



JavaOne™

java.sun.com/javaone

More *Effective Java*™

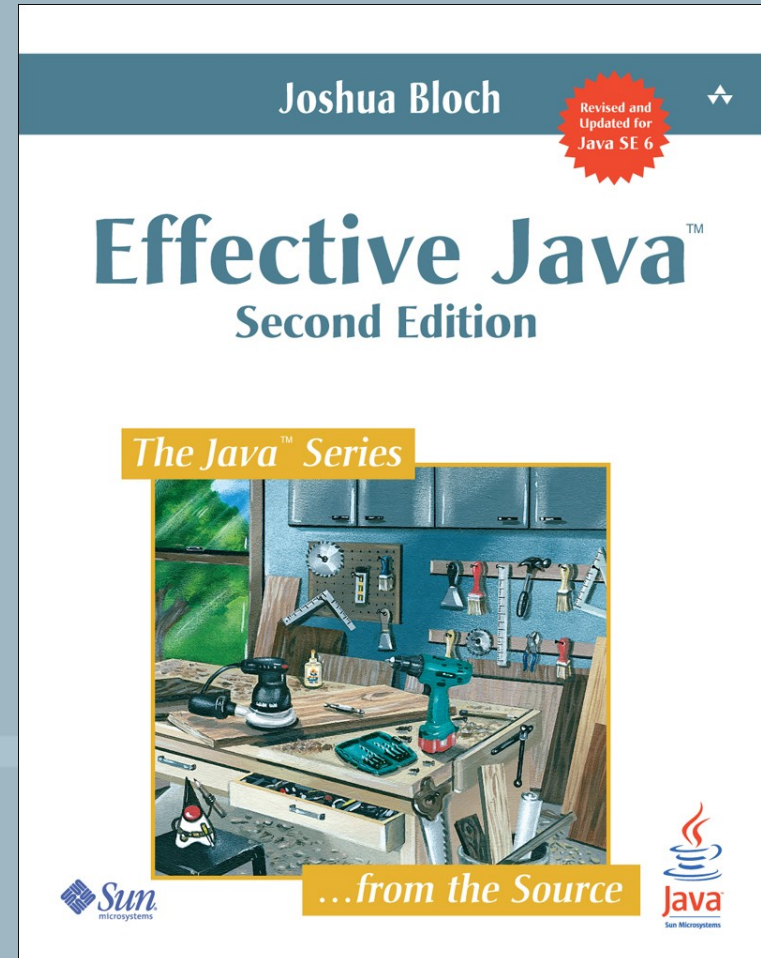
Joshua Bloch, Chief Java Architect, Google Inc.

TS-6623



The wait is over!

GOAL



What's New?

- Chapter 5: *Generics*
- Chapter 6: *Enums and Annotations*
- One or more items on all other Java™ 5 language features
- *Threads* chapter renamed *Concurrency*
 - Rewritten for `java.util.concurrent`
- All existing items updated to reflect current best practices
- A few items added to reflect newly important patterns
- First edition had 57 items; second has 78

Agenda

- Generics (Item 28)
- Enum types (Items 31–34, 77)
- Lazy initialization (Item 71)

Item 28: Bounded Wildcards for API Flexibility

➤ Generic types are *invariant*

- That is, `List<String>` is not a subtype of `List<Object>`
- Good for compile-time type safety, but inflexible

➤ *Bounded wildcard types* provide additional API flexibility

- `List<String>` is a subtype of `List<? extends Object>`
- `List<Object>` is a subtype of `List<? super String>`

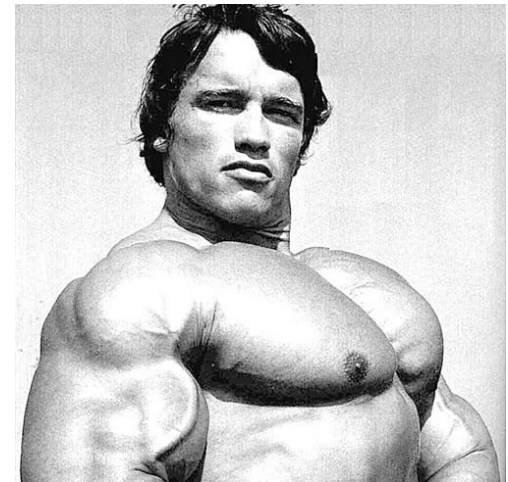
A Mnemonic for Wildcard Usage

➤ PECS — Producer **e**xtends, Consumer **s**uper

- use `Foo<? extends T>` for a `T` producer
- use `Foo<? super T>` for a `T` consumer

➤ Only applies to input parameters

- Don't use wildcard types as return types



Guess who?

