Effective Java, Chapter 7

Methods

Item 49: Check Parameters for Validity

- **Check**
  - If your method has a notion that some values are "*invalid*" and knows how to identify those values *early* in the code.
  - If it's *not very expensive* to check them.
- **Document**
  - How does the client know which parameters are / aren't checked?
  - How does the client know what you're going to do if they're invalid?

Javadoc Example

```
/**
 * Each BankAccount object models the account information for
 * a single user of Fells Wargo bank.
 * @author James T. Kirk
 * @version 1.4 (Aug 9 2008)
 */
public class BankAccount {
       ...

    /**
     * Deducts the given amount of money from this account's
     * balance, if possible, and returns whether the money was
     * deducted successfully (true if so, false if not).
     * If the account does not contain sufficient funds to
     * make this withdrawal, no funds are withdrawn.
     *
     * @param amount the amount of money to be withdrawn
     * @return true if amount was withdrawn, else false
     * @throws IllegalArgumentException if amount is negative
     */
    public boolean withdraw(double amount) {
        ...
    }
}
```

Early Detection

- can handle a precondition violation by throwing an exception:

```
 public boolean withdraw(double amount){
    if ( amount < 0) {
        throw new IllegalArgumentException("The amount
should not be negative."));
    }

    // business logic
    ......
}
```

  - **fail-fast**: Client learns about the problem immediately and can fix it. Passing a bad value usually indicates a bug in the client, so this is good.

Exceptional Case

- Binary search on an `int[]` : from Java API

  *"Searches the specified array of `ints` for the specified value using the binary search algorithm. The array **must** be sorted (as by the `sort` method, above) prior to making this call. If it is not sorted, the results are undefined. ..."*

  `Class Arrays: public static int binarySearch(int[] a, int key)`

- Why doesn't Sun just check whether the array is sorted? :
  - Sort is *costly* (takes O(n log n) or worse; search is O(log n)).
    - Even checking to see whether the array is sorted is costly (O(n)); omitting this check and assuming it to be true makes binary search run much faster.
  - Sort modifies the input array; binarySearch would have a *side effect*.

Preconditions and private

- Private internal methods do not usually test preconditions:

```
// Helper does the real work of removing an item.
private void removeHelper(int index) {
    // should I check 0 <= index < size here?
    for (int i = index; i < size - 1; i++) {
        elementData[i] = elementData[i + 1];
    }
    elementData[size - 1] = 0;
    size--;
}
```

- Why not?
  - Since the method can only be called internally, the class author can make sure to call it only when the preconditions hold.

Item 50: Make Defensive Copies When Needed

- Java is a safe language
  - Yeah!
  - No buffer overflow, array overruns, wild pointers, memory corruption as in C, C++
- But you still need to *insulate your class from client classes*
- Best Approach
  - Assume Clients of your class will do their best to destroy your invariants
  - This is actually what happens in any type of security attack
- Surprisingly easy to unintentionally provide access to internal state

More Item 50

Demo

```
public final class Period {
  private final Date start;
  private final Date end;
  public Period (Date start, Date end) {
    if (start.compareTo(end) > 0) throw new IAE(…);
    this.start = start; this.end = end;   //oops – should make defensive copies
  }
  public Date start() { return start;}    // oops again!
  public Date end()   { return end;}      // oops yet again!
}
```

More Item 50

Demo

```
// Broken "immutable" time period class
public final class Period {
  private final Date start;
  private final Date end;
  public Period (Date start, Date end) {
    if (start.compareTo(end) > 0) throw new IAE(…);
    this.start = start; this.end = end;    //oops – should make defensive copies
  }
  public Date start() { return start;}     // oops again!
  public Date end()   { return end;}       // oops yet again!
}
// Attack code
Date start = new Date();
Date end   = new Date();
Period p   = new Period(start, end);
end.setYear(78);     // Attack 1: Modify internals of p via passed reference
p.end().setYear(78); // Attack 2: Modify internals of p via returned state
```

