Course of Programming in Java

BY ŁUKASZ STAFINIAK

Email: `lukstafi@gmail.com`, `lukstafi@ii.uni.wroc.pl`
Web: `www.ii.uni.wroc.pl/~lukstafi`

Effective Java

BY JOSHUA BLOCH

Selected items

- Item 1: Consider static factory methods instead of constructors
- Item 2: Consider a builder when faced with many constructor parameters
- Item 3: Enforce the singleton property with a private constructor or an enum type
- Item 6: Eliminate obsolete object references
- Item 11: Override clone judiciously
- Item 13: Minimize the accessibility of classes and members
- Item 14: In public classes, use accessor methods, not public fields
- Item 15: Minimize mutability
- Item 16: Favor composition over inheritance
- Item 17: Design and document for inheritance or else prohibit it
- Item 18: Prefer interfaces to abstract classes

- Item 20: Prefer class hierarchies to tagged classes
- Item 21: Use function objects to represent strategies
- Item 22: Favor static member classes over nonstatic
- Item 25: Prefer lists to arrays
- Item 27: Favor generic methods
- Item 32 and 33: Use `EnumSet` and `EnumMap`
- Item 34: Emulate extensible enums with interfaces
- Item 36: Consistently use the `Override` annotation
- Item 38: Check parameters for validity
- Item 39: Make defensive copies when needed
- Item 40: Design method signatures carefully
- Item 41: Use overloading judiciously
- Item 42: Use varargs judiciously

- Item 43: Return empty arrays or collections, not nulls
- Item 44: Write doc comments for all exposed API elements
- Item 45: Minimize the scope of local variables
- Item 46: Prefer for-each loops to traditional for loops
- Item 47: Know and use the libraries
- Item 52: Refer to objects by their interfaces
- Item 67: Avoid excessive synchronization
- Item 68: Prefer executors and tasks to threads
- Item 69: Prefer concurrency utilities to `wait` and `notify`
- Item 70: Document thread safety
- Item 71: Use lazy initialization judiciously
- Item 76: Write `readObject` methods defensively
- Item 78: Consider serialization proxies instead of serialized instances

Item 1: Consider static factory methods instead of constructors

Example:

```java
/** Introduce a new settlement by building its castle, if the
 * settlement already exists return null. */
public static Castle buildCastle (Scheduler scheduler,
                                  Settlement settlement, int x, int y) {
    if (Settlement.castles.containsKey (settlement)) return null;
    Castle castle = new Castle (scheduler, settlement, x, y);
    Settlement.castles.put (settlement, castle);
    return castle;
}
```

1. They have names.
   - For example, the constructor `BigInteger(int, int, Random)`, which returns a `BigInteger` that is probably prime, would have been better expressed as a static factory method named `BigInteger.probablePrime`.
2. They are not required to create a new object each time they are invoked.
   - *Instance control* allows a class to guarantee that it is a *singleton* or *noninstantiable*. Also, it allows an *immutable* class guarantee that no two equal instances exist: `a.equals(b)` if and only if `a==b`.
3. They can return an object of any subtype of their return type.
   - API can return objects without making their classes public. *Interface-based* frameworks: interfaces provide natural return types.
   - Chosing optimal implementation, e.g. `RegularEnumSet` backed by a single long, or `JumboEnumSet` backed by a `long` array.
   - Interfaces cannot have static methods, so by convention, static factory methods for an interface named `Type` are put in a noninstantiable class named `Types`.

4. They reduce the verbosity of creating parameterized type instances.

```
Map<String, List<String>> m =
    new HashMap<String, List<String>>();
```

vs.

```
public static <K, V> HashMap<K, V> newInstance() {
    return new HashMap<K, V>(); }
Map<String, List<String>> m = HashMap.newInstance();
```

