



**JavaOne**<sup>SM</sup>

Sun's 2002 Worldwide Java Developer Conference™

# More Effective Programming With Java™ Technology

**Joshua Bloch**

Senior Staff Engineer

Sun Microsystems, Java Software

# Presentation Goal

Learn to use the Java™ programming language and its libraries more effectively.

# View From 10,000 feet

- This talk consists of
  - Patterns and idioms to emulate
  - Pitfalls to avoid
- Excerpted from *Effective Java™ Programming Language Guide*
  - (Addison-Wesley, June 2001)



# Speaker's Qualifications

- Author, *Effective Java Programming Language Guide* (Addison-Wesley, June 2001)
- Architect
  - Java™ Collections Framework
  - `java.math`
  - `java.util.prefs`
  - the `assert` construct
  - Chained exceptions, `StackTraceElement`
  - `ThreadLocal`, `Timer`, etc.



# Topics Covered in the Book

- The language
- Libraries: `java.lang`, `java.util`, (`java.io`)
- Not covered:  
GUI Programming, Enterprise APIs
- Ten chapters, fifty-seven items

# Topics Covered in the Talk

- I. Duplicate Object Creation (Item 4)
- II. Defensive Copying (Item 24)
- III. Immutable Classes (Item 13)

# I. Avoid Duplicate Object Creation

- Reuse existing object instead
- Simplest example

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

```
String s = "Do this instead";
```

- In loops, savings can be substantial



# Another Example

```
public class Person {
    private final Date birthDate; // Other fields omitted
    public Person(Date birthDate){
        this.birthDate = birthDate;
    }

    // UNNECESSARY OBJECT CREATION - DON'T DO THIS!
    public boolean isBabyBoomer(){ //
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
            birthDate.compareTo(boomEnd) < 0;
    }
}
```

# Another Example (Cont.)

```
// Do This Instead - Clearer and faster
private static final Date BOOM_START;
private static final Date BOOM_END;
static {
    Calendar gmtCal =
        Calendar.getInstance(TimeZone.getTimeZone("GMT"));
    gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
    BOOM_START = gmtCal.getTime();
    gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
    BOOM_END = gmtCal.getTime();
}

public boolean isBabyBoomer(){ // ~100 times faster!
    return birthDate.compareTo(BOOM_START) >= 0 &&
        birthDate.compareTo(BOOM_END) < 0;
}
```



# Summary

- Don't create unnecessary duplicate objects
  - Reuse improves clarity and performance
- But don't be afraid to create objects
  - Object creation is cheap on modern Vms
  - Can enhance simplicity, power, robustness

## II. Defensive Copying

- Unlike C/C++, Java programming language *safe*
  - Immune to buffer overruns, wild pointers, etc.
- Makes it possible to write *robust* classes
  - Correctness doesn't depend on other modules
  - Even in safe language, 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

# This Class Is *Not* Robust!

```
public final class Period {
    private final Date start, end; // Invariant: start <= end

    /**
     * @throws IllegalArgumentException if start > end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + ">" + end);
        this.start = start;
        this.end    = end;
    }

    public Date start() { return start; }
    public Date end()   { return end; }
    ... // Remainder omitted
}
```

# The Problem: Date Is Mutable

```
// Attack the internals of a Period instance
Date start = new Date(); // (The current time)
Date end   = new Date(); // " " "
Period p = new Period(start, end);
end.setYear(78); // Modifies internals of p!
```

# The Solution: *Defensive Copying*

```
// Repaired constructor - defensively copies parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end    = new Date( end.getTime());
    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(start + ">" + end);
}
```

# An Important Detail

- Copies made *before* checking parameters
- Validity check performed on copies
- Eliminate *window of vulnerability* between parameter check and copy
- Thwarts multithreaded attack

```
// BROKEN - Permits multithreaded attack!
public Period(Date start, Date end) {
    if (start.compareTo(end) > 0)
        throw new IllegalArgumentException(start + ">" + end);
    // Window of vulnerability
    this.start = new Date(start.getTime());
    this.end   = new Date( end.getTime());
}
```



# Another Important Detail

- Used constructor, not `clone`, to make copies
- Necessary because `Date` class is nonfinal
- Attacker could implement *malicious subclass*
  - Records reference to each instance in list
  - Provides attacker with access to instance list

# Unfortunately, Constructors Are Only Half the Battle

```
// Accessor attack on internals of Period
Date start = new Date();
Date end   = new Date();
Period p = new Period(start, end);
p.end.setYear(78); // Modifies internals of p!
```

