Principles of Software Construction

’tis a Gift to be Simple or Cleanliness is Next to Godliness

Midterm 1 and Homework 3 Post-Mortem

Josh Bloch          Charlie Garrod
Administrivia

- Homework 4a due Thursday, 11:59 p.m.
  - Design review meeting is mandatory
Outline

• Midterm exam post-mortem
• Permutation generator post-mortem
• Cryptarithm post-mortem
Midterm exam results

MINIMUM 16.0  
MEDIAN 42.5  
MAXIMUM 63.0  
MEAN 41.24  
STD DEV 10.9
Anyone know a simpler expression for this?

```java
if (myDog.hasFleas()) {
    return true;
} else {
    return false;
}
```
Hint: it’s not this

```java
return myDog.hasFleas() ? true : false;
```
Please do it this way from now on

_We reserve the right to deduct points if you don’t_

```java
return myDog.hasFleas();
```
Also, we saw some hash functions like these

\[
\text{return } 31 \times x + 31 \times y; \quad // \text{Multiplication doesn't help!}
\]

\[
\text{return } 31 \times x + 32 \times y; \quad // \text{Multiplication hurts!}
\]

\[
\text{return Objects.hash(map); } \quad // \text{Objects.hash unnecessary!}
\]
Here's how these should look

return 31 * x + 31 * y;

return 31 * x + 32 * y;

return Objects.hash(map);

return 31 * x + y;

return 31 * x + y;

return map.hashCode();
What should a hash code look like, in general?

*Standard Java hash functions - not great, but good enough*

- Single-field object
  - `field.hashCode()`
- Two-field object
  - `31*field1.hashCode() + field0.hashCode()`
- 3-field object
  - `31*(31*field2.hashCode() + field1.hashCode) + field0.hashCode`
  - `= 31^2 * field2.hashCode() + 31 * field1.hashCode() + field0.hashCode`
- N-field object
  - **Repeatedly multiply total by 31 and add in next field**
    - `= \sum 31^i \cdot \text{hashCode}(field_i)`
    - Alternatively: `Objects.hash(field_0, field_1, \ldots field_{N-1})`
- For much more information, see *Effective Java* Item 9
Some solutions were correct but repetitious

- Repetition isn’t just inelegant, it’s toxic
- Avoiding repetition is essential to good programming
- Provides not just elegance, but quality
- Ease of understanding aids in
  - Establishing correctness
  - Maintaining the code
- If code is repeated, each bug must be fixed repeatedly
  - If you forget to fix one occurrence, program is subtly broken
- Train yourself to feel a twinge of pain each time you copy-paste
A good, basic solution – fields and constructor (1/3)
What's the best internal representation if you want to support more base units?
Outline

• Midterm exam post-mortem
• Permutation generator post-mortem
• Cryptarithm post-mortem
Design comparison for permutation generator

- **Command pattern**
  - Easy to code
  - Reasonably pretty to use

- **Iterator pattern**
  - Tricky to code because algorithm is recursive and Java lacks *generators*
  - Really pretty to use

- **Performance is similar**
A complete (!), general-purpose permutation generator

*using the command pattern*
How do you test a permutation generator?

Make a list of items to permute (integers should do nicely)

For each permutation of the list {
    Check that it’s actually a permutation of the list
    Check that we haven’t seen it yet
    Put it in the set of permutations that we have seen
}

Check that the set of permutations we’ve seen has right size (n!)

Do this for all reasonable values of n, and you’re done!
And now, in code – this is the whole thing!

```java
static void exhaustiveTest(int size) {
    List<Integer> list = new ArrayList<>(size);
    for (int i = 0; i < size; i++)
        list.add(i);
    Set<Integer> elements = new HashSet<>(list);
    Set<List<Integer>> alreadySeen = new HashSet<>();
    doForAllPermutations(list, (perm) -> {
        Assert.assertEquals(perm.size(), size);
        Assert.assertEquals(new HashSet(perm), elements);
        Assert.assertFalse("Duplicate", alreadySeen.contains(perm));
        alreadySeen.add(new ArrayList<>(perm));
    });
    Assert.assertEquals(alreadySeen.size(), factorial(size));
}

@Test public void test() {
    for (int size = 0; size <= 10; size++)
        exhaustiveTest(size);
}
```
Pros and cons of exhaustive testing

- Pros and cons of exhaustive testing
  - Gives you absolute assurance that the unit works
  - Exhaustive tests can be short and elegant
  - You don’t have to worry about what to test
    - Rarely feasible; Infeasible for:
      - Nondeterministic code, including most concurrent code
      - Large state spaces
  
- If you can test exhaustively, do!
  
- If not, you can often approximate it with random testing
Outline

• Midterm exam post-mortem
• Permutation generator post-mortem
• Cryptarithm post-mortem
A fast, fully functional cryptarithm solver in 6 slides

To refresh your memory, here’s the grammar

\[
\text{cryptarithm ::= } \langle \text{expr} \rangle \ "=" \ \langle \text{expr} \rangle \\
\text{expr ::= } \langle \text{word} \rangle \ [\langle \text{operator} \rangle \ <\text{word}>]* \\
\text{word ::= } <\text{alphabetic-character}>+ \\
\text{operator ::= } \"+\" \ | \ \"-\" \ | \ \"*\"
\]
Cryptarithm class (1) – fields
Conclusion

• Good habits really matter
  – “The way to write a perfect program is to make yourself a perfect programmer and then just program naturally.” – Watts S. Humphrey, 1994

• Don’t just hack it up and say you’ll fix it later
  – You probably won’t
  – but you will get into the habit of just hacking it up

• Representations matter! Choose carefully.
  – If your code is getting ugly, think again
  – “A week of coding can often save a whole hour of thought.”

• Not enough to be merely correct; code must be clearly correct
  – Nearly correct is right out.