Effective Java™ Reloaded

Joshua Bloch

Chief Java Architect
Google Inc.

Session 1512

Disclaimer

Effective Java™ Hasn't Yet Been Reloaded,
but I Have Plenty of Ammunition

I have lots of fine new material on making effective use of new platform features, and I'd like to share some of it with you

Menu

Appetizer: *Object Creation*

Main Course: *Generics*

Dessert: *Assorted Sweets*

Menu

Appetizer: *Object Creation*

Main Course: *Generics*

Dessert: *Assorted Sweets*

1. Static Factories Have Advantages Over Constructors (Old News)

- Need not create a new object on each call
- They have names
  - Allows multiple factories with same type signature
- Flexibility to return object of any subtype
- But wait! There's more…

New Static Factory Advantage: They Do Type Inference

- Which Looks Better?
  - ```
    Map<String, List<String>> m =
        new HashMap<String, List<String>>();
    ```
  - ```
    Map<String, List<String>> m = HashMap.newInstance();
    ```
- Regrettably **`HashMap`** has no such method (yet)
  - Until it does, you can write your own utility class
- Your generic classes can and should

2. Static Factories and Constructors Share a Problem

- Ugly when they have many optional parameters
  - ```
    new NutritionFacts(
      String name, int servingSize, int servingsPerCntnr,
      int totalFat, int saturatedFat, int transFat,
      int cholesterol, 15 more optional params!);
    ```
- Telescoping signature pattern is a hack
- But you can't provide all 2^n possibilities
- Beans-style setters are not the answer!
  - They preclude immutable classes
    ```

The Solution: Builder Pattern

- Builder constructor takes all required params
- One setter for each optional parameter
  - Setters return the builder to allow for chaining
- One method to generate instance
- Pattern emulates named optional parameters!

```
NutritionFacts twoLiterDietCoke =
  new NutritionFacts.Builder("Diet Coke", 240, 8).sodium(1).build();
```

Builder Example

```
public class NutritionFacts {
    public static class Builder {
        public Builder(String name, int servingSize,
                int servingsPerContainer) { ... }

            public Builder totalFat(int val) { ... }
            public Builder saturatedFat(int val) { ... }
            public Builder transFat(int val) { ... }
            public Builder cholesterol(int val) { ... }
            ... // 15 more setters

            public NutritionFacts build() {
                return new NutritionFacts(this);
            }
        }
        private NutritionFacts(Builder builder) { ... }
    }
}
```

An Intriguing Possibility

```
package java.util;

public interface Builder<T> {
    T build();
}
```

Much safer and more powerful than passing **`Class`** objects around and calling **`newInstance()`**

Menu

Appetizer:	*Object Creation*
Main Course:	*Generics*
Dessert:	*Assorted Sweets*

1. Avoid Raw Types in New Code

```
// Generic type: Good
Collection<Coin> coinCollection = new ArrayList<Coin>();
coinCollection.add(new Stamp());  // Won't compile
  ...
for (Coin c : coinCollection) {
  ...
}

// Raw Type: Evil
Collection coinCollection = new ArrayList();
coinCollection.add(new Stamp());  // Succeeds but should not
  ...
for (Object o : coinCollection) {
    Coin c = (Coin) o;            // Throws exception at runtime
  ...
}
```

Don’t Ignore Compiler Warnings

- If you’ve been using generics, you’ve seen lots
- Understand each warning
- Eliminate it if possible
- Otherwise label it with a comment
  - Most common example: **`// Unchecked`**
- Use **`@SuppressWarnings("unchecked")`** if you can prove it is safe

2. Prefer Wildcards to Type Parameters

```
// Generic method with type parameter E
public <E> void removeAll(Collection<E> coll) {
    for (E e : list)
        remove(e);
}

// Method whose parameter uses wildcard type
public void removeAll(Collection<?> coll) {
    for (Object o : coll)
        remove(o);
}
```

The rule: If a type variable appears only once in a method signature, use wildcard instead

The Exception: Conjunctive Types

```
<T extends Serializable & List<?>> void f(T list) { … }
```

Not the same as:

