Effective Java

Collection of tips for programming in Java
By Joshua Bloch (Sun)
Quick look at 3 topics (out of 57)
1. Duplicate Object Creation
2. Defensive Copying
3. Immutable Classes
Slides borrowed from Bloch & adapted
1) Avoid Duplicate Object Creation

- Reuse existing object instead

- Simplest example
  ```java
  String s = new String("DON’T DO THIS!");
  String s = "Do this instead";
  ```

  - Since Strings constants are reused

- In loops, savings can be substantial
Object Duplication Example

public class Person {
    private final Date birthDate;
    public Person(Date birthDate){
        this.birthDate = birthDate;
    }
    // UNNECESSARY OBJECT CREATION
    public boolean bornBefore2000(){
        Calendar gmtCal = Calendar.getInstance(
            TimeZone.getTimeZone("GMT"));
        gmtCal.set(2000,Calendar.JANUARY,1,0,0,0);
        Date MILLENIUM = gmtCal.getTime();
        return birthDate.before(MILLENIUM);
    }
}

Object Duplication Example (cont.)

```java
public class Person {
    ...
    // STATIC INITIALIZATION CREATES OBJECT ONCE
    private static final Date MILLENIUM;
    static {
        Calendar gmtCal = Calendar.getInstance(
            TimeZone.getTimeZone("GMT"));
        gmtCal.set(2000, Calendar.JANUARY, 1, 0, 0, 0);
        Date MILLENIUM = gmtCal.getTime();
    }
    public boolean bornBefore2000() { // FASTER!
        return birthDate.before(MILLENIUM);
    }
}
```
Object Duplication Summary

- Don’t create unnecessary duplicate objects
  - Reuse improves clarity and performance

- But don’t be afraid to create objects
  - Object creation is cheap on modern JVMs
  - Can enhance simplicity, power, robustness
2) Defensive Copying

- Java programming language is safe
  - Immune to buffer overruns, wild pointers, etc…
  - Unlike C, C++

- Makes it possible to write robust classes
  - Correctness doesn’t depend on other modules
  - Even in safe language, requires effort
Defensive Programming

- Assume clients will try to destroy invariants
  - May actually be true
  - More likely – honest mistakes

- Ensure class invariants survive any inputs
This Class is *Not* Robust!

// GOAL – PERSON’S BIRTHDAY IS INVARIANT
public class Person {

    // PROTECTS birthDate FROM MODIFICATION?
    private final Date birthDate;
    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }

    // RETURNS birthDate
    public Date bday() {
        return birthDate;
    }
}

The Problem – Date is Mutable

Constructors can allow invariant to be modified

// ATTACK INTERNALS OF PERSON
Date today = new Date();
Person p = new Person(today);
today.setYear(78);   // MODIFIES P’S BIRTHDAY!
The Solution – Defensive Copying

```java
public class Person {
    private final Date birthDate;

    // REPAIRED CONSTRUCTOR
    // DEFENSIVELY COPIES PARAMETERS
    public Person(Date birthDate) {
        this.birthDate =
            new Date(birthDate.getTime());
    }

    // RETURNS birthDate
    public Date bday() {
        return birthDate;
    }
}
```
An Important Detail

- First copy parameters, then check copy validity
  - Eliminate window of vulnerability…
  - …between parameter check and copy
- Thwarts multithreaded attack

```java
public Person(Date birthDate){
    // VULNERABLE, DON'T CHECK birthDate HERE
    this.birthDate =
        new Date(birthDate.getTime());
    // CHECK this.birthDate HERE INSTEAD
}
```
Another Important Detail

- Use constructor, not clone, to make copies
  - Necessary because Date class is nonfinal
- Attacker could implement malicious subclass
  - Records reference to each instance in list
  - Provides attacker with access to instance list
- ...and pass subclass to Person( ) constructor
More Defensive Copying

- Constructors are only half the battle
- Accessors can allow invariant to be modified

```java
// ACCESSOR ATTACK ON INTERNALS OF PERSON
Date today = new Date();
Person p = new Person(today);
Date bday = p.bday();
bday.setYear(78); // MODIFIES P’S BIRTHDAY!
```
More Defensive Copying

