Coding Style

October 18, 2022

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:

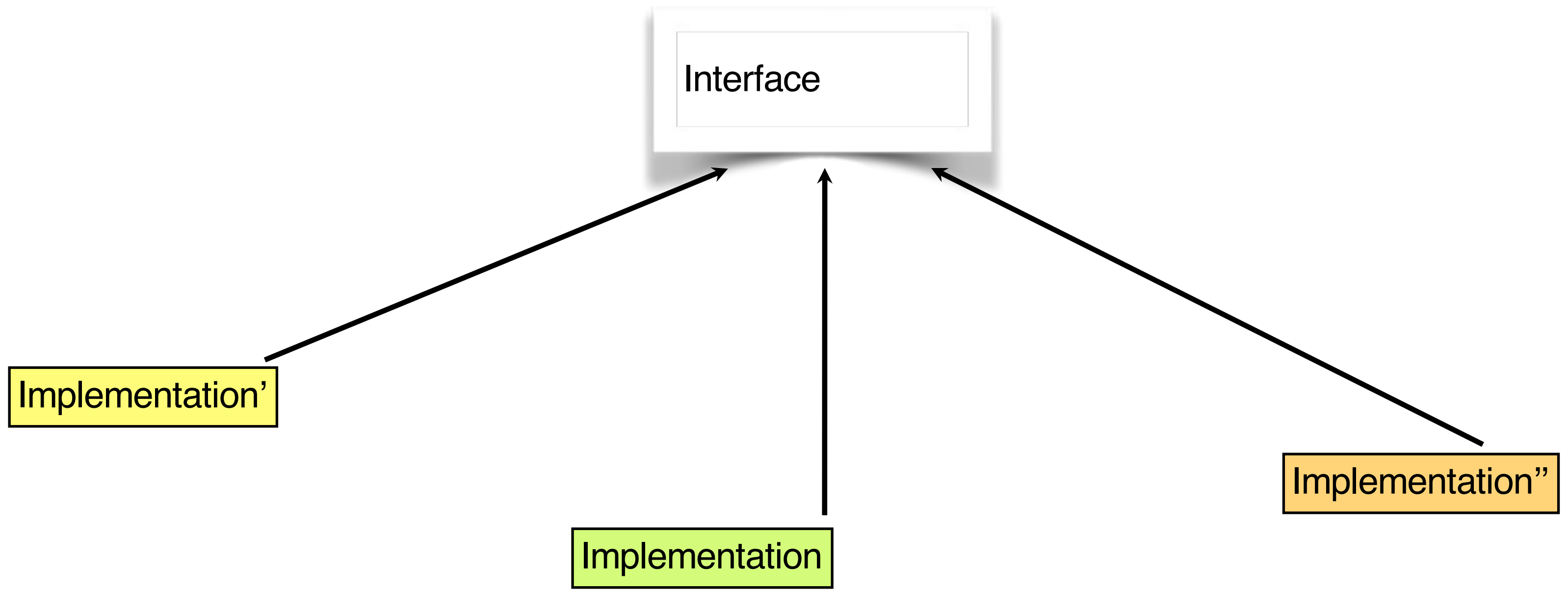
```
result = 12 + (4 % 3) * 7 / 8;
```

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

Python PEP8 Naming

- modules (filenames) should have *short, all-lowercase names*, and they can contain underscores;
- packages (directories) should have *short, all-lowercase names*, preferably without underscores;
- classes should use the CapWords convention.

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

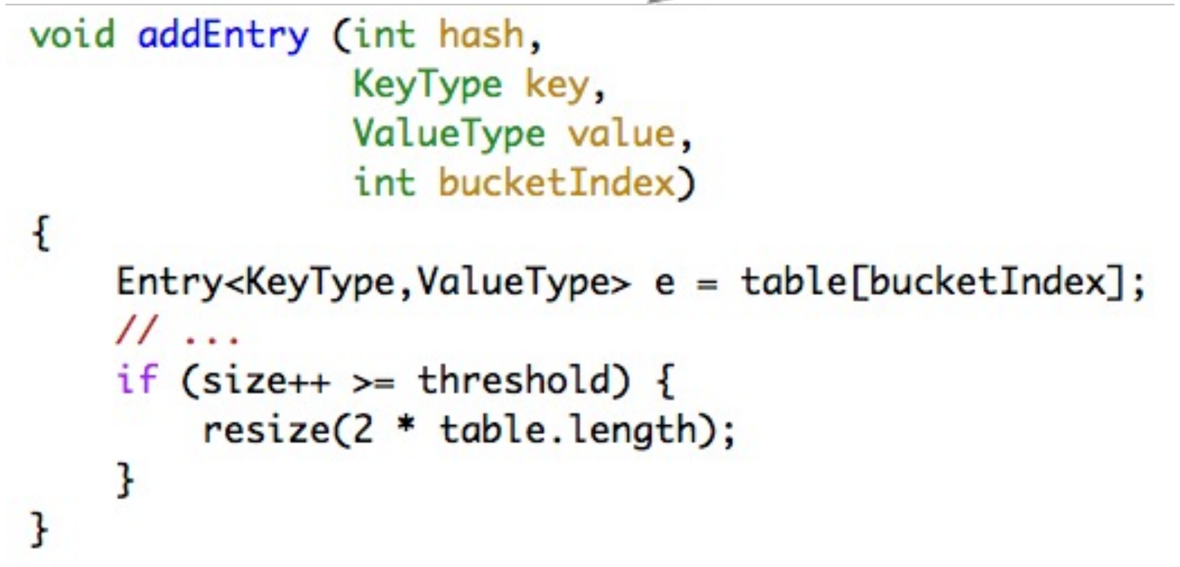
Code Layout (Methods)

