CSE 331 Cheat Sheet
April 14, 2011

Javadoc, Mutability, Class Design, Enums, Exceptions

Javadoc

- Whenever you write a class to be used by clients, you should write full Javadoc comments for all of its public behavior.
- Don't repeat yourself or write vacuous comments.
- Each class constant or enumeration value can be commented
- **precondition**: Something assumed to be true at the start of a call.
- **postcondition**: Something your method promises will be true at the end of its execution, if all preconditions were true at the start.
- **Assertions**: used to check preconditions

On a method or constructor:

tag	description
`@param` ***name description***	describes a parameter
`@return` ***description***	describes what value will be returned
`@throws` ***ExceptionType reason***	describes an exception that may be thrown (and what would cause it to be thrown)
`{@code` ***sourcecode***`}`	for showing Java code in the comments
`{@inheritDoc}`	allows a subclass method to copy Javadoc comments from the superclass version

On a class header

tag	description
`@author` ***name***	author of a class
`@version` ***number***	class's version number, in any format

Mutability

A modification to the state of an object.

- *Horstmann Tip 3.4.3: Whenever possible, keep accessors and mutators separate.* Ideally, mutators return void.
- *Effective Java Tip #15: Minimize mutability.*

Making a class immutable

• 1. Don't provide any methods that modify the object's state.
• 2. Ensure that the class cannot be extended.
• 3. Make all fields final.
• 4. Make all fields private. (ensure encapsulation)
• 5. Ensure exclusive access to any mutable object fields. Don't let a client get a reference to a field that is a mutable object.

final: Unchangeable; unable to be redefined or overridden.

Law of Demeter: An object should know as little as possible about the internal structure of other objects with which it interacts.

Good things that you should strive for when designing classes:
- 1) cohesion: Every class should represent a single abstraction.
- 2) completeness: Every class should present a complete interface.
- 3) clarity: Interface should make sense without confusion.
- 4) convenience: Provide simple ways for clients to do common tasks.
- 5) consistency: In names, param/returns, ordering, and behavior.

A bad thing that you should try to minimize:
- 6) coupling: Amount and level of interaction between classes.

Enums

Effective Java Tip #30: Use enums instead of int constants.

```
public enum Name {
    VALUE, VALUE, ..., VALUE
}
```

Can add fields (using a private constructor) and/or additional methods:

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private int cents;

    private Coin(int cents) {
        this.cents = cents;
    }

    public int getCents() {...}
}
```

How to use enums:
- Compare them with == or compareTo() (ordering is based on the order they were declared in)
- Use them in a switch statement
- Use EnumSet to maintain and manipulate a set of enum values
- Use EnumMap instead of indexing by ordinal number

Enum methods:

method	description
int compareTo(E)	all enum types are Comparable by order of declaration
boolean equals(o)	works, but not needed: can just use ==

String name()	equivalent to toString()
int ordinal()	returns an enum's 0-based number by order of declaration (first is 0, then 1, then 2, …)

method	description
static E valueOf(s)	converts a String into an enum value
static E[] values()	an array of all values of your enumeration

Exceptions

Catch exceptions by surrounding dangerous code in try/catch blocks:

```
try {
    ...
    mightThrowException(s);
    ...
} catch (ExceptionType1 e1) {
    // react to, or do something with, e1...
} catch (ExceptionType2 e2) {
    // do something with e2...
} finally {
    // This code will run regardless of whether there was an exception
}
```

Possible ways to handle an exception:

- retry the operation that failed
- re-prompt the user for new input
- print a nice error message
- quit the program

Effective Java Tip #65: Don't ignore exceptions.

Exceptions are objects, too! Use inheritance relationships to make your exception-catching code handle multiple types of exception objects. Create your own exception class by extending RuntimeException.

Exception methods:

method	description
String getMessage()	text describing the error
String toString()	exception's type and description
void printStackTrace()	prints a stack trace to System.err
And many more!	