Code Layout (1)

September 14, 2017

Byung-Gon Chun

(Slide credits: George Candea, EPFL)

Code Layout
(Overview)

```
/* Use the insertion sort technique to sort the "data" array in
ascending order. This routine assumes that data[ firstElement ]
is not the first element in data and that data[ firstElement-1 ]
can be accessed. */ public void InsertionSort( int[] data, int
firstElement, int lastElement ) { /* Replace element at lower
boundary with an element guaranteed to be first in a sorted
list. */ int lowerBoundary = data[ firstElement-1 ] ; data[
firstElement-1 ] = SORT_MIN; /* The elements in positions
firstElement through sortBoundary-1 are always sorted. In each
pass through the loop, sortBoundary is increased, and the
element at the position of the new sortBoundary probably isn't
in its sorted place in the array, so it's inserted into the
proper place somewhere between firstElement and sortBoundary.
*/ for (int sortBoundary = firstElement+1; sortBoundary <=
lastElement; sortBoundary++ ) { int insertVal = data[
sortBoundary ] ; int insertPos = sortBoundary; while (insertVal
< data[ insertPos-1 ] ) { data[ insertPos ] = data[ insertPos-1
] ; insertPos = insertPos-1; } data[ insertPos ] = insertVal; }
/* Replace original lower-boundary element */ data[
firstElement-1 ] = lowerBoundary; }
```

```
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed. */
public void InsertionSort( int[] data, int firstElement, int lastElement ) {
/* Replace element at lower boundary with an element guaranteed to be first in a
sorted list. */
int lowerBoundary = data[ firstElement-1 ] ;
data[ firstElement-1 ] = SORT_MIN;
/* The elements in positions firstElement through sortBoundary-1 are
always sorted. In each pass through the loop, sortBoundary
is increased, and the element at the position of the
new sortBoundary probably isn' t in its sorted place in the
array, so it' s inserted into the proper place somewhere
between firstElement and sortBoundary. */
for (
int sortBoundary = firstElement+1;
sortBoundary <= lastElement;
sortBoundary++
) {
int insertVal = data[ sortBoundary ] ;
int insertPos = sortBoundary;
while ( insertVal < data[ insertPos-1 ] ) {
data[ insertPos ] = data[ insertPos-1 ];
insertPos = insertPos-1;
}
data[ insertPos ] = insertVal;
}
/* Replace original lower-boundary element */
data[ firstElement-1 ] = lowerBoundary;
}
```

```
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed.
*/
public void InsertionSort( int[] data, int firstElement, int lastElement ) {
    // Replace element at lower boundary with an element guaranteed to be
    // first in a sorted list.
    int lowerBoundary = data[ firstElement-1 ] ;
    data[ firstElement-1 ] = SORT_MIN;
    /* The elements in positions firstElement through sortBoundary-1 are
    always sorted. In each pass through the loop, sortBoundary
    is increased, and the element at the position of the
    new sortBoundary probably isn' t in its sorted place in the
    array, so it' s inserted into the proper place somewhere
    between firstElement and sortBoundary.
    */
    for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;
        sortBoundary++ ) {
        int insertVal = data[ sortBoundary ] ;
        int insertPos = sortBoundary;
        while ( insertVal < data[ insertPos - 1 ] ) {
            data[ insertPos ] = data[ insertPos - 1 ] ;
            insertPos = insertPos - 1;
        }
        data[ insertPos ] = insertVal;
    }
    // Replace original lower-boundary element
    data[ firstElement - 1 ] = lowerBoundary;
}
```

From *Code Complete (2nd edition)*, by Steve McConnell, Microsoft Press, 2004

Evidence that Layout Matters

- Chase & Simon (1973): “Perception in Chess”
- Schneiderman (1976):
  “Exploratory Experiments in Programmer Behavior”
- Subsequently:
  - McKeithen et al. (1981)
  - Soloway & Ehrlich (1984)

Python

- Layout is a first-class citizen

```
def counter(myId, count):
    for i in range(count):
        stdoutmutex.acquire()
        print '[%s] => %s' % (myId, i)
        stdoutmutex.release()
    exitmutexes[myId] = 1  # signal main thread
```

Principles & Objectives

- Visual layout → logical structure of program
  - goal: not to make code look pretty, but understandable
- Leverage common programmer backgrounds
  - enables experts to be quicker [Shneiderman 1976]
- Withstand the test of time
  - how to maintain layout and comments as code evolves
  - modifying one line shouldn't require changing many others

Whitespace

Music is the space between notes.