General Structure

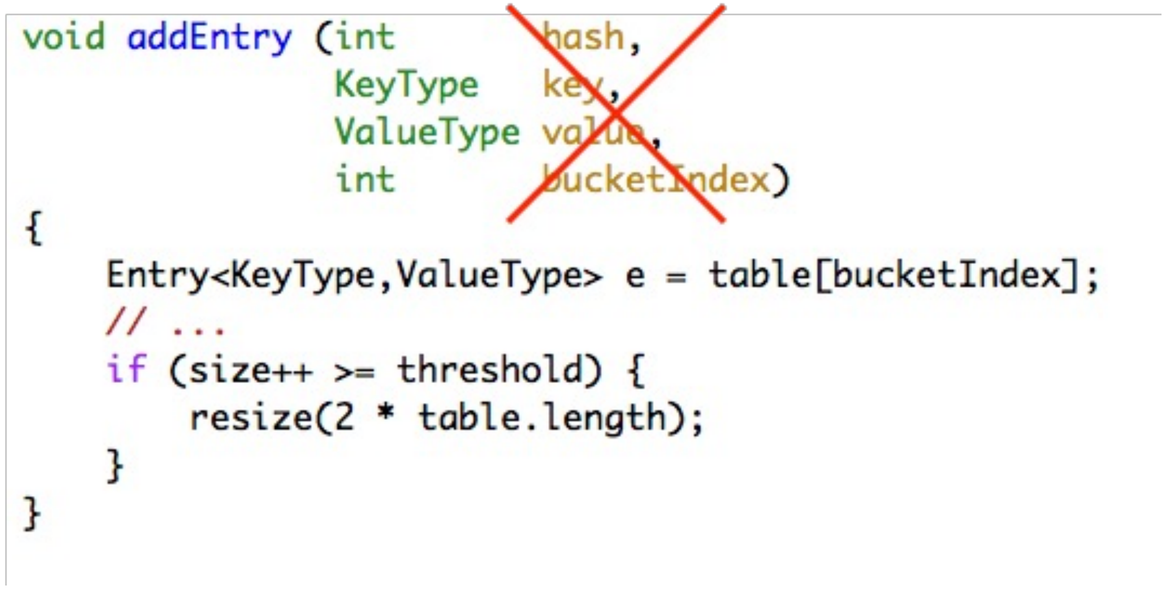
```
Return value + parameters →  private void readObject (java.io.ObjectInputStream s)
               Exceptions →      throws IOException, ClassNotFoundException
            Opening brace →  {
                                 s.defaultReadObject();

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

              Method body →      // ...

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

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);
```

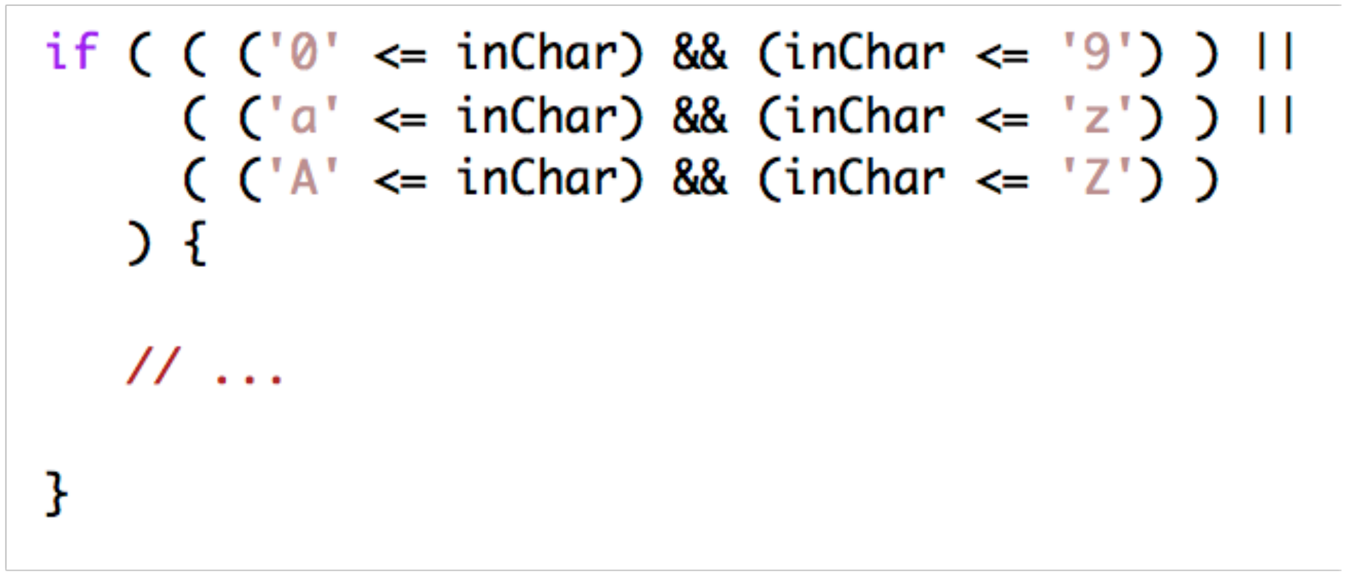
```
window.dimensions = editWindow.dimensions;
window.title = editWindow.title;
window.backgroundColor = userPreferences.backgroundColor;
window.foregroundColor = userPreferences.foregroundColor;

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

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);
```