More Item 50

Demo

```
// Repaired constructor and getters – now Period really is immutable
public Period (Date start, Date end) {
  // Note:   clone() *not* used to make defensive copy
  // Reason: Date class not final; hence return type may not be java.util.Date
  this.start = new Date(start.getTime());  // Defensive copy
  this.end   = new Date(end.getTime());    // Defensive copy

  // Defensive copies made *before* exception check to avoid TOCTOU attack
  if (this.start.compareTo(end) > 0) throw new IAE(…);
}

// clone also safe here, but constructors or static factories better (Item 11)
public Date start() { return new Date(start.getTime()); }
public Date end()   { return new Date(end.getTime());}
}
```

Typical scenario:

```
    return elements.clone();  // when elements is an Array
```

More Item 50

- Make defensive copies of all <u>mutable data provided by client</u>
- Make defensive copies of all <u>mutable internal state returned to client</u>
  - Includes all arrays of length >0
- Arguably, the lesson is that ***Immutable objects should be used where possible***
  - No copying necessary, so you can't forget!
- Caveat: Occasionally, you can't afford the copy ☹
  - Heavy performance penalty?
  - Do you trust the client?
    - From a security perspective, trust is a bad thing
  - Some objects are explicitly handed off

Item 51: Design Method Signatures Carefully

- Choose methods names carefully
  - Obey the standard naming convention
- Don’t go overboard providing convenience methods
  - A class should not have too many methods
- Avoid long parameter lists
  - Reduce to multiple methods
    - See next slide for example
  - Use helper class

Reduce to multiple methods

- Should interface *List* provides a method to find the first/last index of an element in a sublist?

```
int indexOf(int fromIndex, int toIndex, Object o);
int lastIndexOf(int fromIndex, int toIndex, Object o);
```

- However, above requirements are fulfilled by using the combination of methods:
  - High power-to-weight ratio

```
List<E> subList(int fromIndex, int toIndex);
int indexOf(Object o);
int lastIndexOf(Object o);
```

More Item 51

- For parameter types, favor interfaces over classes

```
public static boolean TreeSet<Color> filterPrimary(TreeSet<Color> s)
// vs.
public static boolean Set<Color> filterPrimary(Set<Color> s)
```

- Prefer two-element enum types to boolean parameters

```
boolean Fahrenheit
// vs.
enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Item 52: Use Overloading Judiciously

Demo

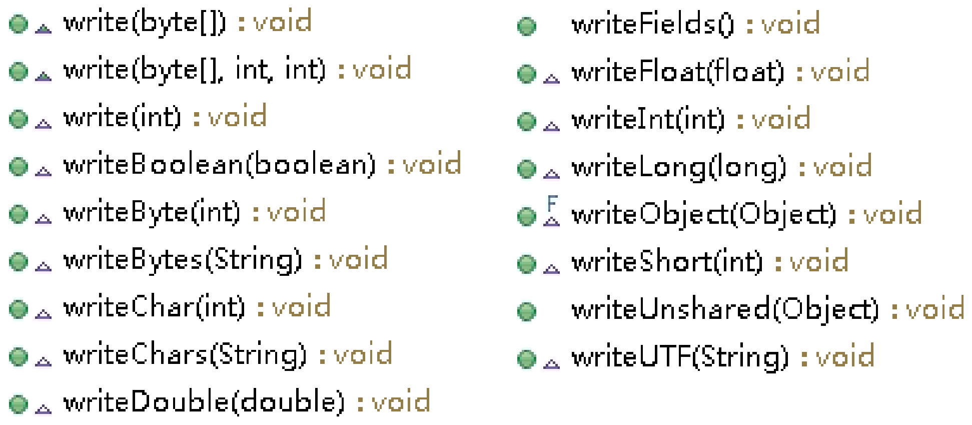
```
// Broken! - What does this program print?
public class CollectionClassifier {
   public static String classify(Set<?> s)        { return "Set"; }
   public static String classify(List<?> l)       { return "List"; }
   public static String classify(Collection<?> c) { return "Collection"; }

