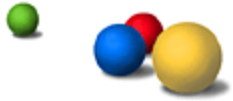
Effective Java™:
Still Effective, After All These Years

Joshua Bloch

Google™



First Edition, 2001; Second Edition, 2008

What's New in the Second Edition?

- Chapter 5: Generics
- Chapter 6: Enums and Annotations
- One or more items on all other Java 5 language features
- Threads chapter renamed Concurrency
  - Completely 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

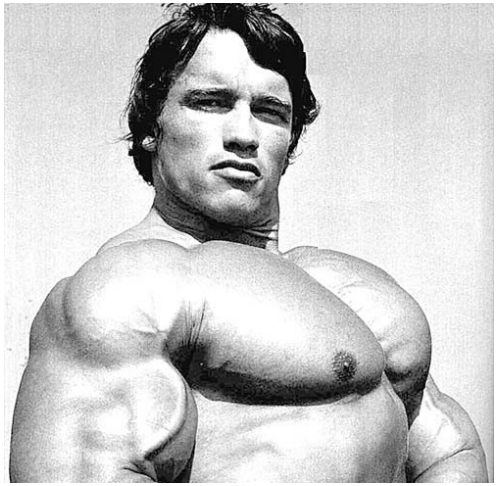
• Generics	Items 28, 29
• Enum types	Item 40
• Varargs	Item 42
• Concurrency	Item 69
• Serialization	Item 78

Item 28: Wildcards for API Flexibility

- Unlike arrays, generic types are *invariant*
  - That is, **`List<String>` is not** a subtype of **`List<Object>`**
  - Good for compile-time type safety, but inflexible
- 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**—**P**roducer **e**xtends, **C**onsumer **s**uper
  - For a T producer, use `Foo<? extends T>`
  - For a T consumer, use `Foo<? super T>`
- Only applies to input parameters
  - Don't use wildcard types as return types



Guess who?

Flex your PECS (1)

- Suppose you want to add bulk methods to **`Stack<E>`**

```
void pushAll(Collection<E> src);

void popAll(Collection<E> dst);
```

Flex your PECS (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);
```

Flex your PECS (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

Flex your PECS (1)

What does it buy you?

```
void pushAll(Collection<? extends E> src);
void popAll(Collection<? super E> dst);
```

- Caller can now `pushAll` from a `Collection<Long>` or a `Collection<Number>` onto a `Stack<Number>`
- Caller can now `popAll` into a `Collection<Object>` or a `Collection<Number>` from a `Stack<Number>`

Flex your PECS (2)

- Consider this generic method:

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

Flex your PECS (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

Flex your PECS (2)

Truth In Advertising – It Doesn't Always "Just Work"

- This code won't compile ☹

```
Set<Integer> ints = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(ints, doubles);
```

- The compiler says

```
Union.java:14: incompatible types
found : Set<Number & Comparable<? extends Number &
            Comparable<?>>>
required: Set<Number>
Set<Number> numbers = union(integers, doubles);
                              ^
```

- The fix – provide an *explicit type parameter*

```
Set<Number> nums = Union.<Number>union(ints, doubles);
```

Summary, in Tabular Form

	Input Parameter Produces `T` Instances?		
Parameter Consumes `T` Instances?		Yes	No
	Yes		**`Foo<? super T>`** (*Contravariant* in `T`)
	No	**`Foo<? extends T>`** (*Covariant* in `T`)	

Filling in The Blanks

		Parameter Produces `T` Instances?	
		Yes	No
Parameter Consumes `T` Instances?	Yes	**`Foo<T>`** (*Invariant* in `T`)	**`Foo<? super T>`** (*Contravariant* in `T`)
	No	**`Foo<? extends T>`** (*Covariant* in `T`)	**`Foo<?>`** (*Independent* of `T`)

Item 29: How to Write A Container With an Arbitrary Number of Type Parameters

- Typically, containers are parameterized
  - For example: `Set<E>`, `Map<K, V>`
  - Limits you to a fixed number of type parameters
- Sometimes you need more flexibility
  - Consider a `DatabaseRow` class
  - You need one type parameter for each column
  - Number of columns varies from instance to instance

The Solution: Typesafe Heterogeneous Container Pattern

- Parameterize *selector* instead of container
  - For **`DatabaseRow`**, **`DatabaseColumn`** is selector
- Present selector to container to get data
- Data is strongly typed at compile time
- Allows for unlimited type parameters

Example: A Favorites Database

API and Client

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}

// Typesafe heterogeneous container pattern - client
public static void main(String[] args) {
    Favorites f = new Favorites();
    f.setFavorite(String.class, "Java");
    f.setFavorite(Integer.class, 0xcafebabe);
    f.putFavorite(Class.class, ThreadLocal.class);

    String s = f.getFavorite(String.class);
    int i = f.getFavorite(Integer.class);
    Class<?> favoriteClass = f.getFavorite(Class.class);
    System.out.println("printf("%s %x %s%n",
        favoriteString, favoriteInteger, favoriteClass);
}
```