Naming Variables

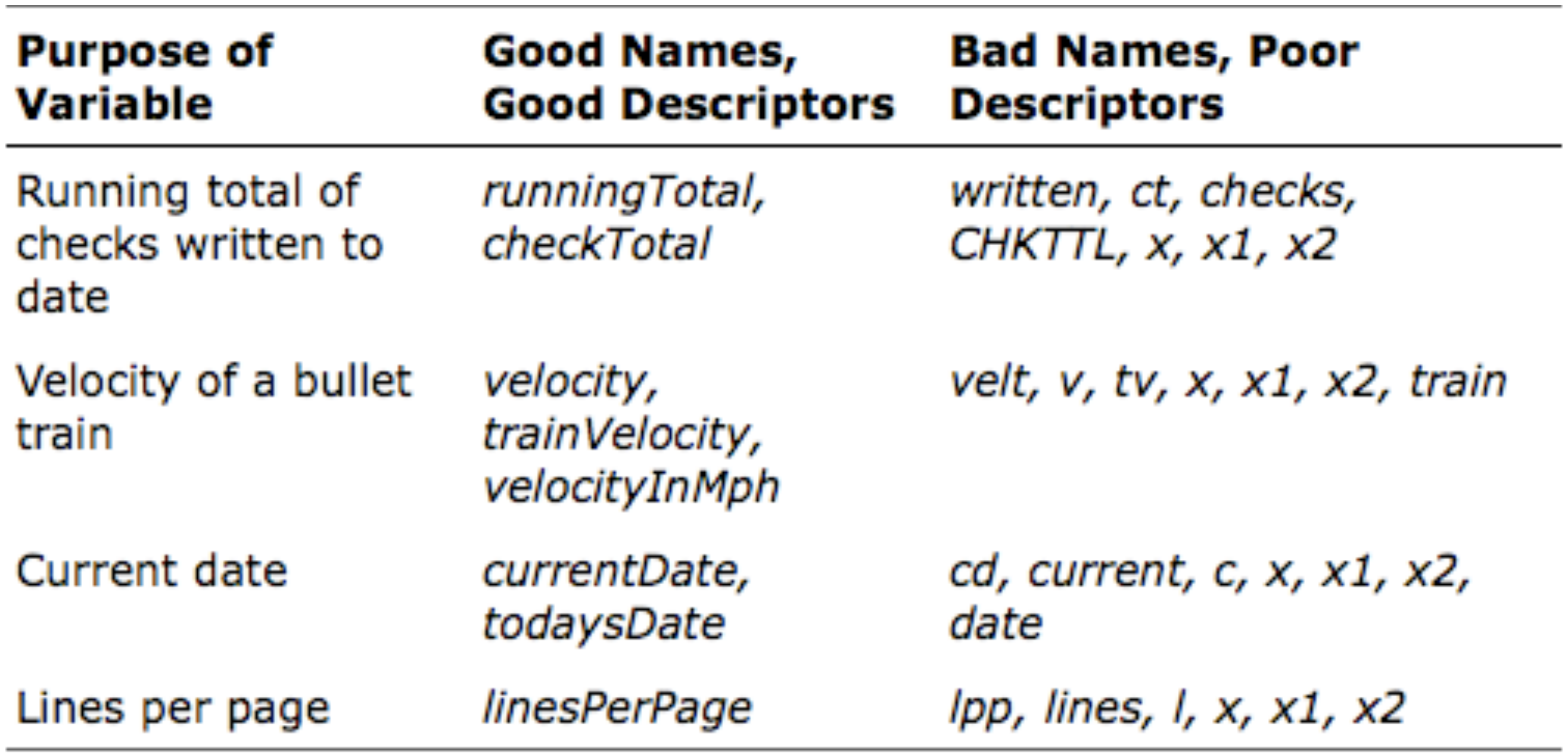
Choosing Good Names

- Think very deeply about variable names !

```
int n;
int noErr;
int nCompConns;
```

```
int connectionIndex;
int numErrors;
int numCompletedConnections;
```

Choosing Good Names



Purpose of Variable	**Good Names, Good Descriptors**	**Bad Names, Poor Descriptors**
Running total of checks written to date	*runningTotal, checkTotal*	*written, ct, checks, CHKTTL, x, x1, x2*
Velocity of a bullet train	*velocity, trainVelocity, velocityInMph*	*velt, v, tv, x, x1, x2, train*
Current date	*currentDate, todaysDate*	*cd, current, c, x, x1, x2, date*
Lines per page	*linesPerPage*	*lpp, lines, l, x, x1, x2*

Name Length

- "Aurea mediocritas"
  [Gorla & Benander 1990]
  - debug effort minimized when variable names have 10-16 chars
- Just make sure short names are clear enough
  - keep in mind to **read-optimize**, not write-optimize !
- Use longer names for rarely used variables [Shneiderman 1980]

Loop Indexes

- Customary: i, j, k, … *BUT*
- Use meaningful name when outside the loop

```
recordCount = 0;
while (moreScores()) {
    score[recordCount] = getNextScore();
    ++recordCount;
}

neededCapacity = recordCount * bytesPerRecord;
...
```

Loop Indexes

- Use full index names in long loops
- Avoid index cross talk in nested loops
  - often i and j get mixed up

```
for (i=0; i<teamCount; ++i) {
    for (j=0; j<eventCount[i]; ++j) {
        score[i][j] = 0;

        // ...

    }
}
```

```
for (teamIndex=0; teamIndex<teamCount ; ++teamIndex) {
    for (eventIndex=0; eventIndex<eventCount[teamIndex] ; ++eventIndex) {
        score[teamIndex][eventIndex] = 0;
```

- Only use i, j, k for loop indexes, no other variables
  - otherwise developers get confused

Computed-Value Qualifiers

- Frequently used
  - Total, Sum, Average, Max, Min, etc.
- Place at end of variable name
  - usersTotal, consumptionAverage, etc.
- Exception: Num
  - use as prefix: numUsers
  - avoid using it (choose Count or Total instead)

Use Common Opposites

- begin/end
- first/last
- locked/unlocked
- min/max
- next/previous
- old/new
- opened/closed
- visible/invisible
- source/target
- source/destination
- up/down

Purpose-Driven Naming

- Use one variable for one purpose only
- Example: solve $ax^2 + bx + c = 0$

```
tmp = Sqrt(b*b - 4*a*c);
solution[0] = (-b + tmp) / (2*a);
solution[1] = (-b - tmp) / (2*a);
// swap the solutions
tmp = solution[0];
solution[0] = solution[1];
solution[1] = tmp;
```

Purpose-Driven Naming

- Explicit names leads to clearer code

```
discriminant = Sqrt( b*b - 4*a*c );
solution[0] = ( -b - discriminant ) / ( 2*a );
solution[1] = ( -b + discriminant ) / ( 2*a );

oldSolution = solution[0];
solution[0] = solution[1];
solution[1] = oldSolution;
```

Status Variables

- Avoid using *flag* in the name
  - it always means something, so name it accordingly

```
if (flag) ...
if (statusFlag & 0x0F) ...
if (16 == printFlag) ...
if (0 == computeFlag) ...

flag = 0x1;
statusFlag = 0x80;
printFlag = 16;
computeFlag = 0;
```

Status Variables

- Use explicit variable names instead

```
if (dataReady) ...
if (characterType & PRINTABLE_CHAR) ...
if (ReportType_Annual == reportType) ...
if (True == recalcNeeded) ...

dataReady = true;
characterType = CONTROL_CHARACTER;
reportType = ReportType_Annual;
recalcNeeded = false;
```

Avoid Hybrid Coupling

- Avoid variables with hidden meanings
- E.g., bytesRead
  - indicates # of bytes received
  - except if < 0, in which case indicates error value
- E.g., accountBalance
  - indicates amount of cash in account
  - except if > 500,000, in which case amount over 500,000 is delinquent amount

Separate Namespaces

- Java packages
  - can distinguish Database::Table from GUI::Table
- C++ *namespace* keyword
- Languages without namespace support
  - use naming conventions (e.g., in C)

Use Enums Liberally

```
public class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    ...
}
```

Boolean Variables

- Use typical boolean names
  - done, error, found, success, ok, ...
- Can only take on true/false values
  - bad names: status, sourceFile, ...
  - using the "is" prefix protects you from bad names (isStatus... ?)
- Favor using *positive* boolean names
  - negatives are cumbersome to negate

Shortening Names

- Some abbreviation tricks
  - remove non-leading vowels (screen -> scrn, pages -> pgs)
  - remove articles (and, the, ...)
  - remove useless endings (-ing, -ed, ...)
- Abbreviate consistently
  - can produce a "Standard Abbreviations" document
- Should pass the telephone test [Kernighan & Plaugher]

Data Declaration

- Put in comments what cannot go in name
- Put unambiguous units in name of variable
  - if not possible, then at least put them in comments

```
Color previousColor;  // color prior to window update
Color currentColor;
Color nextColor;      // color for next update
Point previousTop;    // previous location in pixels
Point previousBottom;
...
Temperature temperatureInKelvin;
```

Names to Avoid

- Words that sound similar
  - wrap vs. rap -> remember telephone test !
- Names should differ in at least two characters
- Avoid numerals (as in file1, file2)
- Avoid commonly mis-spelled words
  - acummulate, calender, concieve, independant, reciept
- Use a single natural language for project
  - and dialect/spelling (e.g., US vs. UK English)