Claude Debussy

Whitespace

- Whitespaceisthekeytounderstandabletext
  - spaces,tabs,linebreaks,blanklines
- Book=chapters+paragraphs+sentences
  - providesamentalmapofthetopic
  - programsaremuchdenserthanbooks!
- Whitespaceprovidesthebasisforgrouping
- Indentationsuggestslogicalstructure
  - aimforbalance:2-4spacesisoptimal [Miaria et al. 1983]

Whitespace

- White space is the key to understandable text
  - spaces, tabs, line breaks, blank lines
- Book = chapters + paragraphs + sentences
  - provides a mental map of the topic
  - programs are much denser than books !
- Whitespace provides the basis for grouping
- Indentation suggests logical structure
  - aim for balance: 2-4 spaces is optimal [Miaria et al. 1983]

Parentheses

- Compute this value:

```
result = 12 + 4 % 3 * 7 / 8;
```

Parentheses

- Compute this value:

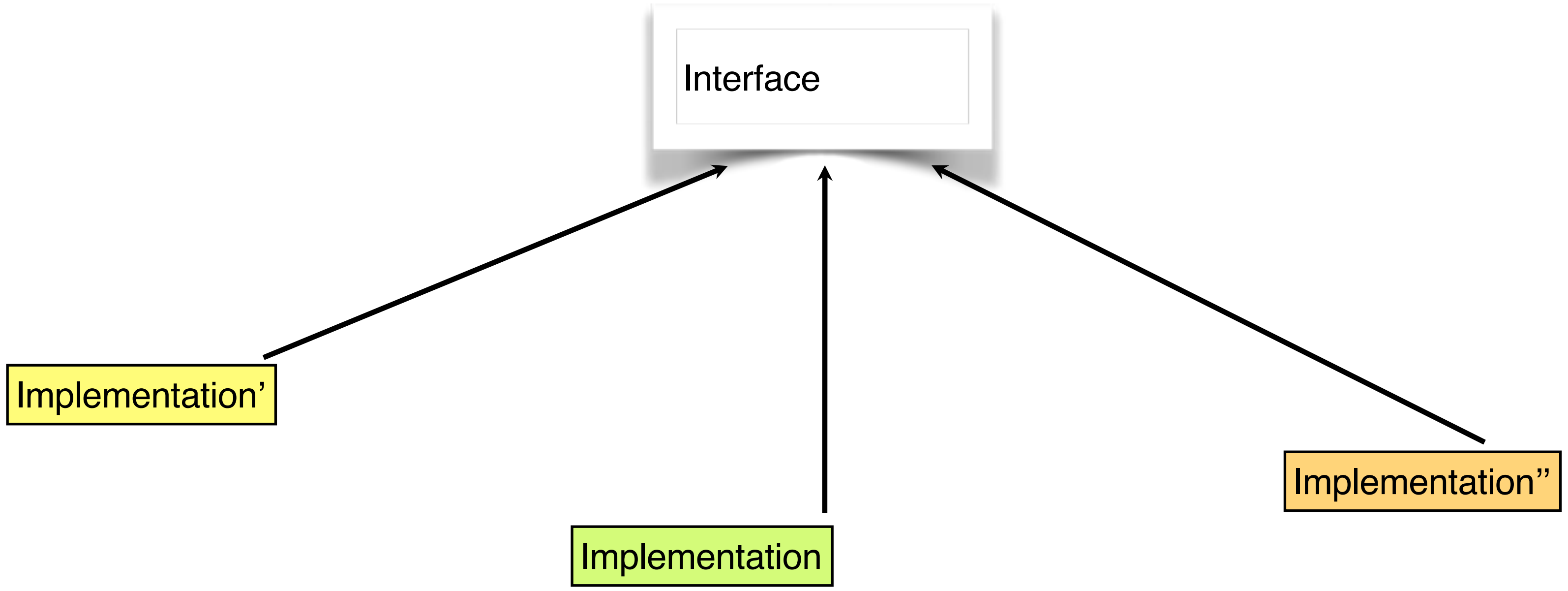
```
result = 12 + (4 % 3) * 7 / 8;
```

Avoid Deceit

- Tell same story to human as to computer

```
arrayLength = 3+4 * 2+7;
// swap left and right elements for whole array
for (i = 0; i < arrayLength; ++i)
    leftElement = left[i];
    left[i] = right[i];
    right[i] = leftElement;
```

Code Layout
(Classes)


Interface
Implementation'
Implementation
Implementation''

Class Layout

1. Header comment
2. Class data
3. Public methods
4. Protected methods
5. Private methods

```
/**
 * Hash table based implementation of the <tt>MyMap</tt> interface.  This
 * implementation provides all of the optional map operations, and permits
 *
 * ...
 *
 */

public class HashMyMap<K,V>
    implements MyMap<K,V>, Cloneable, Serializable
{
    static final int DEFAULT_INITIAL_CAPACITY = 16;
    // ...
    transient Entry[] table;
    // ...

    public HashMyMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " + initialCapacity);
        // ...
    }
    // ...

    protected Statistics getStats (void) {
        // ...
    }
    // ...

    private V getForNullKey() {
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null)
                return e.value;
        }
        return null;
    }

    // ...
}
```

Using Files

- Separate files for interface / implementations
- One class per file
- Name file after the name of the class
  - Java already forces you to do all these = good
  - use upper camel case for class names (e.g., HashMyMap)
- Within a file, separate methods clearly
  - consider using at least two blank lines

Formatting

- Length of line ≤ 120 characters
  - let 120 be the exception, aim for ≤ 80 in the common case
- Length of class ≤ 2,000 lines
  - this is a rough guideline
  - aim for fewer lines if you can

Code Layout (Methods)

