Effective Java

Recently, I've re-read awesome java book Effective Java by Joshua Bloch. The book contains 78 *independent* items, discussing various aspects of programming in java. Something like *mini-design patterns* with emphasis on their pros and cons.

Few notes from each item as a refresher.

Item 1: Consider static factory methods instead of constructors

- Static factory methods have more informative names than constructors
- Same parameters list could be applied
- Not required to create new objects, could return cached instance
- Static factory methods could return object subtype
- Reduced verbosity for generics due to type inference
- Classes without public/private constructor can't be subclassed, but it is good, because it enforces to "favor composition over inheritance"
- Hard to distinguish from other static methods. To avoid confusion use common names like `newIntance`, `valueOf`, etc.

Item 2: Consider a builder when faced with many constructor parameters

- **Telescope Constructor** causes verbosity
- **JavaBeans** may cause inconsistent state, no possibbility to make a class immutable
- **Builder Pattern** is flexible and right way to handle optional parameters

Item 3: Enforce the singleton property with a private constructor or an enum type

Caution: Discussed singleton **without** lazy initialization

- Throw an exception in a private constructor to avoid reflection call to constructor

- If standard serialization is needed make all fields transient and override `readResolve` method
- Best way to use single element enum as a singleton

```
enum Singleton {
  INSTANCE
}
```

Item 4: Enforce noninstantiability with a private constructor

- Include a single private constructor to a class to prevent it from instantiation
- Throw an exception in constructor if it is called
- Almost always used technique for utility classes

Item 5: Avoid creating unnecessary objects

- `"hello"` is better than `new String("hello")`
- `Boolean.valueOf("true")` is better than `new Boolean("true")`
- Immutable objects could be reused for free
- Mutable objects could be reused if you *really* sure, they won't be modified
- prefer primitives to boxed primitives
- watch out for hidden autoboxing

Item 6: Eliminate obsolete object references

- Garbage collector is not savior from memory leaks
- Nullify obsolete references
- Invalidate cache periodically
- Deregister outdated listeners and callbacks

Item 7: Avoid finalizers

- Finalizers are not destructors
- No guarantee finalizers will be executed promptly
- No guarantee finalizers will be executed at all
- `System.gc` just a hint, not a gc call
- Finalizers cause **severe** performance penalty
- Use own explicit methods for finalization like `close()`

Item 8: Obey the general contract when overriding equals

- Overridden `equals` should follow *equivalence relation*
  - Reflexive, `x.equals(x) == true`
  - Symmetric, `x.equals(y) == y.equals(x)`
  - Transitive, `x.equals(y) and y.equals(z) == x.equals(z)`
  - Consistent, consequent calls `x.equals(y)` should produce the same value
  - `x.equals(null) == false` if `x` not null
- There is no way to extend an instantiable class and add a value component while preserving the equals contract
- Do not write an equals method that depends on unreliable resources. This was discussed at the Java Magic: Part I
- Always override `hashCode` when you override `equals`
- Don't substitute param type in `equals`, that cause method **overload** instead of **override**. Use `@Override` annotation to be safe.

Item 9: Always override hashCode when you override equals

- Equal objects must have equal hashcodes
- Unequal objects could have equal hashcodes
- Missing `hashCode` implementation breaks functionality of hash-based collections
- The worst possible legal hash function `return 42`
- Bad hashcode could degrade performance in hash-based collections
- Hashcode could be cached for immutable classes
- Do not try to develop your own `state-of-the-art` hash function unless you are a mathematician

Item 10: Always override toString

- Actually, not required
- Easier to inspect objects
- Include all needed info to `toString`
- Document what exactly `toString` returns and in which format
- Provide programmatic access to all of the information contained in the value returned by `toString`

Item 11: Override clone judiciously

- `Cloneable` is broken

- If you override the `clone` method in a nonfinal class, you should return an object obtained by invoking `super.clone`
- Class that implements `Cloneable` is expected to provide a properly functioning public `clone` method
- `clone` should not corrupt original object
- Pay attention to *deep* and *shallow* copy
- Provide copy constructor or copy factory instead of implementing `clone`

Item 12: Consider implementing Comparable

- Implementing `Comparable` indicates that objects have natural ordering
- `Comparable` allow to use your class in many generic algorithms: search, sorting, etc.
- `compareTo` should be consistent with `equals`
- Implement `compareTo` that returns `-1`, `0` and `1` and do not cause integer overflow
- For non-natural ordering or inability to implement `Comparable` use `Comparator`

Item 13: Minimize the accessibility of classes and members

- Make each class or member as inaccessible as possible
- If a package-private top-level class is used by only one class, consider making the top-level class a private nested class of the sole class that uses it
- If a method overrides a superclass method, it is not permitted to have a lower access level in the subclass than it does in the superclass
- Instance fields should never be public
- Classes with public mutable fields are not thread-safe
- public static final arrays are mutable

Item 14: In public classes, use accessor methods, not public fields

- public fields are acceptable if class is not public
- if a class is accessible outside its package, provide accessor methods

Item 15: Minimize mutability

- Immutable classes are easier to design, implement and use. They are less error-prone and more secure
- To make a class immutable follow the rules

    - Don't provide any mutators

    - Ensure that the class can't be extended
    - Make all fields `final`
    - Make all fields `private`
    - Ensure exclusive access to any mutable components. Return defensive copies