Using Variables

Declarations & Initialization: The Dangers

- Improper initialization → fertile source of bugs
  - result: variable has unexpected value
- Know your programming language's semantics!
- Sources of bugs
  - not initialized at all (default in Java, random in C/C++)
  - inconsistent value (e.g., constructor initializes only part of the class)

Declarations & Initialization: What You can Do

- Move initialization close to declaration

```
double area = 3.14 * radius * radius;
Board myBoard = new Board(8,8);
```

- Move declaration/initialization close to point of first use

```
int accountIndex = 0;
// code using accountIndex
// ...
boolean done = false;
while (!done) {
    // ...
}
```

Exception to the rule...

```
double dist;
try {
    // lots of code
    dist = distance();
} catch (Exception e) {
    // handle ...
}
retVal = 2*dist;
```

Declarations & Initialization: What You can Do

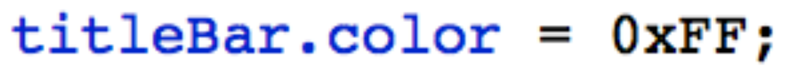
- Be disciplined with initialization
  - *always make values* ***explicit****, even if language provides defaults*
  - *initialize all class members in the constructor*
  - *method parameters: instead of initializing, these should be checked for validity*

```
double hypotenuse(double c1, double c2) throws BadParametersException {
    if (c1<=0 || c2<=0) {
        throw new BadParametersException();
    }
    return Math.sqrt(c1*c1 + c2*c2);
}
```

- Use **final** to protect from reassignment
  - *if you want constants, make them* final static ⇒ *initialized at class load time*

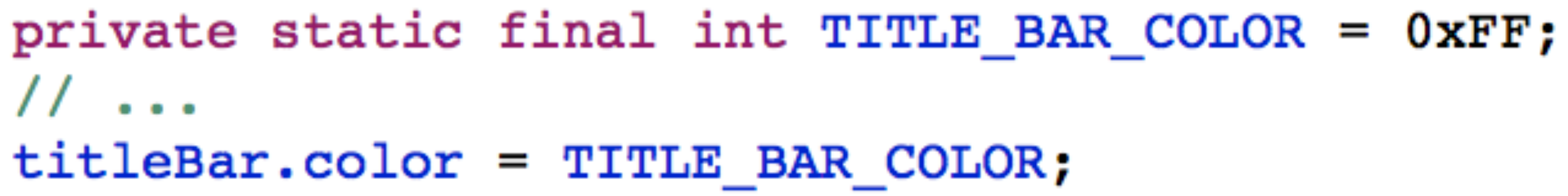
Binding Variables to Values

- **Binding vs. flexibility**
  - *earlier binding → less flexibility → less complexity → less risk of introducing errors*
  - *later binding → more flexibility → more complexity → more risk of introducing errors*
- Can bind at
  - *coding time (i.e., use magic values)*



```
titleBar.color = 0xFF;
```

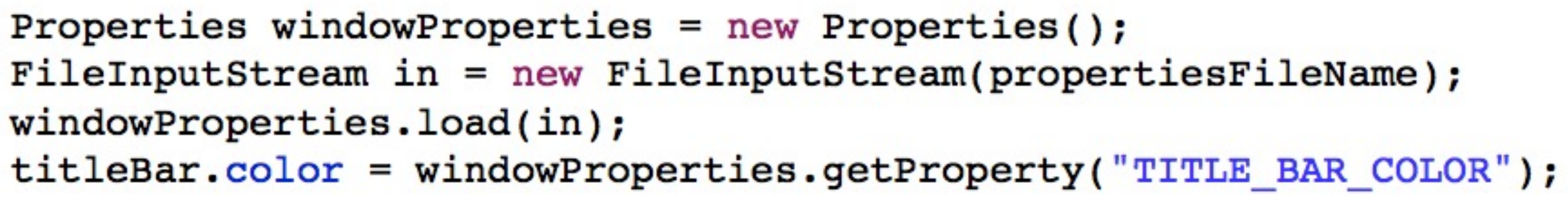
  - *compile time (i.e., use named constants)*



```
private static final int TITLE_BAR_COLOR = 0xFF;
// ...
titleBar.color = TITLE_BAR_COLOR;
```

Variable-to-Value Binding

- *at run time (e.g., use properties file or Windows registry)*



```
Properties windowProperties = new Properties();
FileInputStream in = new FileInputStream(propertiesFileName);
windowProperties.load(in);
titleBar.color = windowProperties.getProperty("TITLE_BAR_COLOR");
```

- *object instantiation time: every time you create a window*
- *just-in-time: every time you draw a window*

Using Variables

- Know your language semantics
- Initialize close to declaration
- Declare close to the first use
- Flexibility vs. reliability trade-off

