
SHARCNET™

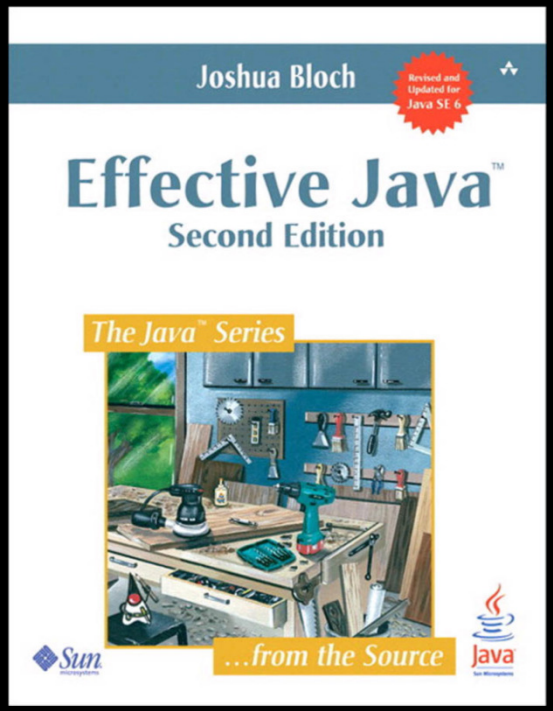
Programming Best Practices

Tips For Defensive Programming

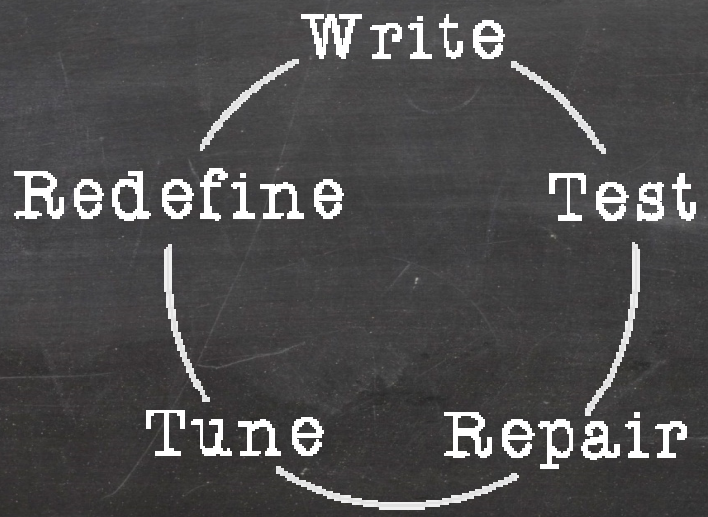
Ed Armstrong

"Effective Java, Second Edition" by Joshua Block.

Chapters and page references are indicated on the slides on which they occur.



Your Own Worst Enemy



Background

- Correctness, clarity, & conciseness of code.
- Abstraction, encapsulation, & information hiding.
- Cohesion & coupling.
- OOP : Object Oriented Programming

OOP Language Support

Abstraction – Interfaces

Inheritance – Method “extension”

Encapsulation – Access level (private, protected, public)

Modularity – Classes

Part 1 : Style

- avoid long code blocks
- use naming conventions
- prefer collections over arrays
- order class variables by scope
- wrap all variables
- declare variables when needed

```
public void process (int i){
 /* long detailed commend on what
   * this it statement is doing */
 if (i == 0){
   blah...
   blah...
   blah...
  }
 /* long detailed commend on what
   * this it statement is doing */
  else if (i == 1){
   blah...
   blah...
   blah...
  }
 /* long detailed commend on what
   * this it statement is doing */
  else if (i == 2){
   blah...
   blah...
   blah...
  }
  ... more else if statements ad nauseam
```

```
public void process (int i){
  if (i == 0) this.addAll();
  else if (i == 1) this.addSome();
  else if (i == 2) this.addNone();
  else this.addRandom();
}

public void addAdd() ...
public void addSome() ...
public void addNone() ...
public void addRandom() ...
```

```
package fruit;

import java.awt.Shape;

public class Fruit implements IFood{
   static final int DISPLAY_RESOLUTION = 640;
   private Flavour flavour;
   private Colour colour;
   private Shape shape;

   public Fruit(){ ... }
   public void ripen(){ ... }
   public void addToBasket(Basket basket){ ... }
}
```

```
package fruit;

import java.awt.Shape;

public class Fruit implements IFood{
   static final int DISPLAY_RESOLUTION = 640;
   private Flavour flavour;
   private Colour colour;
   private Shape shape;

   public Fruit(){ ... }
   public void ripen(){ ... }
   public void addToBasket(Basket basket){ ... }
}
```

Package

Package Tree

Class/Interface

"Global"

Variable

Method

Multi-word Names

```
public class Apple extends Fruit{
   /* private variables */
   private Colour colour;
   private int size;

   /* package private */
   Type type;
   Flavour flavour;

   /* protected */
   Boolean seeded;

   /* public */
   public final int universalIdentifier;
}
```

```
package fruit;

import java.awt.Shape;

public class Fruit implements IFood{
   static final int DISPLAY_RESOLUTION = 640;
   private Flavour flavour;
   private Colour colour;
   private final Shape shape;

   public Flavour getFlavour() { ... }
   public void setFlavour(Flavour flavour) { ... }

   public Flavour getColour() { ... }
   public void setColour() { ... }

   public Shape getShape() {...}
}
```

note: final variable

```
int i, j, k;

for (i = 0; i < 10; i++){
}

for (k = 0; k < 10; k++){
}

for (j = 0; j < 10; j++){
}
```

```
for (int i = 0; i < 10; i++){
}

for (int i = 0; i < 10; i++){
}

for (int i = 0; i < 10; i++){
}
```

```
Iterator <Fruit> i = collection.iterator();

while (i.hasNext()){
 Fruit f = i.next();
 f.eat();
}
```

```
for (Fruit f : collection){
 f.eat();
}
```

```
// Can you spot the bug?

enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();

for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
        for (Iterator<Rank> j = ranks.iterator(); j.hasNext();
                deck.add(new Card(i.next(), j.next()));
```