Example: A Favorites Database

Implementation

```
public class Favorites {
    private Map<Class<?>, Object> favorites =
        new HashMap<Class<?>, Object>();

    public <T> void putFavorite(Class<T> type, T instance) {
        if (type == null)
            throw new NullPointerException("Type is null");
        favorites.put(type, instance);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

Agenda

• Generics	Items 28, 29
• Enum types	Item 40
• Varargs	Item 42
• Concurrency	Item 69
• Serialization	Item 78

Item 40: Prefer 2-element enums to booleans

- Which would you rather see in code, this:

```
double temp = thermometer.getTemp(true);
```

- or this:

```
double temp = thermometer.getTemp(TemperatureScale.FAHRENHEIT);
```

- With static import, you can even have this:

```
double temp = thermometer.getTemp(FAHRENHEIT);
```

Advantages of 2-Element enums Over booleans

- Code is easier to read
- Code is easier to write (especially with IDE)
- Less need to consult documentation
- Smaller probability of error
- Much better for API evolution

Evolution of a 2-Element enum

- Version 1

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

- Version 2

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS, KELVIN }
```

- Version 3

```
public enum TemperatureScale {
    FAHRENHEIT, CELSIUS, KELVIN;
    double toCelsius(double temp);
}
```

Agenda

• Generics	Items 28, 29
• Enum types	Item 40
• Varargs	Item 42
• Concurrency	Item 69
• Serialization	Item 78

Item 42: Two Useful Idioms for Varargs

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

Suppose You Want to Require at Least One Argument

```
// The WRONG way to require one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException(
            "Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

Fails **at runtime** if invoked with no arguments
It's ugly – explicit validity check on number of args
Interacts poorly with for-each loop

The Right Way

```
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

Won't compile if you try to invoke with 1 argument
No validity check necessary
Works great with for-each loop

Varargs when Performance is Critical

```
// These static factories are real
Class EnumSet<E extends Enum<E>> {
    static <E> EnumSet<E> of(E e);
    static <E> EnumSet<E> of(E e1, E e2)
    static <E> EnumSet<E> of(E e1, E e2, E e3)
    static <E> EnumSet<E> of(E e1, E e2, E e3, E e4)
    static <E> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5);
    static <E> EnumSet<E> of(E first, E... rest)
    ... // Remainder omitted
}
```

Avoids cost of array allocation if fewer that *n* args

Agenda

• Generics	Items 28, 29
• Enum types	Item 40
• Varargs	Item 42
• Concurrency	Item 69
• Serialization	Item 78

Item 69: Use ConcurrentHashMap

But Use it Right!

- Concurrent collections manage synchronization internally
  - Lock striping, non-blocking algorithms, etc.
- Combines high concurrency and performance
- Synchronized collections nearly obsolete
- Use **`ConcurrentHashMap`**, not **`Collections.synchronizedMap()`**

With Concurrent Collections, You Can't Combine Operations Atomically

```
private static final ConcurrentMap<String, String> map =
    new ConcurrentHashMap<String, String>();

// Interning map atop ConcurrentMap -- BROKEN!
public static String intern(String s) {
    synchronized(map) { // ALWAYS wrong!
        String result = map.get(s);
        if (result == null) {
            map.put(s, s);
            result = s;
        }
        return result;
    }
}
```

You Could Fix it Like This...

```
// Interning map atop ConcurrentMap - works, but slow!
public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

Calls putIfAbsent every time it needs to read a value
Unfortunately, this usage is very common

But This is Much Butter

```
// Interning map atop ConcurrentMap - the right way!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
```

Calls putIfAbsent only if map doesn't contain entry

250% faster on my machine, and far less contention

One More “Solution” That Doesn’t Work

