Principles of Software Construction: Objects, Design, and Concurrency

A puzzling finale: What you see is what you get?

Josh Bloch    Charlie Garrod    Darya Melicher
Administrivia

• Final exam review session:
  – Saturday, Dec. 15th noon – 2:30 p.m.  Scaife 125

• Final exam:
  – Sunday, Dec 16th 5:30 – 8:30 p.m.:  GHC 4401

• Evaluate us:  https://cmu.smartevals.com/

• Evaluate our TAs:
  https://www.ugrad.cs.cmu.edu/ta/F18/feedback/
Key concepts from Tuesday
Today: A finale of puzzlers
A quick challenge: Implement binary search

/**
 * Searches the specified array of ints for the specified value
 * using the binary search algorithm. If the array is not sorted,
 * the results are undefined. If the array contains multiple
 * elements with the specified value, there is no guarantee which
 * one will be found.
 *
 * @returns the index of the search key if it is in the array;
 *          otherwise ~(insertion point). (Or for you, -1 is fine.)
 */

public static int binarySearch(int[] a, int key);
Logvinenko 1999
Fraser 1908
Fraser 1908
Todorovic 1997
Todorovic 1997
A correct binary search solution?
A correct binary search solution?

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }

    return ~(low + 1); // key not found.
}
```
Integer overflows for large values of low and high:

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }

    return ~(low + 1); // key not found.
}
```
One possible fix

- Avoid overflow, using signed ints:
  
  ```
  int mid = (low + high) / 2;
  int mid = low + ((high - low) / 2);
  ```
Lessons

• Keep it simple
• Use all the tools you know:
  – A good IDE
  – Static analysis tools like FindBugs
  – Verification tools for critical code
  – Unit tests and regression testing
  – Assert statements for known invariants
  – Code review for all code intended for other developers or users
  – Continuous integration testing for any project with multiple developers
“A Big Delight in Every Byte”

class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
            b < Byte.MAX_VALUE; b++) {
            if (b == 0x90)
                System.out.print("Joy! ");
        }
    }
}
What Does It Print?

class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++) {
            if (b == 0x90) {
                System.out.print("Joy! ");
            }
        }
    }
}

(a) Joy!
(b) Joy! Joy!
(c) Nothing
(d) None of the above
What Does It Print?

(a) Joy!
(b) Joy! Joy!
(c) Nothing
(d) None of the above

Program compares a byte with an int; byte is *promoted* with surprising results
Another Look

*bytes are signed; range from -128 to 127*

class Delight {
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE;
            b < Byte.MAX_VALUE; b++) {
            if (b == 0x90) // (b == 144)
                System.out.print("Joy! ");
        }
    }
}

// (byte)0x90 == -112
// (byte)0x90 != 0x90
You Could Fix it Like This...

- Cast int to byte
  ```java
  if (b == (byte)0x90)
      System.out.println("Joy!");
  ```

- Or convert byte to int, suppressing sign extension with mask
  ```java
  if ((b & 0xff) == 0x90)
      System.out.println("Joy!");
  ```
...But This is Even Better

```java
public class Delight {
    private static final byte TARGET = 0x90; // Won't compile!
    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)
            if (b == TARGET)
                System.out.print("Joy!");
    }
}
```

Delight.java:2: possible loss of precision
found    : int
required: byte

    private static final byte TARGET = 0x90; // Won’t compile!
    ^
The Best Solution, Debugged

```java
public class Delight {
    private static final byte TARGET = (byte) 0x90; // Fixed

    public static void main(String[] args) {
        for (byte b = Byte.MIN_VALUE; b < Byte.MAX_VALUE; b++)
            if (b == TARGET)
                System.out.print("Joy!"神通);
    }
}
```
The Moral

- **byte values are signed 😞**
- Be careful when mixing primitive types
- **Compare like-typed expressions**
  - Cast or convert one operand as necessary
  - Declared constants help keep you in line
- For language designers
  - Don’t violate principle of least astonishment
  - Don’t make programmers’ lives miserable
"Strange Saga of a Sordid Sort"

public class SordidSort {
    static final Integer BIG = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000);
    static final Integer ZERO = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) -> i1 - i2);
        System.out.println(Arrays.toString(arr));
    }
}
What does it print?

```java
public class SordidSort {
    static final Integer BIG = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) -> i1 - i2);
        System.out.println(Arrays.toString(arr));
    }
}
```

(a) [-2000000000, 0, 2000000000]
(b) [2000000000, 0, -2000000000]
(c) [-2000000000, 2000000000, 0]
(d) None of the above
What does it print?

(a) \([-2000000000, 0, 2000000000]\)
(b) \([2000000000, 0, -2000000000]\)
(c) \([-2000000000, 2000000000, 0]\)
(d) None of the above: Unspecified;
   In practice, \([2000000000, -2000000000, 0]\)

Comparator is broken!
   It relies on int subtraction
   int too small to hold difference of 2 arbitrary ints
Another Look

public class SordidSort {
    static final Integer BIG = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO = 0;

    public static void main(String[] args) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) -> i1 - i2);
        System.out.println(Arrays.toString(arr));
    }
}

Subtraction overflows.
A possible fix?

```java
public class SordidSort {
    static final Integer BIG = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) ->
            i1 < i2 ? -1 : (i1 == i2 ? 0 : 1));
        System.out.println(Arrays.toString(arr));
    }
}
```
…Another bug!

```java
public class SordidSort {
    static final Integer BIG = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) ->
            i1 < i2 ? -1 : (i1 == i2 ? 0 : 1));
        System.out.println(Arrays.toString(arr));
    }
}
```

Unspecified behavior

== checks for identity, not equality, of object references!
You could fix it like this...

```java
public class SordidSort {
    static final Integer BIG   = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO  = 0;

    public static void main(String[] args) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, (i1, i2) ->
                i1 < i2 ? -1 : (i1 > i2 ? 1 : 0));
        System.out.println(Arrays.toString(arr));
    }
}
```

Works, but fragile!