   public static void main(String[] args) {
      Collection <?>[] collections = {
         new HashSet<String>(),
         new ArrayList<BigInteger>(),
         new HashMap<String, String>().values()
      };
      for (Collection <?> c : collections)
          System.out.println(classify(c));
   }
}
// Problem:  classify() is overloaded, not overridden
```

Overriding vs. overloading

- Overriding
  - Normal OO Pattern
  - Dynamically determined by runtime system
- Overloading
  - Exceptional OO Pattern
  - Statically determined by compiler
- Avoid confusing uses of overloading, especially in APIs
  - Overloading documentation in Java Language Specification is 33 pages long!
- Safe Policy: Don’t export two overloadings with the same number of parameters

Avoid using overloading

- Prefer different method names to overloading
  - E.g. many write methods of *ObjectOutputStream*



```
write(byte[]) : void                 writeFields() : void
write(byte[], int, int) : void       writeFloat(float) : void
write(int) : void                    writeInt(int) : void
writeBoolean(boolean) : void         writeLong(long) : void
writeByte(int) : void                writeObject(Object) : void
writeBytes(String) : void            writeShort(int) : void
writeChar(int) : void                writeUnshared(Object) : void
writeChars(String) : void            writeUTF(String) : void
writeDouble(double) : void
```

- Use static factories
  - since you can not rename a constructor

More on Item 52

- Exceptional circumstance:
  - when at least one corresponding formal parameter in each pair of overloadings has a "radically different" type in the two overloadings.
  - For example: Collection and primitive types; String and Throwable
- But be careful…

More Item 52:

```
// Broken! - Autoboxing and overloading combine for total confusion!
public class SetList {
   public static void main(String[] args) {
      Set <Integer> set  = new TreeSet  <Integer>();
      List<Integer> list = new ArrayList<Integer>();

      for (int i = -3; i < 3; i++) {
          set.add(i);
          list.add(i);
      }
      for (int i = 0; i < 3; i++) {
          set.remove(i);
          list.remove(i);
      }
      System.out.println(set + " " + list);
}
      // Expect [-3, -2, -1] [-3, -2, -1]
      // Actual [-3, -2, -1] [-2, 0, 2]
      // Key:  List interface overloads remove(Object e) and remove(int i)
      // Prior to Java 1.5, this wasn't a problem
```

-3, -2, -1, 0, 1, 2

Demo

- Set.java
  - `boolean remove(Object o);`
- List.java
  - `E remove(int index);`
  - `boolean remove(Object o);`

Guideline on same number of parameters

1. Try to avoid overloading methods have the *same number of parameters*.
2. At least avoid situations where the same set of parameters can be passed to different overloadings by the addition of *casts*.
3. If above situations cannot be avoided, ensure that all overloadings behave identically when passed the same parameters.
   - Example in the book:

```
String.java

public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence)sb);
}
```

4. Note: overloading are more error-prone after *autoboxing* and *generics* are part of the language

Item 54: Return Empty Arrays or Collections, Not Nulls

```
// Common example
private final List<Cheese> cheesesInStock = . . .;
/**
  *@return an array containing all of the cheeses in the shop,
  *    or null if no cheeses are available for purchase
  */
public Cheese[] getCheeses() {
   if (cheesesInStock.size() == 0) return null;
   ...
// Client code
Cheese[] cheeses = shop.getCheeses();
if (cheeses != null &&
    Arrays.asList(cheeses).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
// vs.
if (Arrays.asList(cheeses).contains(Cheese.STILTON))
   System.out.println("Jolly good, just the thing.");
```

More Item 54:

- Making client handle null as special case is undesirable
- Performance penalty of creating an empty array is almost always irrelevant (See Item 55)
- Right way to return an array from a collection

```
// The right way to return an array from a collection
private final List<Cheese> cheesesInStoc

private static final Cheese[] EMPTY_CHEE

/**
 * @return an array containing all
 */
public Cheese[] getCheese(){
   return cheeseInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

List#toArray(T[] a)
If the list fits in the specified array, it is returned therein.
Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list.

