CSSE2002 Course Notes

Ismael Khan

Javadoc

```
/* *
* Do a thing
* @param s Name of species
* @param f Fraction of Avagadro's Number
* @return Time required(in microseconds)
*/

public long doThing(String s, float f)
```

Javadoc comments must

- Begin with /**
- End with */
- Must be immediately above the thing being commented
- Use tags @
- Some tags take params, some just text.

Procedural Abstraction

Procedural abstraction makes programs easier to read and mantain by keeping the size and complexity of methods small.

Guidelines

- Methods should be decomposed according to functionality not lines of code.
- Each method should perform one task (can be an aggregate task)
- Be suspicious of
  - Methods that are hard to name
  - Repeated code
  - Any method which is overly long or complex.
  - Conditional statements on the types of method arguements

Ideally specification

- Allow the implementation of a method to be read/written without needing to look at the implementations of other methods.
- Allow a method to be re-implemented without changes any dependant methods.
- Should
  - Rule out all implementations that are unnacceptable (i.e be suffieciently restrictive).
  - Not prelude acceptable implentations

- Be easy for programmers to understand
- Draw attention to possible consequences of implementation decisions. (Eg if it may affect performance)

Attempt to keep out incorrect implementations, for instance

```
/** Return an index (i) of ar such that
  * ar[i] == x, if any
  */

public int search(int[] ar, int x)
```

What happens if x is not in ar? Don't assume and be clear in the documentation.
Better:

```
/** Return an index (i) of ar such that
  * ar[i] == x, if any
  * else, return -1
  */
```

However don't be overly restrictive and tell readers how you wrote some method.

```
/** Calculate the square root of a number within
  * a given error
  * @param sq Number whose squareroot is to be found
  * @param e The allowable error
  * @return rt such that 0 <= (rt*rt - sq) <= se
  */
public double sqrt(double sq, double e)
```

Formality

Specifications of software range in formality

- Informal - eg. normal comments
- Semiformal - structured English documentation using Javadoc tags.
- Formal - mathematical constraints using Java Syntax for boolean expressions.

Informal Specifications

```
/** Withdraw an amount from this account and
  * return how much is left
  */
pulbic int withdraw(int amount)
```

What happens when

- amount is negative?
- amount is bigger than balance?

Is the balance changes when there is a failure?

Semi-formal specifications

```
/** Withdraw an amount from this account.
  * @param amount The amount to withdraw
  * @return Balance of this account after successful withdrawal
  */
public int withdraw(int amount)
```

Clearer but the same questions still apply.

Contracts

Formal specifications of code can be written using contracts. If tthe program calling the method satisfies the precondition, then the method guarantees to satisfy the post condition (watch out for Exceptions here). If the code calling method does not satisfy the precondition then the method guarantees nothing. This still does not make the questions go away.

Formal specifications

```
/** Withdraw an amount from this account.
  * @require amount >= 0 && amount <= getBalance()
  * @ensure getBalance() == \old(getBalance()) - amount && \result ==
  getBalance()
  */
public int withdraw(int amount)
```

Document using:
javadoc -tag require -tag ensure Thing.java. IntelliJ → pracs.

More complex specifications

Java syntax for boolean expresion plus...

```
\result - return value of method
a ==> - a implies b (if a then b)
a <==> b - a if and only if b
\old(x) - value of x before method occurs
\forall C c; - for all objects c of Class C
\exists C c; - there exists an object c of Class C
```

Defensive programming

Because people are awful

Defensive programming is explicitly checking for invalid inputs and bad situations, ensuring the software does not behave dangerously regardless of input.

What if someone calls the method without checking the precondition?

Particularly when dealing with external input sources or when guarding critical resources, it may be better to be defensive outside the wall and use contracts inside it.

The problem with null

Lots of programs and progammers have with NullPointerException. The root causes (contract progamming) are

- Null not covered by contract - everyone is confused
- Null is covered - progammre is not paying attention
- Unexpected nulls propagate - hard to track down

Best practice for API design:

- Default is that null is NOT a valid argument / result
- Specify when null is allowed / expected / returned and document its meaning.

Best practice for implementation:

- Check for null, implicitly or explicity
- Check early
- NullPointerException or IllegalArgumentException

Example 1.

```
/**
...
  * @param name a non-empty string
*/
public void setName(String name) {
  if (name.length() == 0) { // throws NPE if name is null
    throw new IllegalArgumentException("name empty");
  }
}
```

Substitution Principle

```
class Parent {
  ...
  char f(int x);
}
```

Suppose

- The precondition (@require) for f is $x > 0$
- The postcondition (@ensure) for f is

  ```
  \result
  ```

  ¿ 'A' && ¡ 'Z'

What happens if we override f in a subclass of Parent? Even when we change the implementation, we should still follow the contract commitments of the original version.

Suppose `Child1 extends Parent`, but `Child1.f()` can deal with negative numbers as well.