Using Variables:
Variable Span, Liveness, and Scope

- You want to try to minimize variable span, live time, scope

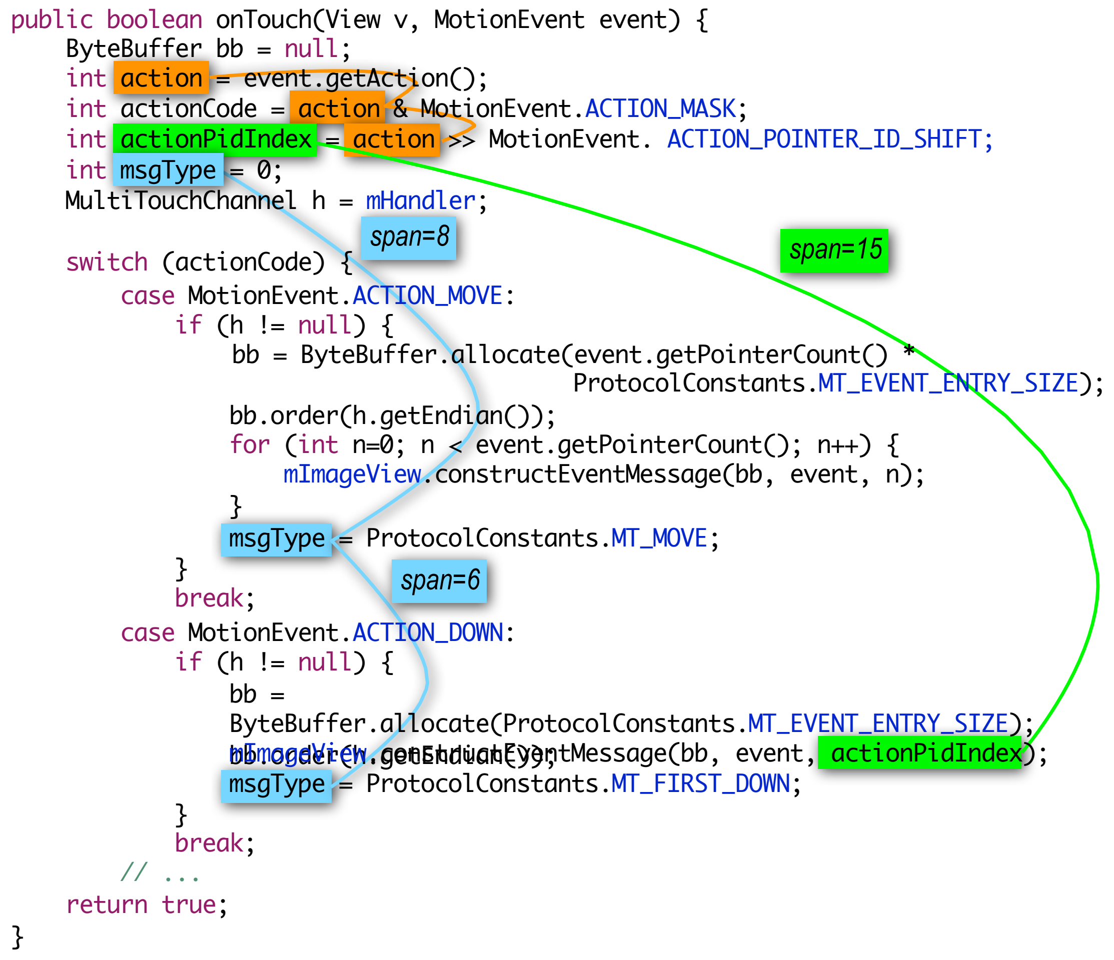
Variable Span

```
public boolean onTouch(View v, MotionEvent event) {
    ByteBuffer bb = null;
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int actionPidIndex = action >> MotionEvent. ACTION_POINTER_ID_SHIFT;
    int msgType = 0;
    MultiTouchChannel h = mHandler;

    switch (actionCode) {
        case MotionEvent.ACTION_MOVE:
            if (h != null) {
                bb = ByteBuffer.allocate(event.getPointerCount() *
                                         ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                for (int n=0; n < event.getPointerCount(); n++) {
                    mImageView.constructEventMessage(bb, event, n);
                }
                msgType = ProtocolConstants.MT_MOVE;
            }
            break;
        case MotionEvent.ACTION_DOWN:
            if (h != null) {
                bb =
                ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                mImageView.constructEventMessage(bb, event, actionPidIndex);
                msgType = ProtocolConstants.MT_FIRST_DOWN;
            }
            break;
        // ...
    return true;
}
```

A variable's span is

space (LOC) between successive references

Variable Span



```
public boolean onTouch(View v, MotionEvent event) {
    ByteBuffer bb = null;
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int actionPidIndex = action >> MotionEvent. ACTION_POINTER_ID_SHIFT;
    int msgType = 0;
    MultiTouchChannel h = mHandler;

    switch (actionCode) {
        case MotionEvent.ACTION_MOVE:
            if (h != null) {
                bb = ByteBuffer.allocate(event.getPointerCount() *
                                         ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                for (int n=0; n < event.getPointerCount(); n++) {
                    mImageView.constructEventMessage(bb, event, n);
                }
                msgType = ProtocolConstants.MT_MOVE;
            }
            break;
        case MotionEvent.ACTION_DOWN:
            if (h != null) {
                bb =
                ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                mImageView.constructEventMessage(bb, event, actionPidIndex);
                msgType = ProtocolConstants.MT_FIRST_DOWN;
            }
            break;
        // ...
    return true;
}
```

A variable's span is

space (LOC) between successive references

Variable Liveness

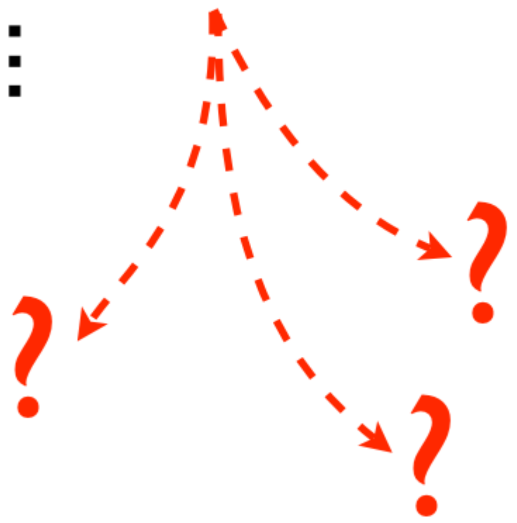
- Programs may contain
  - code which gets executed but which has no useful effect on the program’s overall result;
  - occurrences of variables being used before they are defined; and
  - many variables which need to be allocated registers and/or memory locations for compilation.

- The concept of **variable liveness** is useful in dealing with all three of these situations.

Variable Liveness

- Liveness is a data-flow property of variables: “Is the value of this variable needed?” (cf. dead code)

```
int f(int x, int y) {
   int z = x * y;
   ⋮
```



Variable Liveness

- At each instruction, each variable in the program is either live or dead.
- We therefore usually consider liveness from an instruction's perspective: each instruction (or node of the flowgraph) has an associated set of live variables.

```
   ⋮
n: int z = x * y;
   return s + t;
```

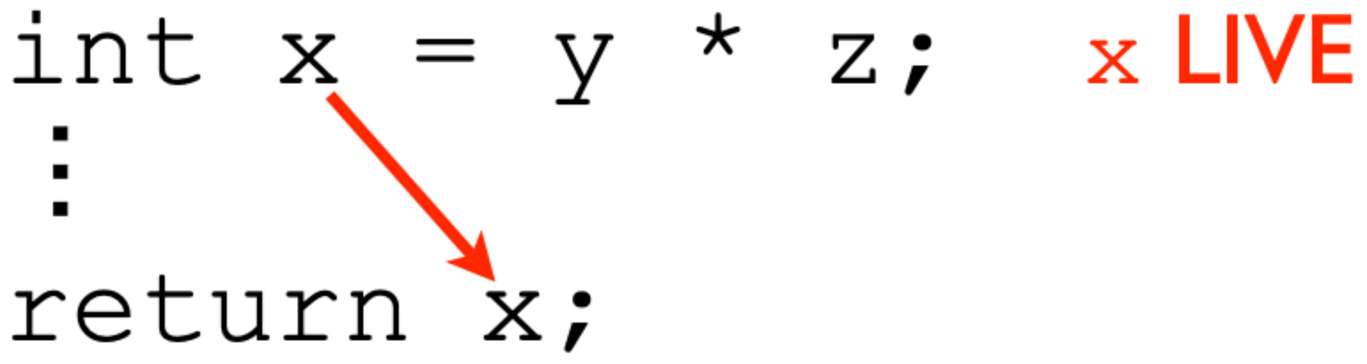
$live(n) = \{ s, t, x, y \}$

Semantic vs. Syntactic

- There are two kinds of variable liveness:
  - Semantic liveness
  - Syntactic liveness