Item 56: Write doc comments for all exposed API Elements

- Precede every exported *class, interface, constructor, method, and field* description with a doc comment
- Doc comment should describe CONTRACT between method and its client
- No two members or constructors should have the same summary description
- Every *method* should have
  - @param tag for each parameter
  - @return tag (unless return type is void)
  - @throws tag for each exception (both checked and unchecked)

More Item 56

```
// Good Example
/**
  * Returns the element at the specified position in this list.
  *
  * <p>This method is <i>not</i> guaranteed to run in constant time.  In some
  * implementations it may run in time proportional to the element position.
  *
  * @param index index of element to return; must be non-negative
  *        and less than the size of this list
  * @return the element at the specified position in this list
  * @throws IndexOutOfBoundsException if the index is out of the range
  *         ({@code index < 0 || index >= this.size()})
  */
 E get(int index)
```

- New Javadoc tags since 1.5
  - {@literal}: suppress processing of HTML markup and nested Javadoc tags
  - {@code}: same effect as {@literal}, plus render the code fragment in code font

Summary Description

- The first "sentence" of each doc comment (as defined below) becomes the *summary description* of the element to which the comment pertains.
- Be careful if the intended summary description contains a period
  - “A college degree, such as B.S., M.S. or Ph.D.”

    will result in the summary description: “A college degree, such as B.S., M.S.”
  - Use literal tag:

```
/**
 * A college degree, such as B.S., {@literal M.S.} or Ph.D.
 * College is a fountain of knowledge where many go to drink.
 */
public class Degree { ... }
```

Summary Description - examples

- For <u>methods/constructors</u>: a full *<u>verb phrase</u>* (including any object) describing the action performed by the method.
  - *ArrayList(int initialCapacity)* — Constructs an empty list with the specified initial capacity.
  - *Collection.size()* — Returns the number of elements in this collection.
- For <u>classes/interfaces/fields</u>: a *<u>noun phrase</u>* describing the thing represented by an instance of the class or interface or by the field itself.
  - *TimerTask* — A task that can be scheduled for one-time or repeated execution by a Timer.
  - *Math.PI* — The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.

Doc comments for Generic

- Generic type/method
  - Be sure to document all *type parameters*

```
/**
 * An object that maps keys to values.  A map cannot contain
 * duplicate keys; each key can map to at most one value.
 *
 * (Remainder omitted)
 *
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K, V> {
    ... // Remainder omitted
}
```

Doc comments for Enum

- Enum type
  - Be sure to document the <u>constants/type/public method</u>
  - Put entire doc comment on one line is OK if its short.

```
/**
 * An instrument section of a symphony orchestra.
 */
public enum OrchestraSection {
    /** Woodwinds, such as flute, clarinet, and oboe. */
    WOODWIND,

    /** Brass instruments, such as french horn and trumpet. */
    BRASS,

    /** Percussion instruments, such as timpani and cymbals */
    PERCUSSION,

    /** Stringed instruments, such as violin and cello. */
    STRING;
}
```

Doc comments for Annotation

- For Type
  - use a verb phrase that says what it means when a program element has an annotation of this type
- For members
  - use noun phrases, as if they were fields.

```
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    /**
     * The exception that the annotated test method must throw
     * in order to pass. (The test is permitted to throw any
     * subtype of the type described by this class object.)
     */
    Class<? extends Exception> value();
}
```

Miscellaneous on documentation

```
The triangle inequality is {@literal |x + y| < |x| + |y|}.

vs.
```

```
The triangle inequality is |x + y|{@literal < }|x| + |y|.
```

- Doc comments should be readable in both the *source code* and in the *generated documentation*

Items mentioning documentation so far

- Item 1: Consider static factory methods instead of constructors
- Item 4: Enforce noninstantiability with a private constructor
- Item 8: Obey the general contract when overriding equals
- Item 10: Always override toString
- Item 17: Design and document for inheritance or else prohibit it
- Item 38: Check parameters for validity
- Item 39: Make defensive copies when needed
- Item 44: Write doc comments for all exposed API elements