CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Effective Java

Department of Computer Science
University of Maryland, College Park

Effective Java Textbook

- Title
  - Effective Java, Second Edition
- Author
  - Joshua Bloch
- Contents
  - Learn to use Java language and its libraries more effectively

Java Puzzlers (By J. Bloch)

- Java
  - Simple and elegant
  - Need to avoid some sharp corners!
- Puzzlers
  - Java code fragments
  - Expose some tricky aspects of Java
- Effective Java
  - Patterns and idioms to emulate
  - Pitfalls to avoid

What's In A Name?

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

Output

1. **True**
2. **False**
3. **It Varies**

Name class violates Java hashCode( ) contract.

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

You're Such A Character

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

Output

1. **Ha**
2. **HaHa**
3. **Neither**

Prints Ha169

'H' + 'a' evaluated as *int*, then converted to String!

Use string concatenation (+) with care. At least one operand must be a String

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);
    }
}
```

Output

1. **Object**
2. **double array**
3. **Neither**

When multiple overloadings apply, the most specific wins

Avoid overloading. If you overload, avoid ambiguity

Time For A Change

- Problem
  - 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);
    }
}
```

Output

1. **0.9**
2. **0.90**
3. **Neither**

Prints 0.899999999999999. Decimal values can't be represented exactly by float or double

Avoid float or double where exact answers are required. Use BigDecimal, int, or long instead

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);
    }
}
```

Output

1. **Derived**
2. **Base**
3. **Neither**

Compiler error in class PrivateMatter: Can't access name

Private field can hide public. Avoid hiding & public fields

Regarding Objects

- Creating and destroying objects
  - Avoid creating duplicate/unnecessary objects
  - Eliminate obsolete object references
  - Avoid finalizers
- Methods common to all objects
  - Obey the general hash contract when overriding equals
  - Always override hashCode when you override equals
  - Always override toString

Classes and Interfaces

- Minimize the accessibility of classes and members
- Favor immutability
- Favor composition over inheritance
- Design and document for inheritance or else prohibit it
- Prefer interfaces to abstract classes
- User interfaces only to define types

Methods

- Check parameters for validity
- Make defensive copies when needed
- Design method signatures carefully
- Use overloading judiciously
- Return zero-length arrays, not nulls
- Write doc comments for all exposed API elements

General Programming

- Minimize the scope of local variables
- Prefer for-each loops to traditional for loops
- Know and use the libraries
- Prefer primitive types to boxed primitives
- Avoid float and double if exact answers are required
- Avoid strings where other types are more appropriate
- Beware the performance of string concatenation
- Adhere to generally accepted naming conventions
- Refer to objects by their interfaces

Exceptions

- Use exceptions only for exceptional conditions
- Use checked exceptions for recoverable conditions and run-time exceptions for programming errors
- Avoid unnecessary use of checked exceptions
- Favor the use of standard exceptions
- Throw exceptions appropriate to the abstraction
- Document all exceptions thrown by each method
- Don't ignore exceptions

Generics

- Don’t use raw types
  - E.g., raw type for List<E> is List
- Prefer lists to arrays
- Favor generic types and methods
  - Define classes and methods using generics when possible
- Use bounded wildcards to increase API flexibility

Avoid Duplicate Object Creation

- Reuse existing object instead
  - Reuse improves clarity and performance
- Simplest example

```
String s = new String("DON'T DO THIS!");
String s = "Do this instead";
```

  - Since Strings constants are reused
- In loops, savings can be substantial
- But don't be afraid to create objects
  - Object creation is cheap on modern JVMs
  - Can enhance simplicity, power, robustness

Object Duplication Example

```
public class Person {
  private final Date birthDate;
  public Person(Date birthDate){
    this.birthDate = birthDate;
  }
  // UNNECESSARY OBJECT CREATION
  public boolean bornBefore2000(){
    Calendar gmtCal = Calendar.getInstance(
      TimeZone.getTimeZone("GMT"));
    gmtCal.set(2000,Calendar.JANUARY,1,0,0,0);
    Date MILLENIUM = gmtCal.getTime();
    return birthDate.before(MILLENIUM);
  }
}
```

Object Duplication Example