Semantic vs. Syntactic

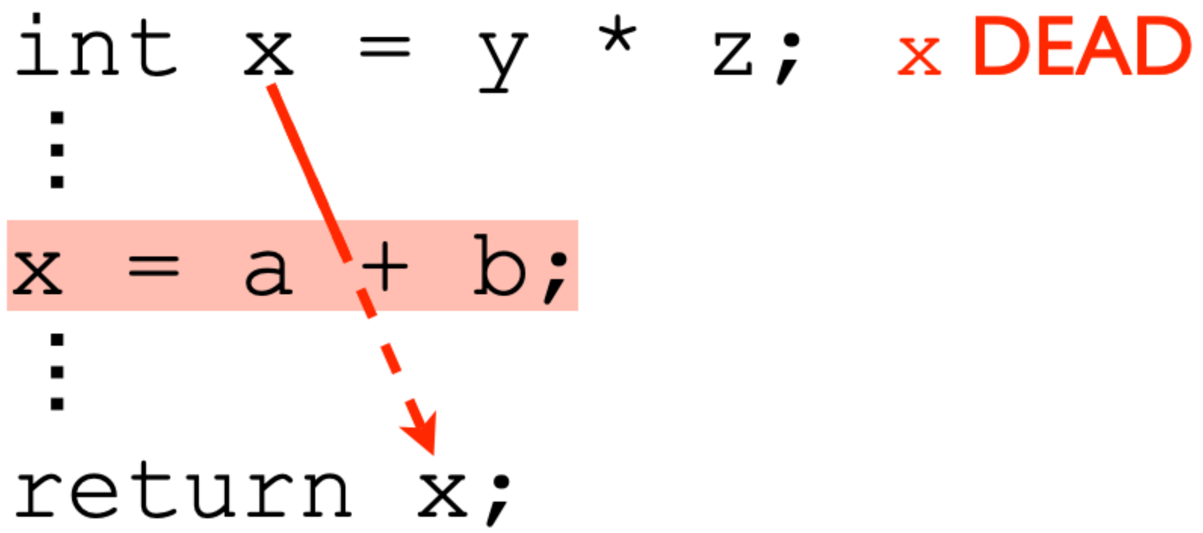
- A variable x is semantically live at a node n if there is some execution sequence starting at n whose (externally observable) behavior can be affected by changing the value of x.



```
int x = y * z;
 ⋮
return x;
```

Semantic vs. Syntactic

- A variable x is semantically live at a node n if there is some execution sequence starting at n whose (externally observable) behavior can be affected by changing the value of x.



```
int x = y * z;
 :
x = a + b;
 :
return x;
```

Semantic vs. Syntactic

- Semantic liveness is concerned with the execution behavior of the program.
- This is undecidable in general. (e.g. Control flow may depend upon arithmetic.)

Semantic vs. Syntactic

- A variable is syntactically live at a node if there is a path to the exit of the flowgraph along which its value may be used before it is redefined.
- Syntactic liveness is concerned with properties of the syntactic structure of the program.
- Of course, this is decidable.
- So what’s the difference?

Semantic vs. Syntactic

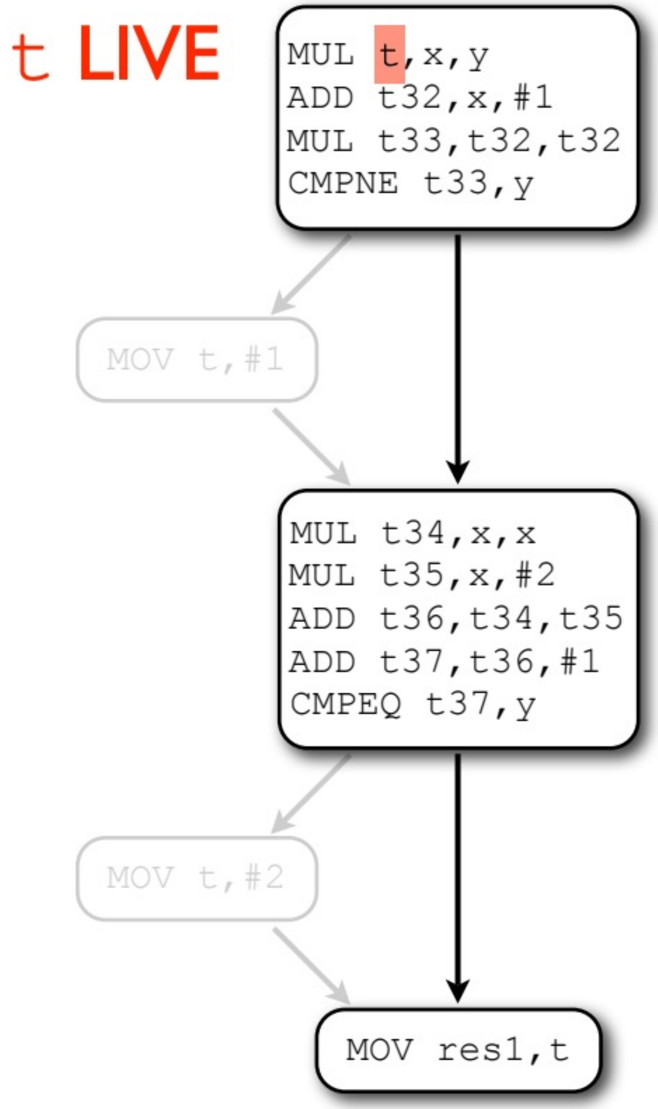
```
int t = x * y;   t DEAD
if ((x+1)*(x+1) == y) {
   t = 1;
}
if (x*x + 2*x + 1 != y) {
   t = 2;
}
return t;
```

$(x+1)^2 = y$

$(x+1)^2 \neq y$

- Semantically: one of the conditions will be true, so on every execution path t is redefined before it is returned. The value assigned by the first instruction is never used.