General Structure

Return value + parameters →

Exceptions →

Opening brace →

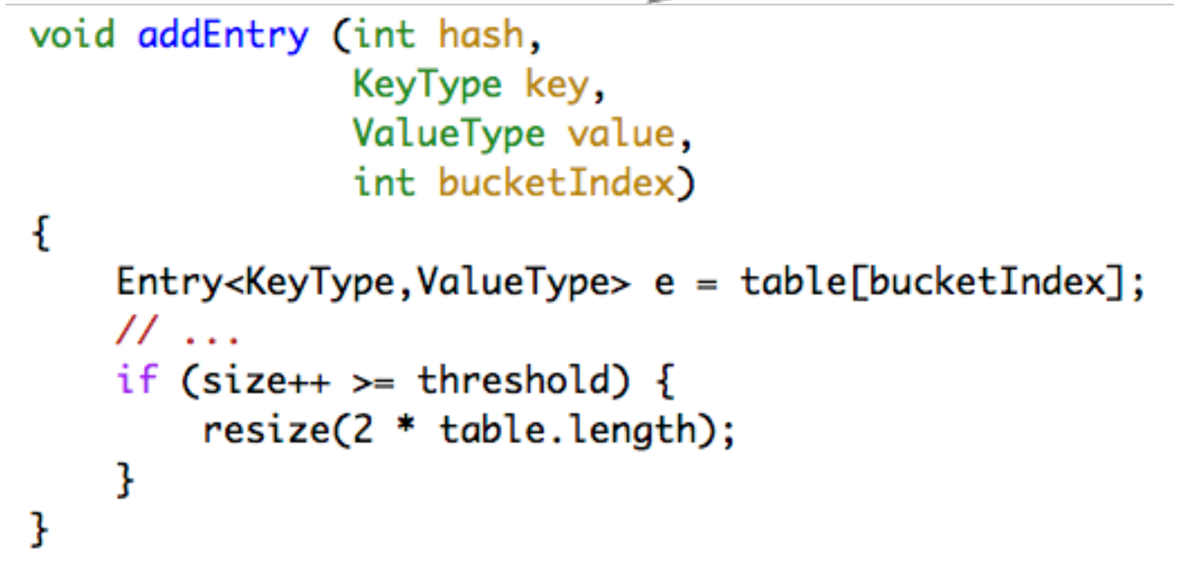
Method body →

Closing brace →

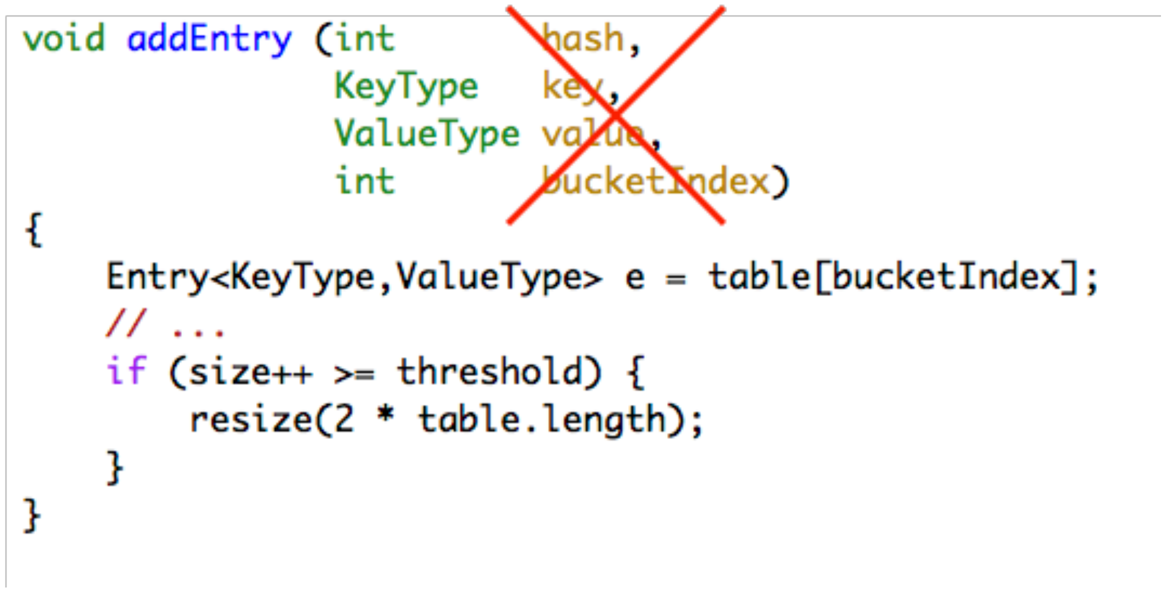
```
private void readObject (java.io.ObjectInputStream s)
    throws IOException, ClassNotFoundException
{
    s.defaultReadObject();

    int numBuckets = s.readInt();
    table = new Entry[numBuckets];

    // ...

    for (int i=0; i<size; i++) {
        K key = (K) s.readObject();
        V value = (V) s.readObject();
        putForCreate(key, value);
    }
}
```

Method Parameters

```java
void addEntry (int hash, KeyType key, ValueType value, int bucketIndex) {
    Entry<KeyType,ValueType> e = table[bucketIndex];
    // ...
    if (size++ >= threshold) {
        resize(2 * table.length);
    }
}
```



```java
void addEntry (int hash,
               KeyType key,
               ValueType value,
               int bucketIndex)
{
    Entry<KeyType,ValueType> e = table[bucketIndex];
    // ...
    if (size++ >= threshold) {
        resize(2 * table.length);
    }
}
```



