Effective Java

Item 7: Avoid Finalizers

- A finalizer in Java is *not* the same as a destructor in C++. In Java:
    - memory is reclaimed using garbage collection, and
    - non-memory resources are reclaimed using `try-finally` blocks.
- There's now guarantee that the finalizer will be executed *in time*.
    - The finalizer thread might run with lower priority.
    - The execution of finalizers depends both on the implementation on the specific platform and on the garbage collection (and its implementation).
    - Never do anything time-critical in a finalizer!
    - Don't try to free limited resources (memory, file descriptors) in a finalizer!
- There's not even a guarantee that the finalizer will be executed *at all*.
    - Never update persistent states in a finalizer!
- Just as `System.gc()` doesn't necessarily run garbage collection, `System.runFinalization()` doesn't run the finalizers.
    - Don't use them, and also don't use the deprecated methods `System.runFinalizersOnExit()` and `Runtime.runFinalizersOnExit()`!
- Uncaught exceptions thrown during finalization are ignored and the finalization will be terminated.
    - The programmer won't even notice that something bad happened.
- Finalizers come with a performance penalty.
- Provide explicit termination methods for classes that need to free non-memory resources.
    - Example: the `close()` method of `InputStream` and `OutputStream`
    - The class implementing `close()` must store it's state and throw an `IllegalStateException` when `close()` was called on it.
    - Termination methods are usually called within a `finally` block to guarantee their execution.
- However, finalizers can be used in some cases:
    - If the client forgot to call the termination method, the finalizer can call it *and should log a warning* so that the client code can be fixed.
    - When working with native objects, whose memory resources cannot be freed by garbage collection.
- Unlike default constructors, the `finalize()` method of the subclass doesn't call the `finalize()` method of the superclass; it must be called manually (inside a `finally` block).

Item 8: Obey the general contract when overriding `equals`

- `Object`'s implementation of `equals()` is adequate:

    - if an instance only has to be equals to itself (`equals()` does the same as `==`: `o.equals(o) == (o == o)`),
    - if the `equals()` method of the superclass is appropriate,
    - or if the class is private or package-private and the `equals()` method is never called.
- Classes implementing the singleton pattern and enums don't need an `equals()` method.
- Implementations of `equals()` must adhere to its *general contract*:
    - *Reflexiveness*: for `o != null`, `o.equals(o)` must return `true`.
    - *Symmetry*: for `a != null && b != null`, `a.equals(b)` must return the same as `b.equals(a)`.
    - *Transitivity*: for `x != null && y != null && z != null`, `x.equals(z)` must return `true` if `x.equals(y)` and `y.equals(z)` return `true`.
    - *Consistency*: for `m != null && n != null`, multiple invocations of `m.equals(n)` must return the same (unless information used in `equals()` has been changed in the meantime).
    - *«Non-nullity»*: for `o != null`, `o.equals(null)` must return `false`.
- Many classes (say, those of the Collections framework) depend on `equals()` adhering to the *general contract*!
    - The behaviour of classes violating the *general contract* is undefined.
- Getting `equals()` right can be hard when using inheritance.
    - Using composition over inheritance makes it easier.
    - `a.equals(b)` should only return true, if `a` and `b` are instances of the same class (`a.getClass() == b.getClass()`).
        * Example: A date might be equals to a timestamp, but a timestamp (containing time information) not to a date, which violates the *symmetry* rule.
    - Don't use unreliable resources in `equals()`.
        * Example: `URL` uses the IP address in its `equals()` implementation, but the IP might change, where's the host name is still the same (DNS).
- When using the `instanceof` operator (to make sure that instances of the same class are compared), checking for `null` is not necessary.
    - For `o == null`, `o instanceof MyClass` always returns `false`.
- Recipe for high-quality `equals()` methods:
    1. Check for identity (to save time):
        - `if (this == o) return true;`
    2. Use `instanceof`:
        - `if (!(o instanceof MyClass)) return false;`
    3. Cast the object, it's safe now:
        - `MyClass other = (MyClass)o;`
    4. Check all the significant fields for equality:
        - for `boolean`, `int`, `long`, `short`, `byte` and `char`, compare with `==`
        - for `float` and `double`, use `Float.compare()` or `Double.compare()`, respectively: `Float.compare(this.a, other.getA()) == 0`

    - for arrays, iterate over all elements or, better, use `Arrays.equals()`; the element order matters: `Arrays.equals(this.arr, other.getArr()) == true`
    - for object references, first check for null and then call `equals()` on it: `this.field == null ? other.getField() == null : this.field.equals(other.getField()`
    - for performance reasons, first check for fields that are more likely to differ, and immediately return `false` after the first difference is detected.
5. Write a test case to check the adherence to the *general contract*.