PECS in Action (1)

- Suppose you want to add bulk methods to `Stack<E>`
- `void pushAll(Collection<E> src) ;`
- `void popAll(Collection<E> dst) ;`

PECS in Action (1)

- > Suppose you want to add bulk methods to `Stack<E>`
- > `void pushAll(Collection<? extends E> src) ;`
 - `src` is an `E` producer
- > `void popAll(Collection<E> dst) ;`

PECS in Action (1)

- Suppose you want to add bulk methods to `Stack<E>`
- `void pushAll(Collection<? extends E> src) ;`
 - `src` is an `E` producer
- `void popAll(Collection<? super E> dst) ;`
 - `dst` is an `E` consumer

PECS in Action (1)

- Suppose you want to add bulk methods to `Stack<E>`
- `void pushAll(Collection<? extends E> src) ;`
- `void popAll(Collection<? super E> dst) ;`
- User can `pushAll` from a `Collection<Long>` or a `Collection<Number>` onto a `Stack<Number>`
- User can `popAll` into a `Collection<Object>` or a `Collection<Number>` from a `Stack<Number>`

PECS in Action (2)

➤ Consider this generic method:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

PECS in Action (2)

- Consider this generic method:

```
public static <E> Set<E> union(Set<? extends E> s1,  
                               Set<? extends E> s2)
```

- Both `s1` and `s2` are **E producers**
- No wildcard type for return value
 - Wouldn't make the API any more flexible
 - Would force user to deal with wildcard types explicitly
 - **User should not have to think about wildcards to use your API**

Agenda

- Generics (Items 28)
- Enum types (Items 31–34, 77)
- Lazy initialization (Item 71)

Item 31: How would you implement this

```
public enum Ensemble {  
    SOLO, DUET, TRIO, QUARTET, QUINTET,  
    SEXTET, SEPTET, OCTET, NONET, DECTET;  
  
    public int numberOfMusicians() {  
        ???  
    }  
}
```

A common but flawed approach

```
public enum Ensemble {  
    SOLO, DUET, TRIO, QUARTET, QUINTET,  
    SEXTET, SEPTET, OCTET, NONET, DECTET;  
  
    public int numberOfMusicians() {  
        return ordinal() + 1;  
    }  
}
```

What's Wrong With This Usage?

- It's a maintenance nightmare
 - If you (or someone else) reorder constants, program breaks silently
- Can't add multiple constants with same `int` value
 - A *double quartet* is 8 musicians, just like an octet
- Awkward to add constants out of sequence
 - A *triple quartet* is 12 musicians, but there's no term for 11

The Solution—Store `int` in an Instance Field

```
public enum Ensemble {  
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),  
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),  
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);  
  
    private final int numberOfMusicians;  
    Ensemble(int size) {  
        numberOfMusicians = size;  
    }  
  
    public int numberOfMusicians() {  
        return numberOfMusicians;  
    }  
}
```

Item 32: Bit Fields are Obsolete

```
public class Text {  
    public static final int STYLE_BOLD          = 1;  
    public static final int STYLE_ITALIC        = 2;  
    public static final int STYLE_UNDERLINE     = 4;  
    public static final int STYLE_STRIKETHROUGH = 8;  
  
    // Param is bitwise OR of 0 or more STYLE_ values  
    public void applyStyles(int styles) { ... }  
}
```

All the Problems of `int` Constants and More

- Bit fields are not typesafe
- No namespace—must prefix constant names
- Brittle—constants compiled into clients
- Printed values are cryptic
- No easy way to iterate over elements represented by bit field
- If number of constants grows beyond 32, you are toast.
Beyond 64, you're burnt toast.

The Solution — EnumSet

A Modern Replacement for Bit Fields

```
public class Text {  
    public enum Style {  
        BOLD, ITALIC, UNDERLINE, STRIKETHROUGH  
    }  
  
    // Any Set could be passed in, but EnumSet is best  
    public void applyStyles(Set<Style> styles) { ... }  
}
```

Client Code

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

EnumSet Combines Safety, Power, Efficiency

- Provides type safety, richness, and interoperability of `Set`
- Internally, each `EnumSet` is represented as a bit vector
 - If underlying enum type has ≤ 64 elements, a single `long`
 - If underlying enum type has > 64 elements, a `long[]`
- Bulk operations implemented with bitwise arithmetic
 - Same as you'd do manually for bit fields
 - Insulates you from the ugliness of manual bit twiddling