```
interface SerializableList<E> extends Serializable, List<E>;
void f(SerializableList<?> list) { … }
```

The first works with classes outside your control

- **`ArrayList<String>`** matches the conjunctive type but does not implement **`SerializableList`**

3. Use Bounded Wildcards to Increase Applicability of APIs

```
// Method names are from the perspective of customer
public interface Shop<T> {
    T buy();
    void sell(T myItem);
    void buy(int numItems, Collection<T> myItems);
    void sell(Collection<T> myItems);
}

class Model { }
class ModelPlane extends Model { }
class ModelTrain extends Model { }
```

Thanks to Peter Sestoft for shop example

Works Fine if You Stick to One Type

```
// Individual purchase and sale
Shop<ModelPlane> modelPlaneShop = ... ;
ModelPlane myPlane = modelPlaneShop.buy();
modelPlaneShop.sell(myPlane);

// Bulk purchase and sale
Collection<ModelPlane> myPlanes = ... ;
modelPlaneShop.buy(5, myPlanes);
modelPlaneShop.sell(myPlanes);
```

Simple Subtyping Works Fine

```
// You can buy a model from a train shop
Model myModel =  modelTrainShop.buy();

// You can sell a model train to a model shop
modelShop.sell(myTrain);

public interface Shop<T> {
    T buy();
    void sell(T myItem);
    void buy(int numItems, Collection<T> myItems);
    void sell(Collection<T> myItems);
}
```

Collection Subtyping Doesn’t Work!

```
// You can't buy a bunch of models from the train shop
modelTrainShop.buy(5, myModels);  // Won't compile

// You can't sell a bunch of trains to the model shop
modelShop.sell(myTrains);            // Won't compile

public interface Shop<T> {
    T buy();
    void sell(T item);
    void buy(int numItems, Collection<T> myStuff);
    void sell(Collection<T> lot);
}
```

Bounded Wildcards to the Rescue

```
public interface Shop<T> {
    T buy();
    void sell(T item);
    void buy(int numItems, Collection<? super T> myStuff);
    void sell(Collection<? extends T> lot);
}

// You can buy a bunch of models from the train shop
modelTrainShop.buy(5, myModels);  // Compiles

// You can sell your train set to the model shop;
modelShop.sell(myTrains);         // Compiles
```

Basic Rule for Bounded Wildcards

- Use **`extends`** when parameterized instance is producer (“for read”)
- Use **`super`** when parameterized instance is consumer (“for write”)

4. Don't Confuse Bounded Wildcards With Bounded Type Variables

- **Bounded Wildcards**

  `void f(List<? extends Number> list) { … }`
  - **`super`** can be used only in bounded wildcards
  - Bounded wildcards can be used only as type params

- **Bounded Type Variables**

  `<T extends Number> void f(List<T> list) { … }`
  - & can be used only for bounded type variables

Avoid Bounded Wildcards in Return Types

- They force client to deal with wildcards directly
  - Only library designers should have to think about wildcards
- Rarely, you do need to return wildcard type
  - In `java.lang.ref.ReferenceQueue`

```
public Reference<? extends T> remove(long timeout);
```

Wildcards Gone Bad

```
public static <T> List<T> longer(List<T> c1, List<T> c2) {
    return c1.size() >= c2.size() ? c1 : c2;
}

// Don't do this!!! More complex and less powerful
public static List<?> longer(List<?> c1, List<?> c2) {
    return c1.size() >= c2.size() ? c1 : c2;
}
```

Wildcards Gone Bad 2: True Life Stories

- In **`java.util.concurrent.ExecutorService`**

```
public Future<?> submit(Runnable task);
```

  - Intent: to show that **`Future`** always returned **`null`**
  - Result: minor pain for API users
- Correct idiom to indicate unused type parameter

```
public Future<Void> submit(Runnable task);
```

  - Type **`Void`** is non-instantiable
  - Easier to use and clarifies intent

5. Pop Quiz
What's Wrong With This Program?