- Make sure to use the correct method signature when overriding `equals()`: `public boolean equals(Object o)`
    - Don't replace `Object` with anything else, otherwise `equals()` won't be called.
    - Use the `@Override` annotation to make sure you have the correct signature.
- Consider letting the IDE generate the `equals()` method.
- Since Java 7, consider using `Objects.equals(this.a, other.getA())` for instance fields.
- When you override `equals()`, also override `hashCode()`.

Item 9: Always override `hashCode` when you override `equals`

- When overriding `equals()`, always also override the `hashCode()` method, otherwise the class might not work properly for `HashMap`, `HashSet` and `Hashtable`, among others.
- The `hashCode()` implementation must adhere to the following contract:
    - As long as no information used by `equals()` is changed on an object, it's `hashCode()` method must return the same value in the same execution context.
    - If `a.equals(b)`, `a.hashCode()` must be the same as `b.hashCode()`.
    - If `!a.equals(b)`, `a.hashCode()` and `b.hashCode()` are *not* required to differ; they should, however, for performance reasons when working with hash tables and the like.
- When two objects that are equal have different hash codes – which is the case when `equals()` has been overridden but not so `hashCode() -, the lookup of those objects in a`HashMap`(or the like) might be slower or even fail (return`null‘).
- A good `hashCode()` implementation produces equal hash codes for equal objects and unequal hash codes for unequal objects; it can be achieved using this recipe:
    1. Store a constant nonzero value in an integer variable, say, 17.
        - `int result = 17;`
    2. For each significant field `f` (i.e. a field used in `equals()`), do the following:

   a. Compute a hash code for the field:
      - for `boolean`: `int c = (f ? 1 : 0);`
      - for `boolean`, `int`, `short`, `byte` and `char`: `int c = (int)f;`
      - for `long`: `int c = (int)(f ^ (f >>> 32));`
      - for `float`: `int c = Float.floatToIntBits(f);`
      - for `double`: `long l = Double.doubleToLongBits(f); int c = (int)(l ^ (l >>> 32));`
      - for object references: `int c = (f == null ? ' : f.hashCode());`
      - for arrays, apply the same steps for every item – or better use `int c = Arrays.hashCode(f);`
   b. Combine the computed hash code into `result` as follows:
      - `result = 31 * result + c;`
3. Return `result`
4. Write a test case to make sure that equal instances have equal hash codes – and unequal instances unequal hash codes.

- 17 and 31 are uneven prime numbers and have some interesting properties that help reduce collisions when computing hash codes. Use those values, unless you are a mathematician.
- Since Java 7, consider using `Objects.hashCode(f1, f2, f3, ...)` by passing all fields that are also used in `equals()`.
- Don't try to optimize `hashCode()` for performance. *Good* hash codes are more important than *quickly generated* hash codes, for any computation time saved for creating worse hash codes can make `HashMap` (et al.) perform worse.

Item 15: Minimize mutability

- An immutable class is a class whose instances cannot be modified.
- All the information of an instance is provided when it is created.
- A class can be made immutable following these rules:
  1. Don't provide any methods that modify the object's state (mutators).
  2. Make sure the class cannot be extended by declaring it with the `final` keyword or make the constructor `private`.
     - Provide a `public static` factory method instead of a `public` constructor.
  3. Make all fields `final` and initialize them in the constructor.
  4. Make all fields `private` so that clients cannot obtain access to objects referred to by them.
  5. Ensure exclusive access to any mutable components.
     - If there are fields that refer to mutable objects, make sure that clients of the class cannot obtain references to those.
     - Don't initialize a field to a object reference provided by the client.
     - Don't return references to objects in accessor methods.

- Instead of setter methods that modify the instance, create a new instance based on the setter's parameter(s) and return it (*functional approach*).
- Immutable objects have only one state and hence no state transitions to deal with, which makes them simpler.
- Immutable objects are thread-safe and don't require synchronisation. So they can be shared freely.
- Immutable objects are greap map keys and set elements, because their values don't change.
- However, immutable classes require an object for each distinct value, which can be costly.

