Effective Programming With Java™ Technology

Joshua Bloch
Senior Staff Engineer,
Libraries Architect
Sun Microsystems, Inc.
Introduction

• Effective use of language and libraries
• Patterns and idioms to emulate
• Traps and pitfalls to avoid
• Excerpted from book:

*Effective Java™ Programming Language Guide* (Addison-Wesley, June 2001)
Topics Covered in Book

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

I. Static factory
II. Singleton
III. Utility class
IV. Typesafe enum
V. Broken random number generation idiom
I. Static Factory

• Normal way to create objects—constructor
• Alternative—*static factory*
• A static method that returns instance
Static Factory Example

Normal Constructor

```java
    public Boolean(boolean b) {
        this.value = b;
    }
```

Static Factory

```java
    public static Boolean valueOf(boolean b) {
        return (b ? Boolean.TRUE : Boolean.FALSE);
    }
```
Static Factory Advantage #1

- Need not create a new object
  - Reuse immutable objects
    - Cache frequently requested values
    - Improves performance
  - Allows you to regulate instances
Static Factory Advantage #2

- Static factories have names
  - Multiple factories with same parameter types

```java
static Complex valueOf(float re,
           float im)
```

```java
static Complex valueOfPolar(float r,
           float theta)
```
Static Factory Advantage #3

- Flexibility to return object of any subtype
  - Separation between interface and implementation
    - Returned class needn’t be public
      - Leads to compact APIs
      - Reduces conceptual weight as well as bulk
  - Can change from release to release
  - Can change at runtime
    - Example—provider framework
Static Factory Disadvantages

• Cannot be subclassed
  – Blessing in disguise?
• Not easily distinguished from other methods
  – Naming conventions help
    • `valueOf(args)`
    • `getInstance(args)`
When to Use a Static Factory

• When there is no need to subclass...
• And one or more of the following
  – Significant performance advantage exists
  – Need flexibility to return different types
  – Can avoid making classes public
  – Need control over instances
II. Singleton

• Class that can have only one instance
  – Represent something intrinsically unique
  – Examples—video display, file system
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    ... // Remainder omitted
}

Singleton Example
Singleton Details

• No accessible constructors
  – Private constructor called only once
  – Guarantees “monoelvistic” universe
  – Static factory can be substituted for field

• Serialization demands care
/**
 * Return the one true Elvis and let the
 * garbage collector take care of the
 * Elvis impersonator.
 */

private Object readResolve() throws ObjectStreamException {
    return INSTANCE;
}
III. Utility Class

• Grouping of static methods, fields
  – Not designed to be instantiated
  – Examples: `java.lang.Math`, `java.util.Collections`
  – Not sufficient to omit all constructors
    • Compiler provides default constructor
public class UtilityClass {
   // Suppresses default constructor
   private UtilityClass() {
      throw new Error("Can't happen.");
   }

   ... // Remainder omitted
}
public class PlayingCard {
    public static final int SUIT_CLUBS    = 0;
    public static final int SUIT_DIAMONDS = 1;
    public static final int SUIT_HEARTS   = 2;
    public static final int SUIT_SPADES   = 3;

    ... // Remainder omitted
}

IV. Traditional int Enum Pattern
Disadvantages of `int` Enums

- Not typesafe
- No namespace—must prefix constant names
- Brittle—constants compiled into clients
- Multiple parties can’t extend independently
- Printed values uninformative
Typesafe Enum Pattern

- Class with instances that represent enum constants
  - Don’t provide any public constructors
  - Public static final field for each constant
Typesafe Enum—Basic Form

```java
public class Suit {
    private final String name;
    public String toString() { return name; }

    private Suit(String name) { this.name = name; }

    public static final Suit CLUBS = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS = new Suit("hearts");
    public static final Suit SPADES = new Suit("spades");
}
```
Fixes All int Enum Disadvantages

• Typesafe
• Provides namespace
• Constants aren’t compiled into clients
• Multiple parties can extend independently
  – Must make constructor protected
• Printed values informative
More Typesafe Enum Advantages

- Can add arbitrary methods
- Can implement interfaces
- Can become full-fledged class over time
- Equality is the same as identity
- Performance comparable to `int` enum
private static final Suit[] VALS =
{
    CLUBS, DIAMONDS, HEARTS, SPADES
};

public static final List VALUES =
    Collections.unmodifiableList(Arrays.asList(VALS));

• Allows iteration

    for (Iterator i = Suit.VALUES.iterator(); i.hasNext(); )
        f((Suit) i.next());
public class Suit implements Comparable {
    ... // Everything from basic form

    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;

    public int compareTo(Object o) {
        return ordinal - ((Suit)o).ordinal;
    }
}
Making a Typesafe Enum Serializable

- Requires ordinal form
- All fields except ordinal are transient
- Requires `readResolve` method:

```java
private Object readResolve() {
    // Canonicalize
    return VALS[ordinal];
}
```
A Few Caveats

- Serializable and extensible variants compatible
  - Must have separate ordinal in each subclass
- Serializable, comparable variants *incompatible*
  - Can’t establish order among subclasses
- Can’t use typesafe enum in switch statement…
  - But you generally shouldn’t switch on enums
V. Common Idiom for Random Number Generation

```java
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

- Deeply flawed
public static void main(String[] args) {
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i=0; i<1000000; i++)
        if (random(n) < n/2)
            low++;
    System.out.println(low);
}
Surprising Answer: ~666,666

- Mapping is unfair
- Example
  - Assume 2 bits in word, n == 3
  - 0 is twice as likely as 1 or 2!
  - If $n \neq 2^k$, distribution is uneven
Can (Rarely) Fail Catastrophically

- Suppose `rnd.nextInt()` returns `Integer.MIN_VALUE`
- `Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE`
- `random(n)` returns negative number if \((n \neq 2^k)\)
- Program failure likely
- Difficult to reproduce
Period Can Be Short

- If \( n = 2^k \), random returns \( k \) low order bits
- Known defect of linear congruential PRNG
- Low order bit has period \( 2^{18} \) (262,144)
- Can repeat itself every few seconds!
Common Idiom Has 3 Flaws

- **Solution: use** `Random.nextInt(int)`
  - Corrects all three flaws of common idiom
  - Documentation guarantees it
  - In platform since release 1.2

- **Moral: Know and use the libraries**
  - New facilities added each release
  - It pays to keep up
Summary

• Use these patterns
  – Static factory
  – Singleton
  – Utility class
  – Typesafe enum

• Use this library method
  – `Random.nextInt(int)`
For (Much) More Information

• *Effective Java™ Programming Language Guide* (Addison-Wesley, 2001)
  – Available at show
  – Bookstores in two weeks

• Two chapters on web
  – “Methods Common to All Objects”
  – “Substitutes for C Constructs”
TS-2425, Effective Programming with Java™ Technology
JavaOne™
Sun's 2001 Worldwide Java Developer Conference®