Item 33: How would you implement this?

```
public enum Phase {  
    SOLID, LIQUID, GAS;  
  
    public enum Transition {  
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;  
  
        // Returns phase transition from one phase to another  
        public static Transition from(Phase src, Phase dst) {  
            ???  
        }  
    }  
}
```

Another common but flawed approach

```
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

        // Rows indexed by src-ordinal, cols by dst-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null,      MELT,      SUBLIME },
            { FREEZE,   null,      BOIL  },
            { DEPOSIT,  CONDENSE, null  }
        };

        // Returns phase transition from one phase to another
        public static Transition from(Phase src, Phase dst) {
            return TRANSITIONS[src.ordinal()][dst.ordinal()];
        }
    }
}
```

What's Wrong With This Usage?

- Mistakes in transition table cause runtime failures
 - If you're lucky `ArrayIndexOutOfBoundsException` or `NullPointerException`
 - If not, silent erroneous behavior
- Maintenance nightmare
 - Easy to mess up table if you add an enum value
- Size of table is quadratic in the number of phases
- If enum is large, table will not be readable

The Solution—Use a (nested) EnumMap (1)

The Right Way to Associate Data With Enums

```
public enum Phase {  
    SOLID, LIQUID, GAS;  
  
    public enum Transition {  
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),  
        BOIL(LIQUID, GAS),    CONDENSE(GAS, LIQUID),  
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);  
  
        private final Phase src;  
        private final Phase dst;  
  
        Transition(Phase src, Phase dst) {  
            this.src = src;  
            this.dst = dst;  
        }  
    }  
}
```

The Solution—Use a (nested) EnumMap (2)

The Right Way to Associate Data With Enums

```
// Initialize the phase transition map
private static final Map<Phase, Map<Phase, Transition>> m =
    new EnumMap<Phase, Map<Phase, Transition>>(Phase.class);

static {
    // Insert empty map for each src state
    for (Phase p : Phase.values())
        m.put(p, new EnumMap<Phase, Transition>(Phase.class));

    // Insert state transitions
    for (Transition trans : Transition.values())
        m.get(trans.src).put(trans.dst, trans);
}

public static Transition from(Phase src, Phase dst) {
    return m.get(src).get(dst);
}
}
```

Adding Support for the *Plasma* State

> With original approach:

- Add the constant **PLASMA** to **Phase**
- Add **IONIZE**, **DEIONIZE** to **Transition**
- Add 1 new row and 7 new entries to the transition table
- Don't make any mistakes or you'll be sorry (at runtime)

> With **EnumMap** approach:

- Add the constant **PLASMA** to **Phase**
- Add **IONIZE (GAS, PLASMA)**, **DEIONIZE (PLASMA, GAS)**
- That's it! Program initializes table for you

What is the `ordinal` Method Good for?

➤ The `Enum` specification says this:

Most programmers will have no use for the `ordinal` method. It is designed for use by general-purpose enum-based data structures such as `EnumSet` and `EnumMap`.

➤ Unless you are writing such a data structure, don't use it

➤ If you do use `ordinal`:

- Assume only a dense mapping of nonnegative `int` values to enums
- Don't depend on which `int` value is assigned to which enum

Item 77: Pop Quiz: Is This Class a Singleton?

```
public class Elvis implements Serializable {  
    public static final Elvis INSTANCE = new Elvis();  
    private Elvis() { }  
  
    private final String[] favoriteSongs =  
        { "Hound Dog", "Heartbreak Hotel" };  
  
    public void printFavorites() {  
        System.out.println(Arrays.toString(favoriteSongs));  
    }  
  
    private Object readResolve() {  
        return INSTANCE;  
    }  
}
```

Answer: Unfortunately Not

The first edition oversold the power of `readResolve`

- `Elvis` has a nontransient field (`favoriteSongs`)
- Cleverly crafted attack can save reference to deserialized `Elvis` instance when this field is deserialized
 - See `ElvisStealer` for details (Item 77)
- `readResolve` works only if all fields are transient

The Solution—Enum Singleton Pattern

The Right Way to Implement a Serializable Singleton

```
public enum Elvis {  
    INSTANCE;  
  
    private final String[] favoriteSongs =  
        { "Hound Dog", "Heartbreak Hotel" };  
  
    public void printFavorites() {  
        System.out.println(Arrays.toString(favoriteSongs));  
    }  
}
```

