == i + 1) {
}

double i = Double.NaN; // see [JLS 15.21.1]
while (i != i) {
}

String i = "foobar"; // see [JLS 15.18.1]
while (i != i + 0) {
}

Integer i = new Integer(0);
Integer j = new Integer(0);
while (i <= j && j <= i && i != j) {
}
```

binary floating-point arithmetic is only an approximation to real arithmetic; operator overloading can be very misleading
public class Confusing {
    public Confusing(Object o) {
        System.out.println("Object");
    }
    public Confusing(double[] d) {
        System.out.println("double array");
    }
    public static void main(String[] args) {
        new Confusing(null);
    }
}

// overloading process operates in two phases [JLS 15.12.2.5]
new Confusing((Object) null);

avoid overloading; use different names for different methods
(not possible for constructors, therefore use static factory methods)
public class Type1 {
    public static void main(String[] args) {
        String s = null;
        System.out.println(s instanceof String);
    }
}

public class Type2 {
    public static void main(String[] args) {
        System.out.println(new Type2() instanceof String);
    } // compile time error!!! [JLS 15.20.2, 15.16, 5.5]
}

public class Type3 {
    public static void main(String[] args) {
        Type3 t = (Type3) new Object();
    } // runtime exception!!!
}
class Point {
    final int x, y;
    final String name;
    Point (int X, int Y) {
        x=X; y=Y;
        name = makeN();
    }
    String makeN() {
        return "["+x+","+y+"]";
    }
    final String toString() {
        return name;
    }
}

class Point2 extends Point {
    final String c;
    Point2(int x,int y,String C) {
        super(x, y);
        c = C;
    }
    String makeN() {
        return super.makeN()+":"+c;
    }
    public static void main (...) {
        System.out.println(
            new Point2(4,2,"purple");
        } // prints "[4,2]:purple"
    } // prints "[4,2]:null"
class Point {
  final int x, y;
  final String name;
  Point (int X, int Y) {
    x=X; y=Y;
    // lazy initializing
  }
  String makeN() {
    return "["+x+","+y+"]";
  }
  String toString() {
    if(name == null) {
      name = makeN();
    }
    return name;
  }
}

class Point2 extends Point {
  final String c;
  Point2(int x,int y,String C) {
    super(x, y);
    c = C;
  }
  String makeN() {
    return super.makeN()+":"+c;
  }
  public static void main (...) {
    System.out.println(
      new Point2(4,2,"purple"));
  }
}

it's possible observing final instance field before its value has been assigned;
never call {overridable methods} from constructors
public class Null {
    public static void greet() {
        System.out.println("Hello world!");
    }
    public static void main(String[] args) {
        System.out.println(((Null) null).greet());
    }
}

System.out.println(Null.greet());

invoke static methods in a static way
public class Name {
    private final 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<Name> set = new HashSet<Name>();
        set.add(new Name("Spar", "Dat");
        System.out.println(set.contains(new Name("Spar", "Dat")));
    }
}
public class Name {
    private final 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 int hashCode() {
        return 37 * first.hashCode() + last.hashCode();
    }
}

you MUST override hashCode whenever you override equals
public class ShadesOfGray {
    public static void main(String[] args) {
        System.out.println(X.Y.Z);
    }
}

class X {
    static class Y {
        static String Z = "Black";
    }
    static C Y = new C();
}
class C {
    static String Z = "White";
}

// when a variable and a type have the same name and
// both are in scope, the variable takes precedence [JLS 6.5.2]
```java
public class ShadesOfGray {
    public static void main(String[] args) {
        System.out.println(Ex.Why.z);
    }
}

class Ex {
    static class Why {
        static String z = "Black";
    }
    static See y = new See();
}
class See {
    String z = "White";
}
```

always obey the standard Java **naming conventions**
A Glossary of Name Reuse

- **Overriding**
  - method overrides other superclass’ instance methods with the same signature (enabling *dynamic dispatch*)

- **Hiding**
  - field/static method/member type hides other with same name (signature) of supertypes

- **Overloading**
  - method with the same name but with another signature

- **Shadowing**
  - variable/method/type shadows other with same name&scope

- **Obscuring**
  - variable obscures a type with the same name
public class Reflector {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("foo");
        Iterator it = set.iterator();
        Method m = it.getClass().getMethod("hasNext");
        System.out.println(m.invoke(it));
    }
}

Exception in thread "main" IllegalAccessException:
    Class Reflector can not access a member of a class HashMap
    $HashIterator with modifiers "public"
// you cannot legally access a member of
// a nonpublic type from another package [JLS 6.6.1]
Method m = Iterator.class.getMethod("hasNext");

when accessing a type reflectively,
use a Class object that represents an accessible type
#17 Lazy Initialization

Class initialization [JLS 12.4.2]

- The class is **not yet** initialized.
- The class is **being** initialized by the **current** thread: a recursive request for initialization.
- The class is **being** initialized by some thread **other** than the current thread.
- The class is **already** initialized
public class Lazy {
    private static boolean initialized = false;
    static {
        Thread thread = new Thread(new Runnable() {
            public void run() {
                initialized = true;
            }
        });
        thread.start();
        try {
            thread.join();
        } catch(InterruptedException e) {
            throw new AssertionError(e);
        }
    }
    public static void main(String[] args) {
        System.out.println(initialized);
    }
}
#18 Class Warfare

API designers should **think** long and hard before exporting a **constant field**
Effective Nuggets
always override `toString`
static factory methods instead constructors
favor immutability
favor composition over inheritance
prefer interfaces to abstract classes
use overloading rarely
string concatenation’s performance
favor static over nonstatic member classes
minimize accessibility
Summary

- binary floating-point arithmetic is inexact
- be aware of silent overflows
- obey general naming conventions
- overriding equals => overriding hashCode
- carefully read API documentation

if you are not shure what a piece of code does, it’s very likely that it doesn’t do what you want it to do
Hardcore Java

- by Robert Simmons
- 344 pages full of *hardcore* stuff
- Covers:
  - Final (!), Immutable Types, Collections, Exceptions, Constants, Nested Classes, Reflection, References
That’s it