

Design Principles

1. Javadoc all public classes and methods. Class comment should be at least two sentences, and provide information not already clear from its definition.
2. Use interface types over concrete classes wherever possible. Exception: immutable "value" objects. Classes with no interface.
3. Fields must always be private. Exception: constants. Methods, classes should be as private as possible.
4. Class should never have public methods not in the interface (aside from constructor).
5. Composition over inheritance.
6. Catch and handle/report errors as early as possible. Use Java compiler checks, enums, final first, runtime checks second.
7. Use class types over strings.
8. Check inputs.

Design Principles (cont)

9. Use exceptions only for exceptional situations -- not for flow control.
10. Checked vs unchecked: checked: reasonable expectation that the program can recover. Unchecked: programmer error (may still be recoverable).
11. Don't leave things in an inconsistent state for any substantive length of time.
12. Beware of references, copies, and mutation. Make defensive copies.
13. Separate responsibilities: one class, one responsibility.
14. Use class hierarchies and dynamic dispatch over tagged classes, complex if/switch statements.
15. Don't duplicate code.
16. Open for extension, closed for modification: make changes without modifying existing code; write code to support later changes without modification.
17. Extensibility: design to make likely later changes easier.

Design Principles (cont)

18. Write tests first, cover the range of situations, edge cases. Write code to be testable (avoid System.out); do not expose fields or add public methods just to allow for testing.
19. Loose coupling over tight coupling (avoid System.out). Write reusable components when possible.
20. You can't change an interface once it's published.
21. If you override equals(), override hashCode(), and vice-versa.
22. Reuse existing exceptions, classes, libraries, and designs.

Scanner Methods

```
public boolean hasNext() Returns true if the scanner
public String next() Finds and returns the next complete line of text as a String
```

Scanner Methods (cont)

```
> Scanner scanner = new Scanner(System.in);
Readable r = new StringReader("Hello");
Appendable a = new StringBuffer();
```

CLASS INVARIANTS

NOT INVARIANTS	INVARIANT
value is small	A logical statement is a claim that is true or false
value never decreases	The instantaneous state of an object is the combination of values of all its fields at some point in time
value is an int	The invariant is ensured by constructors in the sense that whenever a public constructor returns, the logical statement holds



By **Edgar** (edgarjoel21)

Not published yet.

Last updated 27th February, 2018.

Page 1 of 3.

Sponsored by **Readable.com**

Measure your website readability!

<https://readable.com>

CLASS INVARIANTS (cont)

Preserving the logical statement means that the method doesn't introduce nonsense - instead, we know that if given a object in a good state then it will leave the object in a good state as well

Enables a form of reasoning called rely-guarantee.

- If the constructor ensures some property
- and every method preserves the property
- then every public method, on entry, can rely on the property

Composition over inheritance

- Composition over inheritance

- delegate design pattern: take the previous code we want to use and make it a field instead of extending it

- has-a instead of is-a

- always get copies of every private field instead of passing in the real thing

```
public boolean
remove(int i) {
    return delete -
ate.remove(i);
}
```

Composition over inheritance (cont)

```
> public boolean contains(int i) {
    return delegate.contains(i);
}
```

Map methods

```
clear()
containsKey( Object
key)
containsValue( Object
value)
entrySet() returns a set
view of the mappings
(Set<Map.Entry<K,
V>>)
equals (Object o)
get(Object key)
hashCode()
isEmpty()
keySet() returns a set
of the keys Set<K>
put(k key, V value)
Associates the value
with the key
putAll (Map<? extends
K, ? extends V> m)
Copies into new map
remove (Object key)
size()
values() Returns a
Collection view of the
values (Collection<
V>)
```

Equals

```
@Override
public boolean equals -
(Object that) {
    if (this == that) {
        return true;
    }
    if (!(that
 instanceof Duration)) {
        return
false;
    }
    return ((Dura -
tion) that).isEqualTo -
nds() == this.isE -
conds();
}
```

Stack Methods

```
empty() Tests if this
stack is empty
(boolean) |
peek() Looks at top
object of stack without
removing it (E) |
pop() Removes the object
at the top of this stack
and returns that object
(E) |
push(E item) Pushes an
item onto the top of
this stack (E) |
search (Object o)
Returns the 1-based
position where an object
is on this stack (int)
|
```

Stack Methods (cont)

```
> add, add, addAll, addAll,
toString, trimToSize
```

Deque< E> is an interfac
e (double ended queue)
Deque< Integer> stack
= new ArrayDeque< Integer >();

Tips

- check for overflow
- canonicalize means converting data with multiple representations into a standard or normal form
- bug may be that a method used a non-copy of something
- interfaces can be extended to add methods to something



By Edgar (edgarjoel21)

Not published yet.

Last updated 27th February, 2018.

Page 2 of 3.

Sponsored by **Readable.com**

Measure your website readability!

<https://readable.com>

Static

Static: Java's version of global variables

Static Methods: Called on the class rather than an instance and thus don't have a this to work on. Ex: Long.hashCode()

Static classes: behave like normal classes just nested in its enclosing classes namespace.

Outer.nested is how you refer to it. Outer.nested can see outer's private members and vice versa

- constants should be public static final and in all caps

HashCode

```
@Override
public int hashCode() {
    return Object.hashCode( field, field, field);
}
```

Must use fields that equals uses

ABSTRACT TEST

```
protected abstract
FreecellOperations<Card>
freecellModel();

public static class
SingleMove extends
AbstractFreecellModelTests2 {
    @Override
```

ABSTRACT TEST (cont)

```
> protected FreecellOperations<Card> freecellModel() {
    return FreecellModelCreator.create(GameType.SINGLEMOVE);
}

public static class MultiMove
extends AbstractFreecellModelTests2 {
    @Override
    protected FreecellOperations<Card> freecellModel() {
        return FreecellModelCreator.create(GameType.MULTIMOVE);
    }
}
```

Array

- an array is a mutable, fixed-length, constant-time-indexed sequence of values of type t
- new int[]{2, 4, 6, 8} gives you fixed size array
- new int[9]{} gives you a empty array of 9 spaces with null or 0
- uses length function
- mutability (intArray[3] = 17;) means int at index 3 is now 17
- assert ArrayEquals



By Edgar (edgarjoel21)

Not published yet.

Last updated 27th February, 2018.

Page 3 of 3.

Sponsored by [Readable.com](https://readable.com)

Measure your website readability!

<https://readable.com>