Item 34: Coping With a Limitation of Enums

- Enums provide linguistic support for typesafe enum pattern
- All the advantages *, and more
 - Support for `EnumSet` and `EnumMap`
 - Reliable support for serialization
 - Support for `switch` statement
- * But one thing is missing—you can't extend an enum type
 - In most cases, you shouldn't
 - One compelling use case—operation codes

The Solution—Couple Enum With Interface (1)

Emulated Extensible Enum

```
public interface Operation {  
    double apply(double x, double y);  
}
```

```
public enum BasicOperation implements Operation {  
    PLUS    { double apply(double x, double y){ return x + y; } },  
    MINUS   { double apply(double x, double y){ return x - y; } },  
    TIMES   { double apply(double x, double y){ return x * y; } },  
    DIVIDE  { double apply(double x, double y){ return x / y; } };  
}
```

Use **Operation** to represent an operation in APIs

Use **Collection<? extends Operation>** for multiple ops

The Solution—Couple Enum With Interface (2)

Emulated Extendable Enum

```
public enum ExtendedOperation implements Operation {  
    EXP {  
        public double apply(double x, double y) {  
            return Math.pow(x, y);  
        }  
    },  
    REMAINDER {  
        public double apply(double x, double y) {  
            return x % y;  
        }  
    }  
}
```

Enum Summary

- Don't use `ordinal` to store `int` data; use `int` field
- Don't use bit fields; use `EnumSet`
- Don't to use `ordinal` to index arrays; use `EnumMap`
- Don't use `readResolve` for serializable singleton; use `enum`
- Emulate extensible enums with interfaces

Agenda

- Generics (Items 28)
- Enum types (Items 31–34, 77)
- **Lazy initialization (Item 71)**

Item 71: lazy initialization

- Delaying the initialization of a field until its value is needed
- When should you use it?
 - To fix an initialization circularity
 - To solve a performance problem
- Otherwise, **prefer normal initialization**

```
private final FieldType field = computeFieldValue();
```
- What is the best technique for lazy initialization?
 - It depends

To Break an Initialization Circularity, Use a Synchronized Accessor

```
private FieldType field;  
  
synchronized FieldType getField() {  
    if (field == null)  
        field = computeFieldValue();  
    return field;  
}
```

For High-Performance on a Static Field, use the Lazy Initialization Holder Class Idiom

```
private static class FieldHolder {  
    static final FieldType field = computeFieldValue();  
}  
  
static FieldType getField() {  
    return FieldHolder.field;  
}
```

For High-Performance on an Instance Field, use the Double-Check Idiom

```
private volatile FieldType field;

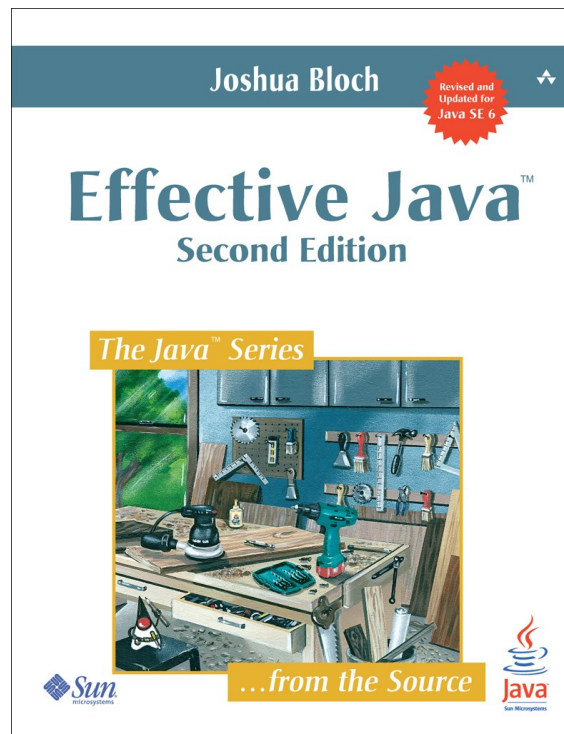
FieldType getField() {
    FieldType result = field;
    if (result == null) {           // 1st check (no lock)
        synchronized(this) {
            result = field;
            if (result == null) // 2nd check (w/ lock)
                field = result = computeFieldValue();
        }
    }
    return result;
}
```


Lazy Initialization Summary

- Your default instinct should be normal (not lazy) initialization
- To break an initialization circularity: synchronized accessor
- For performance on a static field: holder class idiom
- For performance on an instance field: double-check idiom

Shameless Commerce Division

➤ For (much) more information:



➤ I'll be signing copies at the bookstore at **1:00 p.m. today**

THANK YOU