Item 2: Consider a builder when faced with many constructor parameters

```
public class Resources {
    public final int wood, stone, coal, coke, ore,
                     iron, gold, fish, flour, bread;
    public static class Builder {
        private int wood = 0, stone = 0, coal = 0, coke = 0, ore = 0,
                    iron = 0, gold = 0, fish = 0, flour = 0, bread = 0;
        public Builder () { } // can have arguments: obligatory fields
        public Builder wood (int val) { wood = val; return this; }
        public Builder stone (int val) { stone = val; return this; }
        public Builder coal (int val) { coal = val; return this; }
        public Builder coke (int val) { coke = val; return this; }
        // ...
        public Resources build() {
            return new Resources (wood, stone, coal, coke, ore,
                                  iron, gold, fish, flour, bread);
        }
    // ...
}
Resources carry = new Resources.Builder().wood(2).coal(1).build();
```

Item 6: Eliminate obsolete object references

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack() { elements = new Object[DEFAULT_INITIAL_CAPACITY]; }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }
    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
        elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

```
// Corrected "memory leak".
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack() { elements = new Object[DEFAULT_INITIAL_CAPACITY]; }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference return result;
        return elements[--size];
    }
    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
        elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

Item 13: Minimize the accessibility of classes and members

- Make each class or member as inaccessible as possible.
- It is acceptable to make a private member of a public class package-private in order to test it.
- Instance fields should never be public.
  - Classes with public mutable fields are not thread-safe.
- It is wrong for a class to have a public static final array field, or an accessor that returns such a field. (Arrays are always mutable.) Use e.g.:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Item 14: In public classes, use accessor methods, not public fields

```java
// Encapsulation of data by accessor methods and mutators
class Point {
    private double x, y;
    public Point(double x, double y) { this.x = x; this.y = y; }
    public double getX() { return x; } // accessor methods (getters)
    public double getY() { return y; }
    public void setX(double x) { this.x = x; } // mutators (setters)
    public void setY(double y) { this.y = y; }
}
```

- If a class is accessible outside package, provide accessors and mutators.
- If a class is package-private or is a private nested class, there is nothing inherently wrong with exposing its data fields.

Item 15: Minimize mutability

To make a class immutable

1. Don't provide any methods that modify the object's state (known as mutators).
2. Ensure that the class cannot be extended.
3. Make all fields final.
4. Make all fields private.
   - Technically permissible for immutable classes to have public final fields containing primitive values or references to immutable objects.
5. Ensure exclusive access to any mutable components (i.e. that clients of the class cannot obtain references to them).
   - Never initialize such a field to a client-provided ob- ject reference or return the object reference from an accessor. Make defensive copies in constructors, accessors, and `readObject` methods.

- Immutable objects are simple. Invariants only checked at construction.
- Immutable objects are inherently thread-safe; they require no synchronization.
- Not only can you share immutable objects, but you can share their internals.
- The only real disadvantage of immutable classes is that they require a separate object for each distinct value.
  - If a multistep operation is provided as a primitive, the immutable class does not have to create a separate object at each step.
    - E.g. `BigInteger` has a package-private mutable *companion class*.
  - The mutable companion class can be made public, e.g. `StringBuilder`.

- One way to prohibit extension is to make all constructors private or package-private, and to add public static factories.

```java
public class Complex {
    private final double re, im;
    private Complex(double re, double im) {
        this.re = re; this.im = im;
    }
    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }
    public static Complex valueOfPolar(double r, double theta) {
        return new Complex(r * Math.cos(theta), r * Math.sin(theta));
    }
    //...
}
```

- Some immutable classes have nonfinal fields in which they cache the results of expensive computations the first time they are needed.

Item 16: Favor composition over inheritance

- The problem is only with *implementation inheritance* (not interfaces).
- Unlike method invocation, inheritance violates encapsulation.

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet<E> extends HashSet<E> {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet() { }
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor); }
    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```

```
InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>();
s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
```

- A related cause of fragility in subclasses is that their superclass can acquire new methods in subsequent releases – they will not guarantee invariants of the subclass.
- Possible name clashes with subsequent releases of the superclass.
- To avoid these problems, use *composition* and *forwarding*.
    - The reusable *forwarding class* can be provided by the library supplying the underlying class.
    - The `InstrumentedSet` class below can even be used to temporarily instrument a set instance that has already been used without instrumentation:

```
static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<Dog>(dogs);
    //... Within this method use iDogs instead of dogs
}
```