Semantic vs. Syntactic

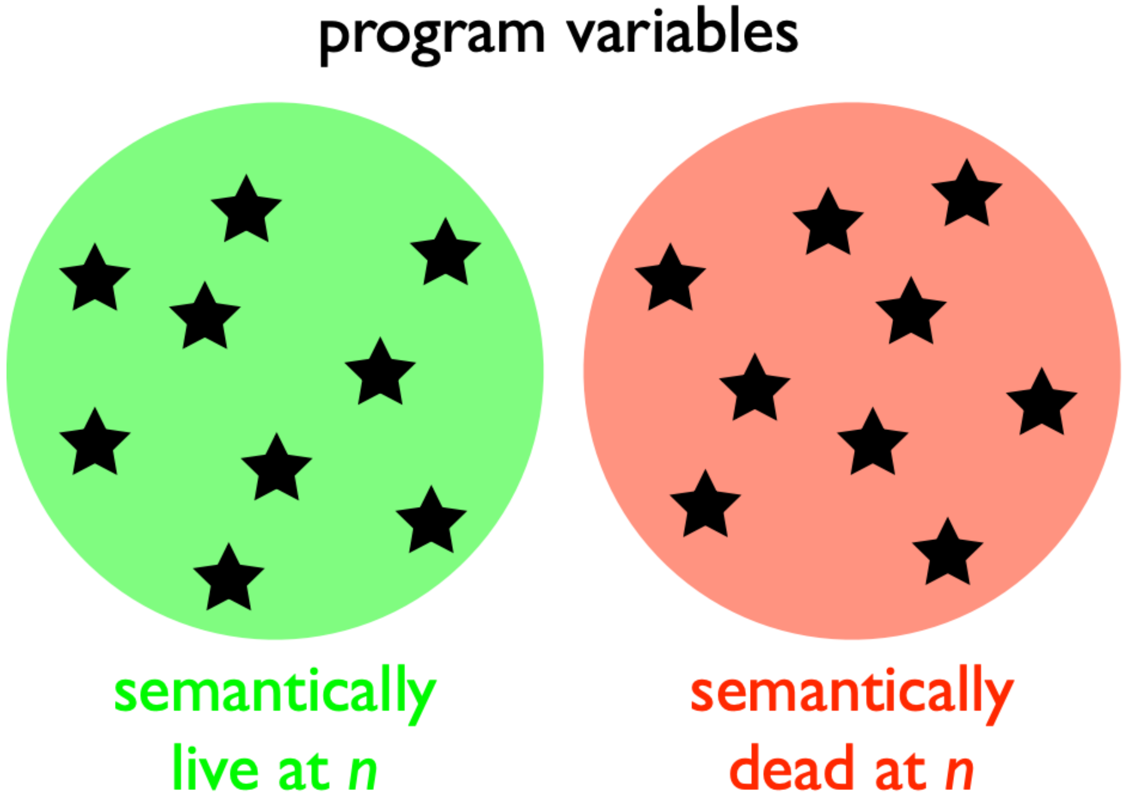


- On this path through the flowgraph, t is not redefined before it's used, so t is syntactically live at the first instruction.

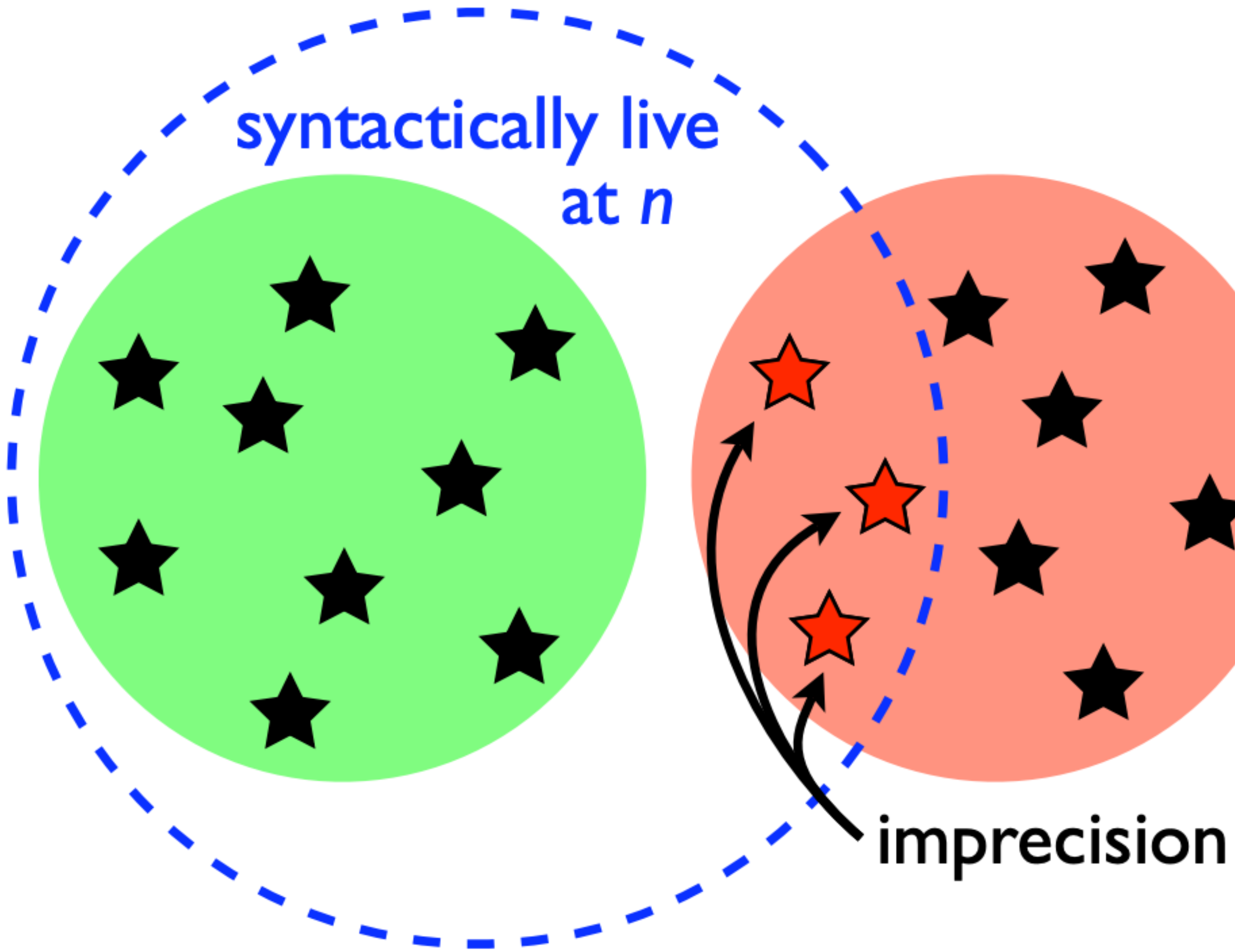
- Note that this path never actually occurs during execution.

Semantic vs. Syntactic

- So, as we've seen before, syntactic liveness is a computable approximation of semantic liveness.



