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:

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

On a class header

<table>
<thead>
<tr>
<th>tag</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@author <strong>name</strong></td>
<td>author of a class</td>
</tr>
<tr>
<td>@version <strong>number</strong></td>
<td>class's version number, in any format</td>
</tr>
</tbody>
</table>

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._

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

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

```java
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:**

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int compareTo(E)</td>
<td>all enum types are Comparable by order of declaration</td>
</tr>
<tr>
<td>boolean equals(o)</td>
<td>works, but not needed: can just use ==</td>
</tr>
</tbody>
</table>
### 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, ...)

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static E.valueOf(s)</td>
<td>converts a String into an enum value</td>
</tr>
<tr>
<td>static E[] values()</td>
<td>an array of all values of your enumeration</td>
</tr>
</tbody>
</table>

### Exceptions

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

```java
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:

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getMessage()</td>
<td>text describing the error</td>
</tr>
<tr>
<td>String toString()</td>
<td>exception’s type and description</td>
</tr>
<tr>
<td>void printStackTrace()</td>
<td>prints a stack trace to System.err</td>
</tr>
</tbody>
</table>

*And many more!*