Google™

JavaOne

Effective Java™ Reloaded: This Time ^ It's for Real
Not

Joshua Bloch

Chief Java Architect
Google Inc.

TS-2689

Disclaimer

Effective Java **Still** Hasn't Been Reloaded, but I Do Have Plenty of Ammunition

I am actively at work revising the book. It will be done later this year for sure. This talk has some of the new material.

Topics

Object Creation	(2 Sections)
Generics	(6 Sections)
Miscellania	(2 Sections)

JavaOne

Topics

Object Creation

Generics

Miscellania

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:

```
public static <K, V> HashMap<K, V> newInstance() {
    return new HashMap<K, V>();
}
```

2. Static Factories and Constructors Share a Problem

- Ugly when they have many optional parameters
  - ```
    new NutritionFacts(int servingSize, int servings,
      int calories, int fat, int sodium, int carbohydrate,
      15 more optional params!);
    ```
- Telescoping signature pattern is a hack
  - ```
    NutritionFacts locoCola =
         new NutritionFacts(240, 8, 0, 0, 30, 28);
    ```
- Beans-style setters are not the answer!
  - Allows inconsistency, mandates mutability

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 locoCola =
    new NutritionFacts.Builder(240, 8)
       .sodium(30).carbohydrate(28).build();
```

Builder Implementation Sketch

```
public class NutritionFacts {
    public static class Builder {
        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }

        public Builder calories(int val) {
            calories = val; return this;
        }
        ... // 15 more setters

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder) {
        <copy data from Builder to NutritionFacts>
    }
}
```

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()`**

Topics

Object Creation

Generics

Miscellania

Java
JavaOne

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
- If you **can't** eliminate a warning, suppress them
  `@SuppressWarnings("unchecked")`
  - But limit the scope as much as possible
  - Declare an extra variable if necessary

2. 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 numToBuy, Collection<T> myCollection);
    void sell(Collection<T> myLot);
}

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 numToBuy, Collection<T> myCollection);
    void sell(Collection<T> myLot);
}
```

Collection Subtyping Doesn't Work!

```
// You can't buy a bunch of models from the train shop
modelTrainShop.buy(5, myModelCollection);  // 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 numToBuy, Collection<T> myCollection);
    void sell(Collection<T> myLot);
}
```

Bounded Wildcards to the Rescue

```
public interface Shop<T> {
    T buy();
    void sell(T item);
    void buy(int numToBuy,
             Collection<? super T> myCollection);
    void sell(Collection<? extends T> myLot);
}

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

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

Basic Rule for Bounded Wildcards

- Use `<? extends T>` when parameterized instance is a `T` producer (“for read/input”)
- Use `<? super T>` when parameterized instance is a `T` consumer (“for write/output”)

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

- **Bounded wildcards**

```
void sell(Collection<? extends T> myLot);
```

  - Major use: restrict input parameters
  - Can use `super`

- **Bounded type variables**

```
<T extends Number> T sum(List<T> x) { … }
```

  - Restricts actual type parameter
    - Works for parameterized classes and methods
  - Can't use `super`

There Is a Strong Relationship Between Wildcards and Type Parameters

- You often have the choice between wildcards and type parameters in parameterized methods
- These two signatures have identical semantics
  - `boolean addAll(Collection<? extends E> c);`
  - `<T extends E> boolean addAll(Collection<T> c);`

Prefer Wildcards to Type Parameters in Parameterized Methods

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

It's Usually Best to 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
  - For example, a read-only list of numbers

```
List<? extends Number> operands();
```

Don't Overuse Wildcards

```
// Perfectly good method
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;
}
```

Don't Overuse Wildcards (2)

- 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

JavaOne

4. Pop Quiz: What's Wrong With This Program?

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

Answer: It Won't Compile

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

```
Rotate.java:6: cannot find symbol
symbol   : method add(java.lang.Object)
location: interface java.util.List<capture#503 of ?>
        list.add(list.remove(0));
            ^
```

Intuition Behind the Problem

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

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

Solution: Control Wildcard-Capture

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

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

Now the list and the element have same type: `E`

5. 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

6. 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
- 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);
    }
}
```

But Suppose You Have a Favorite `List<String>` or `List<Integer>`

```
// Won't Compile!
List<String> stooges = Arrays.asList(
    "Larry", "Moe", "Curly");
List<Integer> fibs = Arrays.asList(
    1, 1, 2, 3, 5, 8 );
f.setFavorite(List<String>.class, Stooges);
f.setFavorite(List<Integer>.class, fibs);
String s = f.getFavorite(List<String>.class);
int i = f.getFavorite(List<Integer>.class);
```

Generics use *type erasure*: **`List<String>`** and **`List<Integer>`** have the same class object

The Solution: *Super Type Tokens*

```
import java.lang.reflect.*;

public abstract class TypeRef<T> {
    private final Type type;
    protected TypeRef() {
        ParameterizedType superclass = (ParameterizedType)
            getClass().getGenericSuperclass();
        type = superclass.getActualTypeArguments()[0];
    }
    @Override public boolean equals (Object o) {
        return o instanceof TypeRef &&
            ((TypeRef)o).type.equals(type);
    }
    @Override public int hashCode() {
        return type.hashCode();
    }
}
```

Idea due to Neal Gafter

Typesafe Heterogeneous Container With Super Type Tokens

```
public class Favorites2 {
    private Map<TypeRef<?>, Object> favorites =
        new HashMap< TypeRef<?> , Object>();
    public <T> void setFavorite(TypeRef<T> type, T thing) {
        favorites.put(type, thing);
    }
    @SuppressWarning("unchecked")
    public <T> T getFavorite(TypeRef<T> type) {
        return (T) favorites.get(type);
    }
    public static void main(String[] args) {
        Favorites2 f = new Favorites2();
        List<String> stooges = Arrays.asList(
            "Larry", "Moe", "Curly");
        f.setFavorite(new TypeRef<List<String>>(){}, stooges);
        List<String> ls = f.getFavorite(
            new TypeRef<List<String>>(){});
    }
}
```

Generics Summary

- Avoid raw types; Don’t ignore compiler warnings
- Use bounded wildcards to increase power of APIs
- Understand the relationship between bounded wildcards and bounded type variables
- Generics and arrays don’t mix; prefer generics
- Use typesafe heterogeneous container pattern
- **Generics are tricky, but worth learning. They make your programs better!**

Topics

Object Creation

Generics

Miscellania

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
```

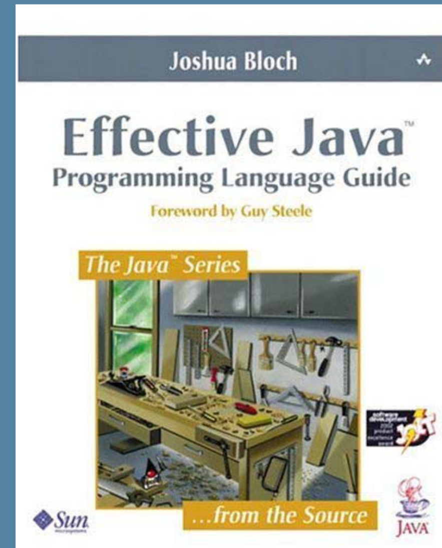
2. `final` Is the New `private`

- *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`**)

Summary

- Releases 5 and 6 contain many 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 constraints
- Next year *Effective Java*™ really will be reloaded
  - I swear

Q&A



Google™

JavaOne

Effective Java™ Reloaded: This Time ^ it's for Real

Not

Joshua Bloch

Chief Java Architect
Google Inc.

TS-2689