Semantic vs. Syntactic



Variable Liveness

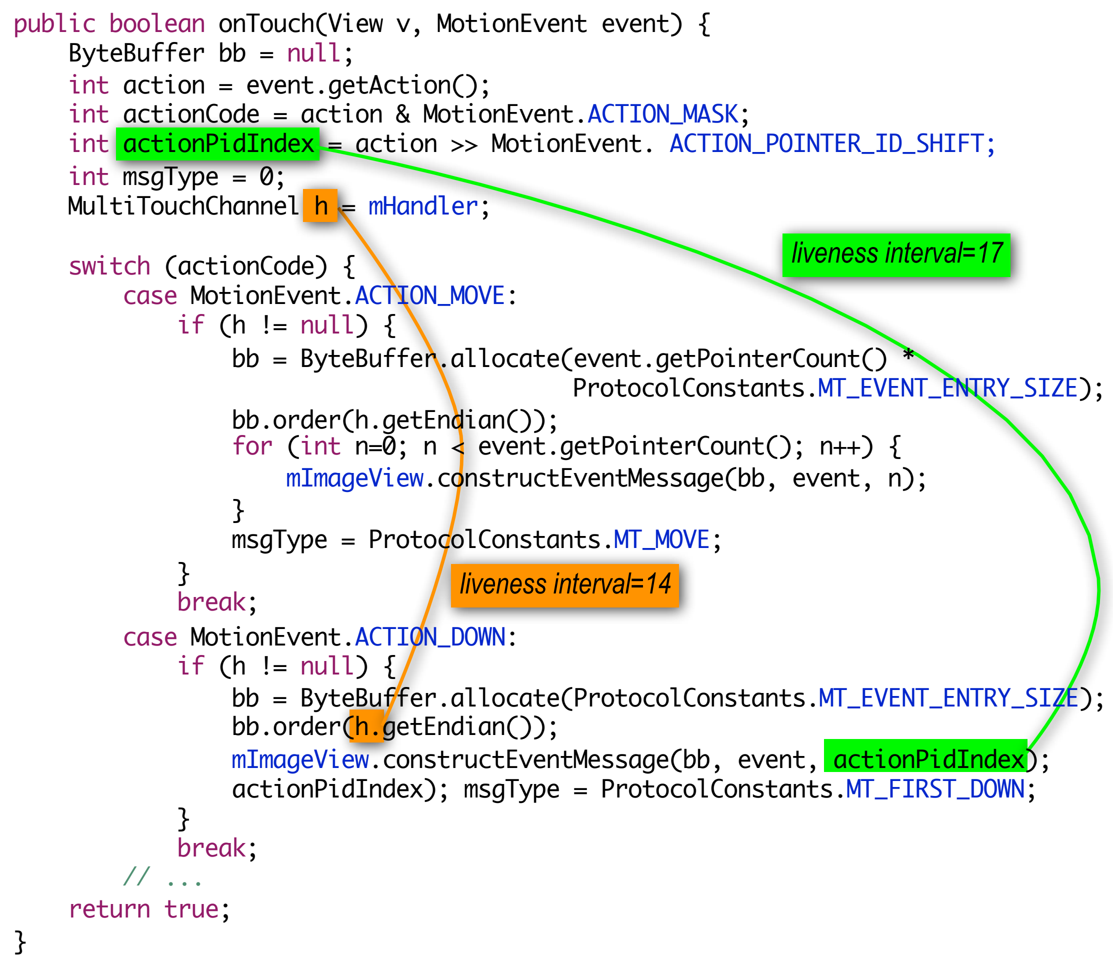
```
public boolean onTouch(View v, MotionEvent event) {
    ByteBuffer bb = null;
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int actionPidIndex = action >> MotionEvent. ACTION_POINTER_ID_SHIFT;
    int msgType = 0;
    MultiTouchChannel h = mHandler;

    switch (actionCode) {
        case MotionEvent.ACTION_MOVE:
            if (h != null) {
                bb = ByteBuffer.allocate(event.getPointerCount() *
                                         ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                for (int n=0; n < event.getPointerCount(); n++) {
                    mImageView.constructEventMessage(bb, event, n);
                }
                msgType = ProtocolConstants.MT_MOVE;
            }
            break;
        case MotionEvent.ACTION_DOWN:
            if (h != null) {
                bb = ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                mImageView.constructEventMessage(bb, event, actionPidIndex);
                actionPidIndex); msgType = ProtocolConstants.MT_FIRST_DOWN;
            }
            break;
        // ...
    return true;
}
```

A variable's liveness **interval** is

interval over which changing the variable's value will affect program behavior

# LOC (incl.) between first and last reference

Variable Liveness



```
public boolean onTouch(View v, MotionEvent event) {
    ByteBuffer bb = null;
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int actionPidIndex = action >> MotionEvent. ACTION_POINTER_ID_SHIFT;
    int msgType = 0;
    MultiTouchChannel h = mHandler;

    switch (actionCode) {
        case MotionEvent.ACTION_MOVE:
            if (h != null) {
                bb = ByteBuffer.allocate(event.getPointerCount() *
                                         ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                for (int n=0; n < event.getPointerCount(); n++) {
                    mImageView.constructEventMessage(bb, event, n);
                }
                msgType = ProtocolConstants.MT_MOVE;
            }
            break;
        case MotionEvent.ACTION_DOWN:
            if (h != null) {
                bb = ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                mImageView.constructEventMessage(bb, event, actionPidIndex);
                actionPidIndex); msgType = ProtocolConstants.MT_FIRST_DOWN;
            }
            break;
        // ...
    return true;
}
```

A variable's liveness interval is

- *interval over which changing the variable's value will affect program behavior*
- *# LOC (incl.) between first and last reference*

Variables should have short lives

- *life is a window of vulnerability*
- *short lifetime→few interactions→low complexity*
- *new code inside span might (mistakenly) alter the variable's value*
- *when reading code, might forget variable's value*

Variable Scope

```
public boolean onTouch(View v, MotionEvent event) {
    ByteBuffer bb = null;
    int action = event.getAction();
    int actionCode = action & MotionEvent.ACTION_MASK;
    int actionPidIndex = action >> MotionEvent.
    ACTION_POINTER_ID_SHIFT; int msgType = 0;
    MultiTouchChannel h = mHandler;

    switch (actionCode) {
        case MotionEvent.ACTION_MOVE:
            if (h != null) {
                bb = ByteBuffer.allocate(event.getPointerCount() *
                                         ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                for (int n=0; n < event.getPointerCount(); n++) {
                    mImageView.constructEventMessage(bb, event, n);
                }
                msgType = ProtocolConstants.MT_MOVE;
            }
            break;
        case MotionEvent.ACTION_DOWN:
            if (h != null) {
                bb =
                ByteBuffer.allocate(ProtocolConstants.MT_EVENT_ENTRY_SIZE);
                bb.order(h.getEndian());
                mImageView.constructEventMessage(bb, event,
                actionPidIndex); msgType = ProtocolConstants.MT_FIRST_DOWN;
            }
            break;
        // ...
}   return true;
```

Program space where variable is valid and can be referenced

Typical scopes

block within { }, method,
class (and possibly derived classes),
package, whole program, ...

Variable Span, Liveness, and Scope

- Variable span
  - LOC between successive references (exclusive)
- Variable liveness interval
  - LOC between first and last reference (inclusive)
- Variable scope
  - Program context where variable is valid
- In general, try to minimize these

Coding Style