```
// Interning map atop ConcurrentMap - SLOW AND BROKEN
public static String intern(String s) {
    map.putIfAbsent(s, s);  // Ignores return value
    return s;  // Fails if map already contained string!
}
```

This bug is surprisingly common!
We found 15% of **`putIfAbsent`** uses ignore result

Summary

- Synchronized collections are largely obsolete
- Use **`ConcurrentHashMap`** and friends
- **Never** synchronize on a concurrent collection
- Use **`putIfAbsent`** (and friends) properly
  - Only call **`putIfAbsent`** if **`get`** returns null
  - And always check the return value
- API designers: make it easy to do the right thing

Agenda

• Generics	Items 28, 29
• Enum types	Item 40
• Varargs	Item 42
• Concurrency	Item 69
• Serialization	Item 78

Item 74: Serialization is Fraught with Peril

- Implementation details leak into public API
  - Serialized form derived from implementation
- Instances created without invoking constructor
  - Constructors may establish invariants, and instance methods maintain them, yet they can be violated
- Doesn't combine well with final fields
  - You're forced to make them nonfinal or use reflection
- The result: increased maintenance cost, likelihood of bugs, security problems,
- There is a better way!

The Serialization Proxy Pattern

The basic idea is very simple

- Don’t serialize instances of your class; instead, serialize instances of a small, struct-like class class that concisely represents it
- Then reconstitute instances of your class at deserialization time using only its public APIs!

The Serialization Proxy Pattern

Step-by-step (1)

- Design a struct-like proxy class that concisely represents logical state of class to be serialized
- Declare the proxy as a static nested class
- Provide one constructor for the proxy, which takes an instance of the enclosing class
  - No need for consistency checks or defensive copies

The Serialization Proxy Pattern

Step-by-step (2)

- Put **`writeReplace`** method on enclosing class

```
// You can always use exactly this code
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

- Put a **`readResolve`** method on the proxy
  - Use any methods in the public API of the enclosing class to reconstitute the instance

A Real-Life Example

EnumSet's Serialization Proxy

```
private static class SerializationProxy <E extends Enum<E>>
    implements Serializable {
    private final Class<E> elementType;
    private final Enum[] elements;

    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(EMPTY_ENUM_ARRAY);
    }

    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);
        for (Enum e : elements)
            result.add((E)e);
        return result;
    }
    private static final long serialVersionUID = ... ;
}
```

Truth in Advertising

The Serialization Proxy Pattern is not a Panacea

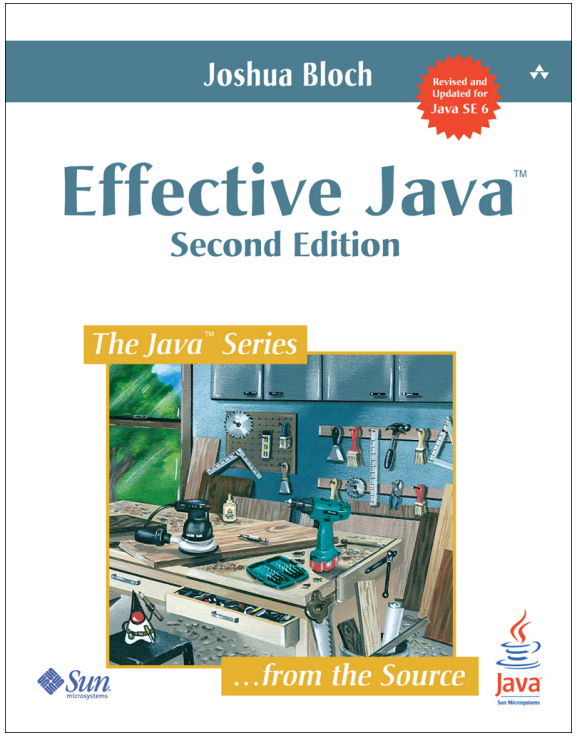
- Incompatible with extendable classes
- Incompatible with some classes whose object graphs contain circularities
- Adds 15% to cost of serialization/deserialization
- **But when it’s applicable, it's the easiest way to robustly serialize complex objects**

Key Ideas to Take Home

- Remember the PECS mnemonic for wildcards
- When a fixed number of type parameters won't do, use a Typesafe Heterogeneous Container
- Prefer two-element enums to booleans
- Never synchronize on a concurrent collection; use **`putIfAbsent`**, and check the return value
- When your plans call for serialization, remember the Serialization Proxy pattern

Shameless Commerce Division

- There's plenty more where that came from!



Effective Java™:
Still Effective, After All These Years

Joshua Bloch

Google™