```
public static void rotate(List<?> list) {
    list.add(list.remove(0));
}
```

Answer
It Won't Compile

```
public static void rotate(List<?> list) {
    list.add(list.remove(0));
}
```

```
Rotate.java:5: cannot find symbol
symbol  : method add(java.lang.Object)
location: interface java.util.List<capture of ?>
        list.add(list.remove(0));
            ^
```

Intuition Behind the Problem

```
public static void rotate(List<?> list) {
    list.add(list.remove(0));
}
```

`remove` and **`add`** are two distinct operations

Invoking each method “captures” the wildcard type

Type system doesn’t know captured types are identical

This Program Really *Is* Unsafe

```
public class Rotate {
    List<?> list;
    Rotate(List<?> list) { this.list = list; }

    public void rotate() {
        list.add(list.remove(0));
    }
    ...
}
```

Another thread could set list field from **`List<Stamp>`** to **`List<Coin>`** between **`remove`** and **`add`**

Solution: Controlled Wildcard-Capture

```
public static void rotate(List<?> list) {
    rotateHelper(list);
}

// Generic helper method captures wildcard once
private static <E> void rotateHelper(List<E> list) {
    list.add(list.remove(0));
}
```

Now both lists have same type: E

6. Generics and Arrays Don’t Mix; Prefer Generics

- Generic array creation error caused by
  - `new T[SIZE]`, `Set<T>[SIZE]`, `List<String>[SIZE]`
- Affects varargs (warning rather than error)
  - `void foo(Class<? extends Thing>... things);`
- Avoid generic arrays; use `List` instead
  - `List<T>`, `List<Set<T>>`, `List<List<String>>`
- Some even say: Avoid arrays altogether

7. Cool Pattern: Typesafe Heterogeneous Container

- Typically, containers are parameterized
  - Limits you to a fixed number of type parameters
- Sometimes you need more flexibility
  - Database rows
  - Type-based publish-subscribe systems
- You can parameterize *selector* instead
  - Present selector to container to get data
  - Data is strongly typed at compile time
  - Effectively allows for unlimited type parameters

Typesafe Heterogeneous Container Example

```
public class Favorites {
    private Map<Class<?>, Object> favorites =
        new HashMap<Class<?>, Object>();
    public <T> void setFavorite(Class<T> klass, T thing) {
        favorites.put(klass, thing);
    }
    public <T> T getFavorite(Class<T> klass) {
        return klass.cast(favorites.get(klass));
    }
    public static void main(String[] args) {
        Favorites f = new Favorites();
        f.setFavorite(String.class, "Java");
        f.setFavorite(Integer.class, 0xcafebabe);
        String s = f.getFavorite(String.class);
        int i = f.getFavorite(Integer.class);
    }
}
```

Generics Summary

- Avoid raw types; Don't ignore compiler warnings
- Prefer wildcards to parameterized methods
- Use bounded wildcards to increase power of APIs
- Use wildcard capture to get a handle on wildcards
- Generics and arrays don't mix; prefer generics
- Use typesafe heterogeneous container pattern
- **Generics aren't that scary once you get to know them. They make your programs better**

Menu

Appetizer:	*Object Creation*
Main Course:	*Generics*
Dessert:	*Assorted Sweets*

1. Use the @Override Annotation *Every Time* You Want to Override

- It's so easy to do this by mistake

```
public class Pair<T1, T2> {
    private final T1 first;  private final T2 second;
    public Pair(T1 first, T2 second) {
        this.first = first;  this.second = second;
    }
    public boolean equals(Pair<T1, T2> p){
        return first.equals(p.first) && second.equals(p.second);
    }
    public int hashCode() {
        return first.hashCode() + 31 * second.hashCode();
    }
}
```

- The penalty is random behavior at runtime
- Diligent use of **@Override** eliminates problem

```
@Override public boolean equals(Pair<T1, T2> p) { // Won't compile
```

2. `final` Is the New `private`

- *Effective Java™* says make all fields **`private`** unless you have reason to do otherwise
- I now believe the same holds true for **`final`**
  - Minimizes mutability
  - Clearly thread-safe—one less thing to worry about
- Blank finals are fine
- So get used to typing **`private final`**
- But watch out for **`readObject`** (and **`clone`**)

3. HashMap Makes a Fine Sparse Array: Just Add Generics and Autoboxing

