

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
 - 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 overloads 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 0.8999999999999999. 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

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
- Prefer interfaces to abstract classes

Methods

- Check parameters for validity
- Make defensive copies when needed (more about this topic later on)
- 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
- 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
- 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

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));  
}
```

Advantage 1 – Simplicity

- Instances have exactly one state
- Constructors establish invariants
- Invariants can never be corrupted

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

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

Defensive Copying

- Java programming language is **safe**
 - Immune to buffer overruns, wild pointers, etc...
 - Unlike C, C++
- Makes it possible to write **robust** classes
 - Correctness doesn't depend on other modules
 - Even in safe language, it requires effort
- **Defensive Programming**
 - Assume clients will try to destroy **invariants**
 - May actually be true
 - More likely – honest mistakes
 - Ensure class invariants survive any inputs

Defensive Copying

- The following class is not robust!

```
// GOAL - PERSON'S BIRTHDAY IS INVARIANT
public class Person {
    // PROTECTS birthDate FROM MODIFICATION????
    private final Date birthDate;
    public Person(Date birthDate){
        this.birthDate = birthDate;
    }
    public Date bday() { return birthDate; }
}
```

- Problem #1: Constructor can allow invariant to be modified

```
// ATTACK INTERNALS OF PERSON
Date today = new Date();
Person p = new Person(today);
today.setYear(78); // MODIFIES P'S BIRTHDAY!
```

Defensive Copying

- Problem #2: Accessor can allow invariant to be modified

```
// ACCESSOR ATTACK ON INTERNALS OF PERSON
Date today = new Date();
Person p = new Person(today);
Date bday = p.bday();
bday.setYear(78); // MODIFIES P'S BIRTHDAY!
```

- Solution

- Defensive copying in constructors and accessors

```
public class Person {
    private final Date birthDate;
    // REPAIRED CONSTRUCTOR
    // DEFENSIVELY COPIES PARAMETERS
    public Person(Date birthDate){
        this.birthDate =
            new Date(birthDate.getTime());
    }
    // REPAIRED ACCESSOR DEFENSIVELY COPY FIELDS
    public Date bday() { (Date) birthDate.clone(); }
}
```

Defensive Copying Summary

- Don't incorporate mutable parameters into object
 - Make defensive copies
- Return defensive copies of mutable fields
 - Accessors
- **Important**
 - First copy parameters, then check copy validity
 - Eliminate window of vulnerability...
 - ...between parameter check and copy
 - Thwarts multithreaded attack
- Use of immutable components eliminates the need for defensive copying

Common Errors

- See “Frequently Seen Java Errors” in Resources section of the class web page