Code snippet from “Essential Java 2nd Edition”

```
// Can you spot the bug?

enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();

for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
        for (Iterator<Rank> j = ranks.iterator(); j.hasNext();
                deck.add(new Card(i.next(), j.next()));
```

Code snippet from “Essential Java 2nd Edition”

```
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

for (Suit suit : suits){
   for (Rand rank : ranks){
      deck.add(new Card(suit, rank));
   }
}
```

Code snippet from “Essential Java 2nd Edition”

Part 2 : Patterns & Practices

- exceptional circumstances
- null values
- over-riding equals
- interfaces

Exceptional Circumstances

Assert : Things that should never occur; to protect against programming errors (private methods).

RtuntimeException : To ensure that a class is used correctly (public methods).

Exception : (!RuntimeException) Things out of the programmer's control, that the end user wants to know about.

Assert

java -jar -enableassertions Examples.jar

```
public class AssertExample {
   static public void printString(String string){
      assert string != null;
      System.out.println(string);
   }
}
```

Exception

```
public class AssertExample  throws Exception {
   static public void printString(String string){
      if (string == null) throw new Exception();
      System.out.println(string);
   }
}
```

Runtime exception

```
public class AssertExample {
   static public void printString(String string){
      if (string == null) throw new RuntimeException();
      System.out.println(string);
   }
}
```

Avoid External NULLs

I call it my billion-dollar mistake ...the invention of the null reference in 1965. At that time, I was designing [ALGOL] ... I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

- Richard Hoare, the inventor of the NULL reference.

Guidelines to using NULL

Only use null privately in a class.
Don't return NULLs.
Throw an exception when null is passed in.
Return empty collections.
Return NULL representative objects.

Return Empty Collections

in class

```
public Fruit[] getFruit(){
  if (collection.size() == 0) return null;
  …
}
```

in class

```
public Fruit[] getFruit(){
  if (collection.size() == 0) new Fruit[0];
  …
}
```

Return Empty Collections

in class
```
public Fruit[] getFruit(){
  if (collection.size() == 0) return null;
  …
}
```

in main
```
Fruit f = getFruit();
if (f != null){
   for (Fruit f : getFruit()) {
      [do something]
   }
}
```

in class
```
public Fruit[] getFruit(){
  if (collection.size() == 0) new Fruit[0];
  …
}
```

in main
```
for (Fruit f : getFruit()) {
   [do something]
}
```

Single Instance

```
class Basket{
   private ArrayList <Fruit> fruits;
   public Fruit getFruit(){
      if (fruits.size() == 0) return null;
      else return fruits.remove(0);
   }
}
```

Single Instance Fixed

```
class Basket{
   private ArrayList <Fruit> fruits;

   public boolean hasFruit(){
      return fruits.size() > 0;
   }

   public Fruit getFruit(){
      if (fruits.size() == 0) throw new NullPointerException();
      else return fruits.remove(0);
   }
}
```

NULL misrepresenting errors

```
Fruit aPieceOfFruit = fruitBasket.getFruit();

[do some stuff]

aPieceOfFruit.eat(); /* throws NullPointerException */
```

NULL misrepresenting errors

```
Fruit fruit = basket.getFruit(); /* throws NullPointerException */

[do some stuff]

aPieceOfFruit.eat();
```

Overriding Equals

Reflexive if a=a
Symmetric if a=b then b=a
Transitive if x=y and y=z then x=z
Consistent if x=y now, x=y always
Non-Empty Object x != NULL

Immutable Objects

"Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible."

– Joshua Block, Effective Java

- Simple to use and implement.
- Doesn't need a copy, or synchronization.
- Make good keys.
- Never indeterminate.

- Make the class final.
- Make all fields final & private.
- Provide no accessor methods.
- Set all values in constructor (or factory).

Interfaces

```
interface IFood {
    Flavour getFlavour();
    Size getSize();
    void setPortions(int i);
    int getPortions();

    default void bite(){
        this.setPortions(this.getPortions() - 1);
    }

}
```

Interfaces

```
interface IFood {
    Flavour getFlavour();
    Size getSize();
    void setPortions(int i);
    int getPortions();

    default void bite(){
        this.setPortions(this.getPortions() - 1);
    }
}
```

default = public

default

Code to the interface

```
Class Bucket{
  private ArrayList myList= new
ArrayList();
}
```

```
Class Bucket{
  private List myList = new ArrayList();
}
```

--- REFACTOR ---

```
Class Bucket{
  private List myList = new TreeList();
}
```

Immutable Interface

```
public interface ImmutableFruit {
   Flavour getFlavour();
   Colour getColour();
   Shape getShape();
}
```

```
public class Fruit
implements ImmutableFruit{

   private Flavour flavour;
   private Colour colour;
   private final Shape shape;

   public Flavour getFlavour() {...}
   public Colour getColour() {...}
   public Shape getShape() {...}

   public Fruit(){ ... }

   public void ripen(){ ... }
}
```

Citations

Collected Java Practices, [online], http://www.javapractices.com (accessed 2015)

Bloch, J. "Effective Java, 2nd edn. The Java Series." (2008).