# The Solution: More Defensive Copying

```
// Repaired accessors - defensively copy fields
public Date start() {
    return (Date) start.clone(); // (clone OK)
}
public Date end() {
    return (Date) end.clone(); // " "
}
```

**Now Period class is robust!**



# Summary

- Don't incorporate mutable parameters into object; make defensive copies
  - Constructors, static factories, pseudo-constructors, mutators
- Return defensive copies of mutable fields
  - Accessors
- Real lesson—use immutable components
  - Eliminates the need for defensive copying

# III. 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

# Example

```
public final class Complex {
    private final float re, im;

    public Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    // Accessors without corresponding mutators
    public float realPart()      { return re; }
    public float imaginaryPart() { return im; }

    // subtract, multiply, divide similar to add
    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
}
```

# Example (Cont.)

```
public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof Complex)) return false;
    Complex c = (Complex)o;
    return (Float.floatToIntBits(re) ==
            Float.floatToIntBits(c.re)) &&
           (Float.floatToIntBits(im) ==
            Float.floatToIntBits(c.im));
}

public int hashCode() {
    int result = 17 + Float.floatToIntBits(re);
    result = 37*result + Float.floatToIntBits(im);
    return result;
}

public String toString() {
    return "(" + re + " + " + im + "i)";
}
}
```

# 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
- 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 Complex ZERO = new Complex(0, 0);
public static final Complex ONE  = new Complex(1, 0);
public static final Complex I    = new Complex(0, 1);

// Static factory caches commonly requested values
public static Complex valueOf(float re, float im) {
    ...
}
private Complex(float re, float im) {
    this.re = re;
    this.im = im;
}
```



# Advantage 4: 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 case - **Map** keys and **Set** elements

# The Major Disadvantage

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

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

- Problem magnified for multistep operations
  - Well-designed immutable classes provide common multistep operations as primitives
- Alternative: 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` were mistakes!

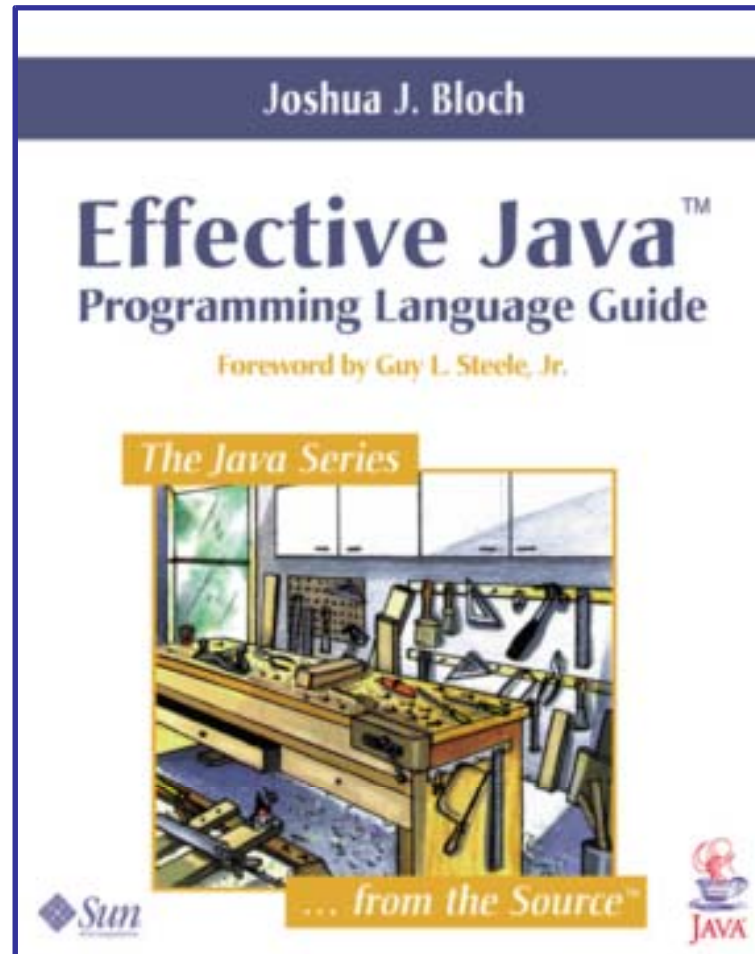
# When to Make Classes *Mutable*

- 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

# Conclusion

- Reuse objects where appropriate
  - Improves clarity and performance
- Make defensive copies where required
  - Provides robustness
- Write immutable classes
  - Simple, thread-safe, sharable and reusable

# For (Much) More Information



# Q&A



**JavaOne**<sup>SM</sup>

Sun's 2002 Worldwide Java Developer Conference™

**BEYOND**  
BOUNDARIES