```java
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;
    public InstrumentedSet(Set<E> s) {
        super(s);
    }
    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```

```
// Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }
    public void clear() { s.clear(); }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty() { return s.isEmpty(); }
    public int size() { return s.size(); }
    public Iterator<E> iterator() { return s.iterator(); }
    public boolean add(E e) { return s.add(e); }
    public boolean remove(Object o) { return s.remove(o); }
    public boolean containsAll(Collection<?> c) { return s.containsAll(c); }
    public boolean addAll(Collection<? extends E> c) { return s.addAll(c); }
    public boolean removeAll(Collection<?> c) { return s.removeAll(c); }
    public boolean retainAll(Collection<?> c) { return s.retainAll(c); }
    public Object[] toArray() { return s.toArray(); }
    public <T> T[] toArray(T[] a) { return s.toArray(a); }
    @Override public boolean equals(Object o) { return s.equals(o); }
    @Override public int hashCode() { return s.hashCode(); }
    @Override public String toString() { return s.toString(); }
}
```

- Warious names: forwarding class, wrapper class, instrumented class, decorator pattern. Composition + forwarding is also called *delegation*.
- Problem when object passes `this` for callbacks bypassing the wrapper.

Item 17: Design and document for inheritance or else prohibit it

- The class must document its self-use of overridable methods.
  - For each public or protected method or constructor, the documentation must indicate which overridable methods the method or constructor invokes, in what sequence, and how the results of each invocation affect subsequent processing.
- A class may have to provide hooks into its internal workings in the form of protected methods. (E.g. `removeRange` in `AbstractList`.)
- You must test your class by writing subclasses before you release it.
  - Three subclasses are usually sufficient to test an extendable class.
- Constructors must not invoke overridable methods. (Including `clone` and `readObject`.)
- You can eliminate self-use of overridable methods mechanically: Move the body of each overridable method to a private helper method and just invoke it from a non-private method. Only call the helper internally.

Item 18: Prefer interfaces to abstract classes

- Existing classes can be easily retrofitted to implement a new interface.
- Interfaces are ideal for defining mixins.
  - A *mixin* is a type that a class can implement in addition to its primary type to declare that it provides some optional behavior. (E.g. `Comparable`.)
- Interfaces allow the construction of nonhierarchical type frameworks.
- Interfaces enable safe, powerful functionality enhancements via the wrapper class idiom.
- You can combine the virtues of interfaces and abstract classes by providing an abstract *skeletal implementation* class to go with each nontrivial interface that you export.
  - By convention, named `AbstractT` for an interface `T`.
- Drawback: It is far easier to evolve an abstract class than an interface. Once an interface is released and widely implemented, it is almost impossible to change.

Example: a static factory method containing a complete, fully functional List implementation:

```
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(final int[] a) {
    if (a == null) throw new NullPointerException();
    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i]; // Autoboxing (Item 5)
        }
        @Override public Integer set(int i, Integer val) {
            int oldVal = a[i];
            a[i] = val; // Auto-unboxing
            return oldVal; // Autoboxing
        }
        public int size() { return a.length; }
    };
}
```

Example: a skeletal implementation of the `Map.Entry` interface:

```
// Skeletal Implementation
public abstract class AbstractMapEntry<K,V> implements Map.Entry<K,V> {
    // Primitive operations
    public abstract K getKey();
    public abstract V getValue();
    // Entries in modifiable maps must override this method
    public V setValue(V value) { throw new UnsupportedOperationException(); }
    // Implements the general contract of Map.Entry.equals
    @Override public boolean equals(Object o) {
        if (o == this) return true;
        if (! (o instanceof Map.Entry)) return false;
        Map.Entry<?,?> arg = (Map.Entry) o;
        return equals(getKey(), arg.getKey())
              && equals(getValue(), arg.getValue());
    }
    private static boolean equals(Object o1, Object o2) {
        return o1 == null ? o2 == null : o1.equals(o2); }
    // Implements the general contract of Map.Entry.hashCode
    @Override public int hashCode() {
        return hashCode(getKey()) ^ hashCode(getValue());
    }
    private static int hashCode(Object obj) {
        return obj == null ? 0 : obj.hashCode(); }
}
```