Item 18: Prefer interfaces to abstract classes

- A Java class can only inherit from one (abstract) class but many interfaces.
  - Providing an interface-definition as an abstract class takes flexibility away from the client.
  - If the client already inherits from another class, he either needs change inheritance or build a deeper class-hierarchy.
- Existing classes easily can implement new interfaces but not so easily inherit from a new abstract class.
  - Implementing another interface is *extending the code*, inheriting from another abstract class is *changing the code*.
- Combining abstract classes to types requires large and unflexible class hierarchies, wheres interfaces can be combined easily and freely (interfaces have multi-inheritance).

Item 25: Prefer List to Array

- If you put any object (that is not an instance of `Object`) into an `Object` array, you get a runtime error. If you put any such object into an `Object` array list, you get a compile-time error.
  - A compile-time error allows you to fix the problem early and is hence preferrable.
- `List` implementations (and Collections and generic classes in general) use type-erasure and hence work well together with legacy code.
- Arrays don't support generics, lists do.
- Arrays and (generic) lists don't mix well. Lists are much more powerful, so try to use lists instead of arrays whereever possible.
  - Varargs, implemented as arrays, therefore should be used with caution.

Item 38: Check parameters for validity

- Restrictions on method parameters must be checked and the beginning of the method and documented.
- Parameters need to be checked before the method works with them or stores them somehow.
  - Failing early helps detecting problems early.
  - Otherwise errors hard to track down can occur later in the execution.
  - Returning a result based on illegal parameters can be dangerous. The client thinks that everything went ok and continues to work with a pointless return value.
- Throw exceptions to make the client aware of violated restrictions on the parameters and document them with the `@throws` tag.
  - `NullPointerException` for `null` references
  - `IndexOutOfBoundsException` for invalid indices
  - `IllegalArgumentException` for all other illegal values
- For exported (i.e. non-`private`) methods, throw exceptions; for not-exported (i.e. `private`) methods, use assertions.
- Don't restrict parameters arbitrarily, but naturally. Example:
  - «Nobody will ever order more than 1000 items at a time!» What if? *Don't* do an upper-bound check here!
  - «Nobody will ever order a negative amount.» True. *Do* a lower-bound check here!

Item 51: Beware the performance of string concatenation

- String concatenation using the `+` operator doesn't scale well.
- Strings are immutable, for every modification to a String, a new object has to be created.
- To concatenate long strings, use a `StringBuilder`.
  - Create a `StringBuilder` with optional length: 'StringBuilder sb = new StringBuilder(length);
  - Add to a `StringBuilder`: `sb.append("..."); // any type goes, not only String`
  - Get the concatenated `String` when done: `sb.toString()`

Item 52: Refer to objects by their interfaces

- If an appropriate interface type exist, make the declaration using the interface name instead of the class name.
  - do so for parameters, return values, variables and fields
- This will make the code more flexible, because the specific implementation can be replaced by modifying only one line: the instantiation.

- Besides the instantiation, only use the class name as the reference type when you really need implementation specific methods.
- When no interface is available, it could be a good idea to refer to a (maybe abstract) base class instead of a specific implementation.

Item 57: Use exceptions only for exceptional conditions

- Exceptions are supposed to be used for exceptional cases, not for control flow.
- Relying on exceptions does *not* perform better then explicit tests.
  - JVM implementations have no incentive to optimize the performance for exceptional cases.
  - Code inside a `try-catch` block cannot be optimized like other code.
  - The JVM *can* optimize explicit control flow tests.
- Using exceptions for control flow obfuscates the meaning of the code.
- If exception handling is used for control flow, *real* exceptions are dismissed silently, which makes it hard to detect bugs.
- Don't write APIs that throw exceptions for non-exceptional cases.

Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

- There are three kinds of throwables in Java:
  1. Checked exceptions (subclasses of `Exception`): for conditions from which the caller can recover. Handle those in a `catch` block.
  2. Runtime exceptions (subclasses of `RuntimeException`): for programming errors. Don't catch those, but fix the code that causes them.
  3. Errors (subclasses of `Error`): `StackOverflowError`, `OutOfMemoryError` and the like. Handle those like runtime exceptions.
- When writing subclasses of `Exception`, notice that exceptions are full-fledged classes.
  - Don't just provide a text message, but consider storing relevant information in the exception class and provide accessors to them.

Item 59: Avoid unnecessary use of checked exceptions

Item 60: Favor the use of standard exceptions

Item 61: Throw exceptions appropriate to the abstraction

Item 64: Strive for failure atomicity