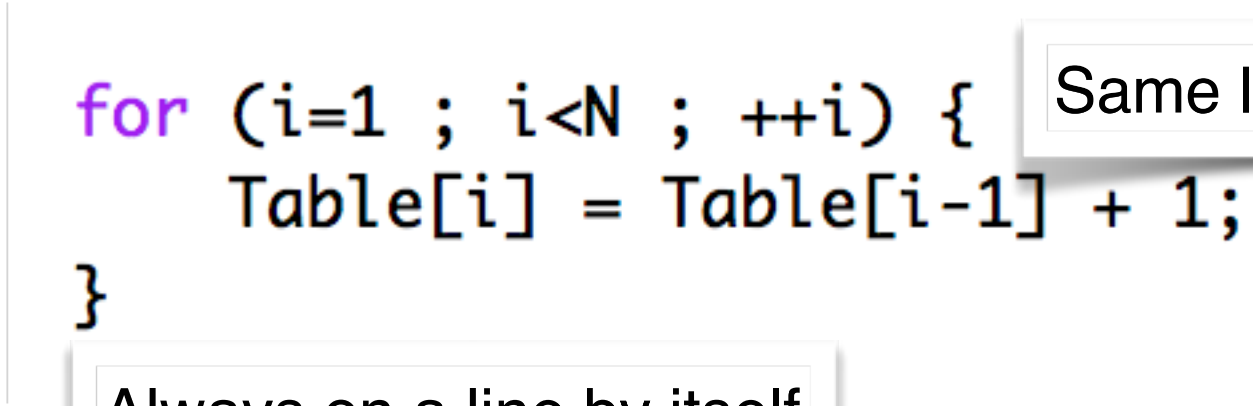
```java
void addEntry (int       hash,
               KeyType   key,
               ValueType value,
               int       bucketIndex)
{
    Entry<KeyType,ValueType> e = table[bucketIndex];
    // ...
    if (size++ >= threshold) {
        resize(2 * table.length);
    }
}
```

Simple Methods

- Keep methods small and focused
  - if >50 lines, ask yourself whether you shouldn’t restructure
  - do not exceed 150 lines
- Maximum 7 parameters
- Simple control flow
  - avoid indirect (mutual) recursion

Code Layout (Statements)

Braces

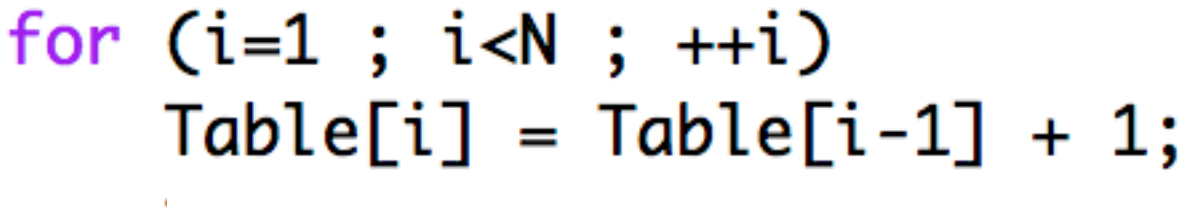


```
for (i=1 ; i<N ; ++i) {
    Table[i] = Table[i-1] + 1;
}
```

Same line as previous statement

Always on a line by itself

Use braces even for single-statement blocks



```
for (i=1 ; i<N ; ++i)
    Table[i] = Table[i-1] + 1;
```

Statement Indentation

Use 4 spaces

```
for (int i=0; i<MAX_LINES; ++i) {
    ReadLine(i);
    ProcessLine(i);
}
```

```
for (int i=0; i < MAX_LINES; ++i)
    {
        ReadLine(i);
        ProcessLine(i);
    }
```

Demarcate Blocks with Whitespace

- Some blocks are not "naturally" demarcated
  - use blank lines to "ungroup" unrelated statements

Demarcate Blocks with Whitespace

```
cursor.start = startingScanLine;
cursor.end   = endingScanLine;
window.title = editWindow.title;
window.dimensions      = editWindow.dimensions;
window.foregroundColor = userPreferences.foregroundColor;
cursor.blinkRate       = editMode.blinkRate;
window.backgroundColor = userPreferences.backgroundColor;
SaveCursor(cursor);
SetCursor(cursor);
```