Prints `[-2000000000, 0, 2000000000]`
...But this is better

public class SordidSort {
    static final Integer BIG = 2_000_000_000;
    static final Integer SMALL = -2_000_000_000;
    static final Integer ZERO = 0;

    public static void main(String args[]) {
        Integer[] arr = new Integer[] {BIG, SMALL, ZERO};
        Arrays.sort(arr, Integer::compareTo);
        System.out.println(Arrays.toString(arr));
    }
}

Prints [-2000000000, 0, 2000000000]
Moral (1 of 2)

- ints aren’t integers
  - Think about overflow
- The comparison technique \((i1, i2) \rightarrow i1 - i2\)
  requires \(|i1 - i2| \leq \text{Integer.MAX_VALUE}\)
  - For example: all values non-negative
- Don’t write overly clever code
- Use standard idioms
  - But beware; some idioms are broken
Moral (2 of 2)

• ints aren’t Integers
  – Think about identity vs. equality
  – Think about null

• For language designers
  – Don’t violate the principle of least astonishment
  – Don’t insist on backward compatibility
```java
class Indecisive {
    public static void main(String[] args) {
        System.out.println(decision());
    }

    static boolean decision() {
        try {
            return true;
        }
        finally {
            return false;
        }
    }
}
```

"Indecision"
What does it print?

(a) true
(b) false
(c) None of the above
What does it print?

(a) true
(b) false
(c) None of the above
(d) Who cares?!?
What does it print?

(a) true  
(b) false  
(c) None of the above

The finally is processed after the try.
Another look

class Indecisive {
    public static void main(String[] args) {
        System.out.println(decision());
    }

    static boolean decision() {
        try {
            return true;
        } finally {
            return false;
        }
    }
}
The moral

- Don't rely on obscure language or library details
- Here: Avoid abrupt completion of finally blocks
  - Don't return or throw exception from finally
  - Wrap unpredictable actions with nested try
public class LongDivision {
    private static final long MILLIS_PER_DAY
        = 24 * 60 * 60 * 1000;
    private static final long MICROS_PER_DAY
        = 24 * 60 * 60 * 1000 * 1000;

    public static void main(String[] args) {
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
public class LongDivision {
    private static final long MILLIS_PER_DAY
        = 24 * 60 * 60 * 1000;
    private static final long MICROS_PER_DAY
        = 24 * 60 * 60 * 1000 * 1000;

    public static void main(String[] args) {
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}

What does it print?

(a) 5
(b) 1000
(c) 5000
(d) Throws an exception
What does it print?

(a) 5  
(b) 1000  
(c) 5000  
(d) Throws an exception

Computation overflows
Another look

public class LongDivision {
    private static final long MILLIS_PER_DAY
        = 24 * 60 * 60 * 1000;
    private static final long MICROS_PER_DAY
        = 24 * 60 * 60 * 1000 * 1000; // >> Integer.MAX_VALUE

    public static void main(String[] args) {
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
How do you fix it?

public class LongDivision {
    private static final long MILLIS_PER_DAY
        = 24L * 60 * 60 * 1000;
    private static final long MICROS_PER_DAY
        = 24L * 60 * 60 * 1000 * 1000;

    public static void main(String[] args) {
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
The moral

• When working with large numbers, watch out for overflow—it’s a silent killer

• Just because variable can hold result doesn’t mean computation won’t overflow

• When in doubt, use larger type
“It’s Elementary” (2004; 2010 remix)

```java
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
        System.out.println(01234 + 43210);
    }
}
```

The Periodic Table of the Elements
What does it print?

public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
        System.out.println(01234 + 43210);
    }
}

(a) 17777  44444
(b) 17777  43878
(c) 66666  44444
(d) 66666  43878
What does it print?

(a) 17777  44444
(b) 17777  43878
(c) 66666  44444
(d) 66666  43878

Program doesn’t say what you think it does!
Also, leading zeros can cause trouble.
Another look

```java
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
        System.out.println(01234 + 43210);
    }
}
```

1 - the numeral one
1 - the lowercase letter el
Another look, continued

```java
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
        System.out.println(01234 + 43210);
    }
}
```

01234 is an octal literal equal to 1,234_{8}, which is 668
How do you fix it?

```java
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
        System.out.println(1234 + 43210); // No leading 0
    }
}
```

Prints 66666 44444
The moral

• Always use uppercase el (L) for long literals
  – Lowercase el makes the code unreadable
  – $5432L$ is clearly a long, $5432l$ is misleading

• Never use lowercase el (l) as a variable name
  – Not this: `List<String> l = ... ;`
  – But this: `List<String> list = ... ;`

• Never precede an int literal with 0 unless you actually want to express it in octal (base 8)
  – And add a comment if this is your intent
Lessons (reprised)

• Keep it simple
• Use all the tools you know:
  – A good IDE
  – Static analysis tools like FindBugs
  – Verification tools for critical code
  – Unit tests
  – Assert statements for known invariants
  – Code review for all code intended for other developers or users
  – Continuous integration testing for any project with multiple developers