API design considerations

API design guidelines

The following guidelines are meant to make the user experience more pleasant. They are aimed to help the user make use of existing tools and increase productivity. As a first guideline, **keep in mind that somebody other than you *will* use your code**.

### Bug reports

A vital component of the project is the bug database. The software is actually more than a mere reporting system for bugs. You can use it to suggest improvements and monitor progress as well as organize workflow for tasks in your own project.
Rule: Use the JIRA system often and extensively
Rule: Search the database before reporting a bug

### Object orientation

Object orientation is a design approach that attempts to model real world objects in the code.
Try to think in objects.
Make use of the visibility of objects.
Hide your implementation from the user.
On the other hand, not every class needs to have a set of private data members with a full-blown set of Setters and Getters. It is ok to have a class with only public data members.
Rule: What belongs together logically should be handled together
Example:

👍	👎

`class SimpleParameterClass {`
`public int spaceX;`
`public double tanPolarPhi;`
`public float cosLine1Line2;`
`}`
`...`
`SimpleParameterClass parms = new SimpleParameterClass();`
`parms.spaceX = 17; // you can simply set it`
`...`
`System.out.printf("%f", parms.cosLine1Line2); // or get it -- no need for encapsulation`
	`double[] parms = new double[] {17, 3.4, 2.5};`
	`parms[2] = 2.5; // Which one is it ???`

### Optimization

Optimization obfuscates code. It also forces you to change your intuitive approach.
Rule: First, write your code. Then profile it. Then, *if necessary*, optimize it.

### Status codes

In the good old Fortran and C days, obscure statuscodes and bitfields were common sight. While it is true that your code should be optimized at the persistence level, make sure you include definitions of your statuscodes in the API. The use of Enum, EnumSet and EnumMap is encouraged.

👍	👎

```
enum MyClassFlags {doThingX, doThingY, doThingZ};
    class MyClass {
    MyClassFlags flags;
    void setFlags(EnumSet f) {flags = f;}
}
...
EnumSet<MyClassFlags> myFlags = new EnumSet<MyClassFlags>();
myFlags.add(MyClassFlags.doThingX); // yes, it is more verbose.
That is a good thing !
MyClass c;
c.setFlags(myFlags);
```

```
class Obscure {
    int obscureFlag;
    void setObscureFlag(int o)
{obscureFlag = o;}
}
...
Obscure o
o.setObscureFlag(3);
```

⚠ Never hardcode numbers like this

"Overloaded Operators"

The Java language does not support the concept of overloading operators. If you decide to write a custom operator for your class, consider the following:

👍

```
vec1.add(vec2);
```

or

```
import static x.y.add; add(vec1, vec2);
```

👎

```
vec1.add(vec1, vec2);
```

Using strings to access parameters

Strings should be used for input/output operations **only**. They are slow to parse and cannot be checked for typos at compile time.
Using them to set flags and parameters is not recommended.
See the section about status codes above. If you need to set different parameters with the same function, use an EnumMap

Deprecation

No matter how well you design your classes, sooner or later you will come to the point where you need to change the signature of a function. For example, your Vector class was understood to be cartesian, but now you decide to support spherical polar and cylindrical coordinates as well. You need to change your acces points to your class, but that will break hundreds of already existing programs. Just what are you supposed to do ?
Help comes in form of deprecation tags that help prepare your users transition to the new code. There are two kinds of deprecation tags:
The JavaDoc @deprecated tag tells readers of the documentation that this function is not supposed to be used any more.
The @Deprecated annotation is a function modifier that issues warning messages upon compilation, so that all your users are aware that this function may be removed in the near future.
Rule: Deprecate your functions for at least one full release version, i.e. if you checked in your code shortly before the 0.8 release, don't break backwards compatibility until at least release 0.9.
Rule: Don't deprecate a function without providing an alternacite
Example:

```
/**
 * @deprecated Old version.
 *             Doesn't comply with Coding Guidelines any more
 *             Use {@link #getFurthestPoint(SpacePoint)} instead
 */
@Deprecated double[] furthestPoint(double[] point) {
    ...
}
```

See also Sun's Guidelines on deprecation

JavaDoc

While it is true that your code **should** be self-documenting, this does not save you from writing **proper** documentation and code comments. Paraphrasing the function name is not helpful.
👎 How Not to do it:

```
// returns x
int getRL() {}
```

👍 This makes more sense, doesn't it ?

```
// calculates the radiation length from the previously set parameters. NOTE: The parameter Y gets invalidated by the call.
// returns the radiation length in a range between 0 and 10
int getRadiationLength() {}
```