Documenting code, Javadoc, Defensive Programming, Asserts, Exceptions & Try/Catch

Most important reason to comment

A) To summarize the code

B) To explain how the code works

C) To mark locations that need further work

D) To help the reader know as much as the writer did

E) To store non-code information with the code

Intent

Ideal Comment Density

- Software engineering studies have studied what commenting frequency makes the code most readable. Guess what was the ideal comment density?

A) 1 comment per 10 statements

B) 1 comment per 5 statements

C) 1 comment per 3 statements

D) 1 comment per 2 statements

E) 1 comment per statement

How could more comments be bad?

if good code => more ~~some~~ comments would be redundant

problem:

1. more stuff to read

2. maintenance

DRY

The best documentation is …

code that doesn't need comments to be understood

- **Write straightforward code**
- **Use expressive variable and function names**
- **Follow common conventions**
  - getSomething()
  - isEmpty()

Marking end of blocks with comments

```
if (condition) {
   callSomeFunction(with, some, arguments);
} // if
```

Is this appropriate commenting?

A) Yes

B) No

Javadoc example (what could be improved?)

description of what the function does.

```
/**
 * Test the primality of a number.  See: https://en.wikipedia.org/wiki/Prime_number
 *
 * @param candidate  the number to be tested for primality
 * @return           true if the candidate is prime, false otherwise
 */
public static boolean isPrime(int candidate) {
  // Negative numbers, 0, 1, and even numbers (other than 2) are not prime
  if (candidate < 2 ||
      ((candidate % 2 == 0) && (candidate != 2))) {
    return false;
  }

  // If a number can be evenly divided by a number other than 1 and itself,
  // then it is not prime.  It is sufficient to test using only odd numbers (as
  // we've already eliminated even candidates) and to only test up to the square
  // root of the candidate.
  int sqrt = (int) Math.ceil(Math.sqrt(candidate));
  for (int divisor = 3; divisor <= sqrt; divisor += 2) {
    if (candidate % divisor == 0) {
      return false;
    }
  }

  return true;
}
```

Pseudo-code approach to programming

1. write a series of comments outlining the steps
2. Implement each step in code, leaving the comment in place

Defensive Programming

- **Key Idea: Protect yourself from invalid inputs**

  behave reasonably: (1) correct (2) robust

- **Where do invalid inputs come from?**
  - Command line arguments
  - User input during run ← inform
  - Programming errors ← crash
  - Bad data files
  - Configuration

- **check all data from external sources, input parameters**

Pre-conditions/Post-conditions

- **Pre-condition: a condition/predicate that must be true just prior to the execution of some section of code**
  - If a pre-condition is violated, the effect of a section of code is undefined.

- **Post-condition: a condition that must be true after the code**

```
f( List<String>  listOfStrings) {
    if ( listOfString != null) {  ←—— precondition.
     listOfStrings. add("new String");
}
```

What to do if a pre-condition is violated?

robustness: $\left\{ \begin{array}{l} \text{we could manufacture data.} \\ \text{do nothing} \end{array} \right.$ (handle internally)

inform the user

crash

$\left.\right\}$ correctness

passing the buck

exceptions,

Chirest.get ("...").

Asserts

- Java includes an 'assert' statement to check pre-conditions

```
assert list != null;
```

- If the condition evaluates to false, it throws an AssertionError ← crash program

- Java also provides two argument version; second argument (any object type) is included into the thrown AssertionError object

```
assert list != null : "List was null";
```

Which is better?

A
```
public static void main(String [] args) {
    assert args.length >= 2 && args.length <= 3 :
      "This program takes 2 or 3 arguments";
    ...
```

B
```
public static void main(String [] args) {
    if (args.length < 2 || args.length > 3) {
S.o.pln(*);
      printUsage();
      ~~return~~; ← exit out of the program.
       System.exit(-1);
    }
    ...
```

C Both are fine

D Both are problematic

Error / Exceptions "Throwables"

- **Events that occur during program execution**
- **Disrupt the normal flow of the program**
  - (e.g. divide by zero, array access out of bound, etc.).
- **In Java, an exception is an object that wraps an error event**
  - contains information about the error including its type
- **Typically handled through the use of try/catch**
- **Important piece of the interface of a method**
  - Method signature includes what exceptions it might throw

Kinds of Exceptions

"abstract" base class

```
          +-----------+
          | Throwable |
          +-----------+
           /         \
          /           \
  +-------+         +-----------+
  | Error |         | Exception |
  +-------+         +-----------+
   /  |  \          / |       \
 \________/      \______/      \
                              +------------------+
unchecked       checked       | RuntimeException |
                              +------------------+
                                /   |   |      \
                               \________________/

                                unchecked
```

How does try/catch work?

array.length = 20;

```
public static int Sum(int [] array){
     int sum = 0;
     try {                         i < array.length
          for (int i = 0; true; i++) {
               sum += array[i];  20 X
          }
     } catch (ArrayIndexOutOfBoundsException e) {
          return sum;
     } catch (Exception e){
            ...
     }
}
```

WARNING: BAD STYLE; BAD DESIGN; I WILL FAIL YOU..

To catch or to propagate/re-throw

- **Fundamental question of exception handling:**
  - Do I have enough information here to decide how to respond to this error?
  - If not, then propagate / re-throw
  - If yes, then handle it here

Throwing Exceptions

- You can manually throw exceptions if you want:

```
throw new Exception(“Invalid status”);
```

- You can define your own kind of exception:

```
public class MyOwnException extends Exception {
    // put anything you want in here!
}

throw new MyOwnException();
```