```
public class Person {
  …
  // STATIC INITIALIZATION CREATES OBJECT ONCE
  private static final Date MILLENIUM;
  static {
    Calendar gmtCal = Calendar.getInstance(
      TimeZone.getTimeZone("GMT"));
    gmtCal.set(2000,Calendar.JANUARY,1,0,0,0);
    Date MILLENIUM = gmtCal.getTime();
  }
  public boolean bornBefore2000(){ // FASTER!
    return birthDate.before(MILLENIUM);
  }
}
```

Immutable Classes

- Class whose instances cannot be modified
- Examples
  - String
  - Integer
  - BigInteger
- How, why, and when to use them

How to Write an Immutable Class

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure exclusive access to any mutable components

Immutable Fval Class Example

```
public final class Fval {
  private final float f;
  public Fval(float f) {
    this.f = f;
  }
  // ACCESSORS WITHOUT CORRESPONDING MUTATORS
  public float value( ) { return f; }

  // ALL OPERATIONS RETURN NEW Fval
  public Fval add(Fval x) {
    return new Fval(f + x.f);
  }
  // SUBTRACT, MULTIPLY, ETC. SIMILAR TO ADD
```

Immutable Float Example (cont.)

```
public boolean equals(Object o) {
   if (o == this) return true;
   if (!(o instanceof Fval))
     return false;
   Fval c = (Fval) o;
   return (Float.floatToIntBits(f) ==
           Float.floatToIntBits(c.f));
}
```

Distinguishing Characteristic

- Return new instance instead of modifying
- Functional programming
- May seem unnatural at first
- Many advantages

Advantage 1 – Simplicity

- Instances have exactly one state
- Easy to design, implement
- Constructors establish invariants
- Invariants can never be corrupted
- Requires no effort on the part of clients

Advantage 2 – Inherently Thread-Safe

- No need for synchronization
  - Internal or external
  - Since no writes to shared data
- Can’t be corrupted by concurrent access
- By far the easiest approach to thread safety

Advantage 3 – Can Be Shared Freely

```
// EXPORTED CONSTANTS
public static final Fval ZERO = new Fval(0);
public static final Fval ONE  = new Fval(1);
// STATIC FACTORY CAN CACHE COMMON VALUES
public static Fval valueOf(float f) { ...
}
// PRIVATE CONSTRUCTOR MAKES FACTORY MANDATORY
private Fval (float f) {
    this.f = f;
}
```

Advantage 4 – No Copies

- No need for defensive copies
- No need for any copies at all!
- No need for clone or copy constructor
- Not well understood in the early days
  - public String(String s); // Should not exist

Advantage 5 – Composability

- Excellent building blocks
- Easier to maintain invariants
  - If component objects won't change
- Special cases
  - Map keys
  - Set elements

The Major Disadvantage

- Separate instance for each distinct value
- Creating these instances can be costly

```
BigInteger moby = ...;  // A million bits
moby = moby.flipBit(0); // Ouch!
```

- Problem magnified for multistep operations
  - Provide common multistep operations as primitives
  - Alternatively provide mutable companion class

When to Make Classes Immutable

- Always, unless there's a good reason not to
- Always make small “value classes” immutable
  - Examples
    - Color
    - PhoneNumber
    - Price
  - Date and Point (both mutable) were mistakes!
  - Experts often use long instead of Date

When to Make Classes Mutable

- Class represents entity whose state changes
  - Real-world
    - BankAccount, TrafficLight
  - Abstract
    - Iterator, Matcher, Collection
  - Process classes
    - Thread, Timer
- If class must be mutable, minimize mutability
  - Constructors should fully initialize instance
  - Avoid reinitialize methods

Common Errors

- See

http://www.cs.umd.edu/class/spring2012/cmsc132/resources/CommonErrors.html