Solution – defensive copying in accessors

```java
// REPAIRED ACCESSOR DEFENSIVELY COPY FIELDS
public class Person {
    // RETURNS CLONE (COPY) OF birthDate
    public Date bday() {
        return (Date) birthDate.clone();
    }
}
```

Now Person class is robust!
Defensive Copying Summary

Don’t incorporate mutable parameters into object – make defensive copies
- Constructors
- Static factories
- Pseudo-constructors
- Mutators

Return defensive copies of mutable fields
- Accessors

Real lesson – use **immutable** components
- Eliminates the need for defensive copying
3) Immutable Classes

- Class whose instances cannot be modified

- Examples
  - String
  - Integer
  - BigInteger

- How, why, and when to use them
How to Write an Immutable Class

- Don’t provide any mutators
- Ensure that no methods may be overridden
- Make all fields final
- Make all fields private
- Ensure exclusive access to any mutable components
public final class Fval {
    private final float f;
    public Fval(float f) {
        this.f = f;
    }
    // ACCESSORS WITHOUT CORRESPONDING MUTATORS
    public float value() { return f; }

    // ALL OPERATIONS RETURN NEW Fval
    public Fval add(Fval x) {
        return new Fval(f + x.f);
    }
    // SUBTRACT, MULTIPLY, ETC. SIMILAR TO ADD
public boolean equals(Object o) { 
    if (o == this) return true;
    if (!((o instanceof Fval))
        return false;
    Fval c = (Fval) o;
    return (Float.floatToIntBits(f) ==
            Float.floatToIntBits(c.f));
}
}
Distinguishing Characteristic

- Return new instance instead of modifying
- Functional programming
- May seem unnatural at first
- Many advantages
Advantage 1 – Simplicity

- Instances have exactly one state
- Easy to design, implement
- Constructors establish invariants
- Invariants can never be corrupted
- Requires no effort on the part of clients
Advantage 2 – Inherently Thread-Safe

- No need for synchronization
  - Internal or external
  - Since no writes to shared data
- Can’t be corrupted by concurrent access
- By far the easiest approach to thread safety
// EXPORTED CONSTANTS
public static final Fval ZERO = new Fval(0);
public static final Fval ONE = new Fval(1);

// STATIC FACTORY CAN CACHE COMMON VALUES
public static Fval valueOf(float f) {
...
}

// PRIVATE CONSTRUCTOR MAKES FACTORY MANDATORY
private Fval (float f) {
    this.f = f;
}

Advantage 3 – Can Be Shared Freely
Advantage 4 – No Copies

- No need for defensive copies
- No need for any copies at all!
- No need for clone or copy constructor
- Not well understood in the early days

```java
public String(String s); // Should not exist
```
Advantage 5 – Composability

- Excellent building blocks
- Easier to maintain invariants
  - If component objects won't change
- Special cases
  - Map keys
  - Set elements
The Major Disadvantage

- Separate instance for each distinct value
- Creating these instances can be costly
  
  ```java
  BigInteger moby = ...;  // A million bits
  moby = moby.flipBit(0); // Ouch!
  ```

- Problem magnified for multistep operations
  
  - Provide common multistep operations as primitves
  - Alternatively provide mutable companion class
When to Make Classes Immutable

- Always, unless there's a good reason not to
- Always make small “value classes” immutable

  - Examples
    - Color
    - PhoneNumber
    - Price

  - Date and Point (both mutable) were mistakes!
  - Experts often use long instead of Date
When to Make Classes Mutable

Class represents entity whose state changes

- Real-world
  - BankAccount, TrafficLight
- Abstract
  - Iterator, Matcher, Collection
- Process classes
  - Thread, Timer

If class must be mutable, minimize mutability

- Constructors should fully initialize instance
- Avoid reinitialize methods
Effective Java Conclusion

- Reuse objects where appropriate
  - Improves clarity and performance

- Make defensive copies where required
  - Provides robustness

- Write immutable classes
  - Simple, thread-safe, sharable and reusable