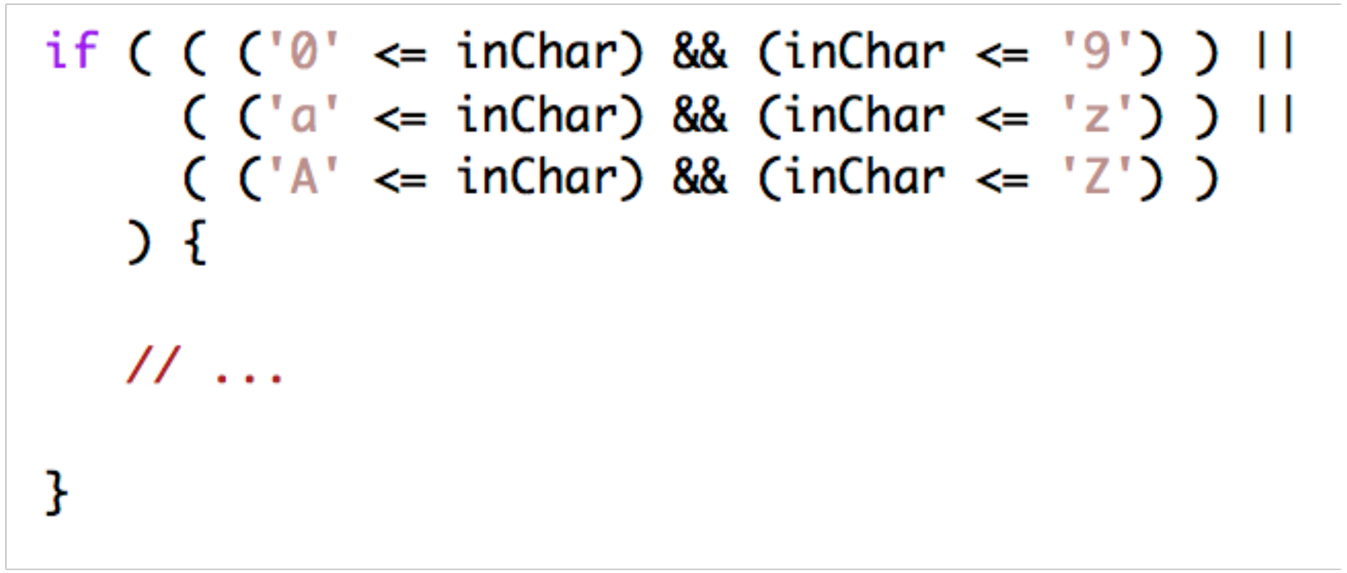
```
window.dimensions = editWindow.dimensions;
window.title = editWindow.title;
window.backgroundColor = userPreferences.backgroundColor;
window.foregroundColor = userPreferences.foregroundColor;

cursor.start = startingScanLine;
cursor.end = endingScanLine;
cursor.blinkRate = editMode.blinkRate;
SaveCursor(cursor);
SetCursor(cursor);
```

Vertical Alignment

```
static final List<String> favorites = Arrays.asList ("Comedy", "Action", "Musical", "Drama");
static final List<Integer> monthIdx = Arrays.asList (2       , 5       ,  1       , 11     );
```

Split Complex Predicates Cleanly

```
if ((('0' <= inChar) && (inChar <= '9') ) || (('a' <= inChar) && (inChar <= 'z')) || (('A' <=
    inChar) && (inChar <= 'Z'))) {

    // ...

}
```



```
if ( ( ('0' <= inChar) && (inChar <= '9') ) ||
     ( ('a' <= inChar) && (inChar <= 'z') ) ||
     ( ('A' <= inChar) && (inChar <= 'Z') )
   ) {

   // ...

}
```

Multi-Line Statements

- Make incompleteness obvious

```
while (pathName[startPath + position] ! = ';') &&
    ( (startPath + position) <= pathName.length() )

    // ...

totalBill = totalBill + customerPurchases[customerID] +
    SalesTax(customerPurchases[customerID]);

    // ...

DrawLine (window.north, window.south, window.east, window.west,
    currentWidth, currentAttribute);
```

Multi-Line Statements

- Can also put continuation at start of line

```
while (pathName[startPath + position] ! = ';')
    && ( (startPath + position) <= pathName.length() )

    // ...

totalBill = totalBill + customerPurchases[customerID]
    + SalesTax(customerPurchases[customerID]);

    // ...
```

One Statement per Line

```
i = 0; j = 0; k = 0; DestroyBadLoopNames (i, j, k);
```

```
if (map.isEmpty()) ++i;
```

Each Line Should Do One Thing

```
PrintMessage (++n, n+2);
```