```
public class SparseArray<T> {
    Map<Integer, T> map = new HashMap<Integer, T>();
    private final T defaultVal;
    public SparseArray(T defaultVal) {
        this.defaultVal = defaultVal;
    }
    public T get(int i) {
        T result = map.get(i);
        return result == null ? defaultVal : result;
    }
    public T put(int i, T val) {
        if (val == defaultVal) {
            T result = map.remove(i);
            return result == null ? defaultVal : result;
        }
        if (val == null) throw new NullPointerException();
        T result = map.put(i, val);
        return result == null ? defaultVal : result;
    }
}
```

Test Program to Exercise SparseArray

```
public static void main(String[] args) {
    SparseArray<Long> a = new SparseArray<Long>(-1L);
    Random rnd = new Random();
    long i = 0, j;  // Indices
    int r;          // Last random number generated
    do {
        r = rnd.nextInt();
        j = a.put(r, ++i);
    } while(j < 0);
    System.out.println("Calls " + i + " & " + j +": " + r);
}
```

4. Cool Pattern: Serialization Proxy

- Default serialized form depends on implementation details
- Even carefully designed serialized forms depend on implementation class
- Serialization builds objects without constructors
- So make a new class representing logical state
  - Use **`writeReplace`** to convert object to proxy
  - Use **`readResolve`** to convert proxy back to object, using only public APIs!

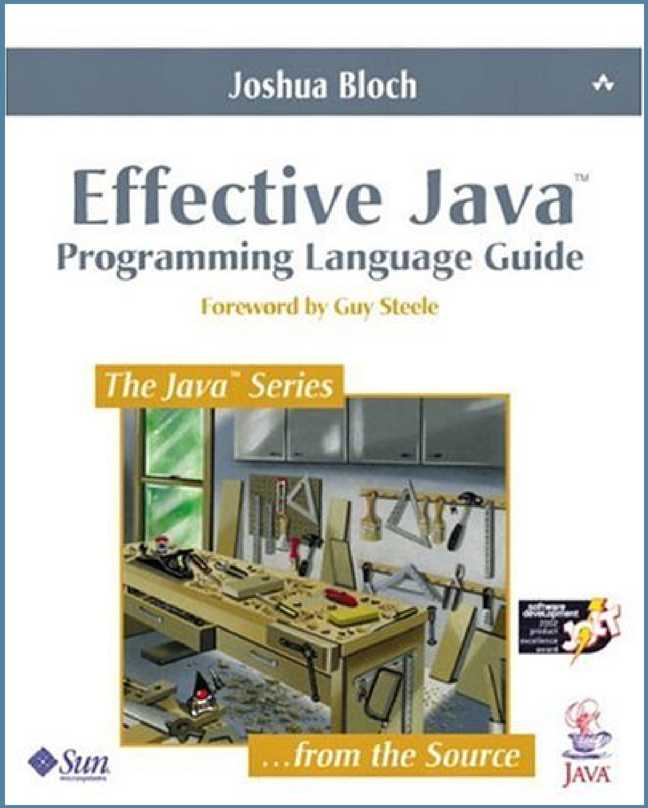
Serialization Proxy Example: EnumSet

```
Object writeReplace() {
    return new Proxy<E>(this);
}
private static class Proxy<E extends Enum<E>>
        implements Serializable {
    private final Class<E> elementType;
    private final Enum[] elements;
    Proxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(ZERO_LENGTH_ENUM_ARRAY);
    }
    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);
        for (Enum e : elements)
            result.add(elementType.cast(e));
        return result;
    }
}
```

Summary

- Release 5 contains many great new features
- We are still figuring out to make best use of them
- This talk contained a sampling of best practices
    - Many areas omitted due to time constrains
- Next year *Effective Java*™ really will be reloaded

Q&A


Joshua Bloch
Effective Java™
Programming Language Guide
Foreword by Guy Steele
The Java™ Series
...from the Source
Sun
JAVA

Effective Java™ Reloaded

Joshua Bloch

Chief Java Architect
Google Inc.

Session 1512