- Immutable objects are thread-safe
- Immutable objects are shared freely
- Not only can you share immutable objects, but you can share their internals
- The only real disadvantage of immutable classes is that they require a separate object for each distinct value
- Classes should be immutable unless there's a very good reason to make them mutable.
- If a class cannot be made immutable, limit its mutability as much as possible.
- Make every field final unless there is a compelling reason to make it nonfinal

Item 16: Favor composition over inheritance

- Unlike method invocation, inheritance violates encapsulation
- Subclasses depend on their superclasses, which could be changed and as result broken functionality in subclasses
- Use composition and forwarding instead of inheritance, especially if an appropriate interface to implement a wrapper class exists.
- Use inheritance when class is designed for inheritance

Item 17: Design and document for inheritance or else prohibit it

- Class must document its self-use of overridable methods
- Good API documentation *for inheritance* should describe what a given method does and how it does it.
- The only way to test a class designed for inheritance is to write subclasses
- Constructors must not invoke overridable methods
- If supperclass implements `Cloneable` or `Serializable` neither `clone` nor `readObject` may invoke an overridable method, directly or indirectly
- Prohibit subclassing in classes that are not designed and documented to be safely subclassed
- Prohibit subclassing by making class final

Item 18: Prefer interfaces to abstract classes

- Existing classes can be easily retrofitted to implement a new interface

- Interfaces are ideal for defining mixins
- Interfaces allow the construction of nonhierarchical type frameworks
- Interfaces enable safe, powerful functionality enhancements via wrapper classes
- Abstract classes are useful for sceletal implementation
- You could safely add a method to abstract class with default implementation (*the same applies to interfaces since Java 8 release, with help of default methods*)
- Once an interface is released and widely implemented, it is almost impossible to change

Item 19: Use interfaces only to define types

- Do not use interface for defining constants
- If in a future release the class is modified so that it no longer needs to use the constants, it still must implement the interface to ensure binary compatibility
- If a nonfinal class implements a constant interface, all of its subclasses will have their namespaces polluted by the constants in the interface
- Add constant to class if they are strongly tied to it
- Make constants as enum or noninstantiable utility classes

Item 20: Prefer class hierarchies to tagged classes

- Tagged class use internal state to indicate its type
- Tagged classes are verbose, error-prone, and memory inefficient
- Hierarchy classes provide more compile time checks

Item 21: Use function objects to represent strategies

- Function objects are simulate functions in OOP
- Function objects should be stateless
- Primary use of function objects is to implement the Strategy pattern

Item 22: Favor static member classes over nonstatic

- A nested class should exist only to serve its enclosing class
- There are four kinds of nested classes

    - Static member classes
    - Nonstatic member classes
    - Anonymous classes
    - Local classes

- Static member classes could exist without enclosing *instance*
- If you declare a member class that does not require access to an enclosing instance, always put the static modifier in its declaration
- Anonymous classes could be instantiated *only* at the point they are declared
- Anonymous classes have enclosing instances if they are defined ina nonstatic context
- Local classes can be declared anywhere a local variable can be declared and have the same scoping rules

Item 23: Don't use raw types in new code

- Generic types provide compile-time checking for incompatible types
- Not needed manual cast to type when you retrieve element from collections
- Raw types exists only for backward compatibility
- You lose type safety if you use a raw type
- Raw types could be used with `instanceof` operator

Item 24: Eliminate unchecked warnings

- Eliminate every unchecked warning that you can, that means your code is typesafe
- Use `@SuppressWarnings("unchecked")` only if you can prove the code is typesafe
- Always use the `@SuppressWarnings` annotation on the smallest scope possible
- Every time you use an `@SuppressWarnings` annotation, add a comment saying why it's safe to do so
- Every unchecked warning represents the potential for a `ClassCastException` at runtime. Do not ignore them blindly

Item 25: Prefer lists to arrays

- Arrays are covariant (if `Sub` is subtype of `Super`, then `Sub[]` is a subtype of `Super[]`)
- Generics are invariant (`List<Sub>` is not a subtype of `List<Super>`)
- Arrays are reified (enforce their element types at runtime)
- Generics are non-reified and implemented by erasure (enforce types at a compile time, but erased at a runtime)
- Generic array creation errors at compile time (`List<E>[]`)
- Array of non-reified types can not be created

Item 26: Favor generic types

- `Object` type in collections are good candidate to replace with generic types
- `new E[]` cause compile time error, use `(E[]) new Object[]` instead

Item 27: Favor generic methods

- Generic type parameter list, which declares the type parameter, goes between the method's modifiers and its return type (`public static <T> void method()`)
- Generic methods could infere type of arguments

Item 28: Use bounded wildcards to increase API flexibility

- Generics are invariant (`List<Integer>` is not a subtype of `List<Number>`)
- For maximum flexibility, use wildcard types on input parameters that represent producers or consumers
- **PECS**: **P**roducer - **E**xtends , **C**onsumer - **S**uper
- Producer: `add(List<? extends Number>)`
- Consumer: `get(List<? super Number>)`
- `Comparable` and `Comparator` are consumers
- Do not use wildcard types as return types, it would force use wildcards in the client code
- Use explicit types if compiler can't infere them `Union<Number>.union()`
- if a type parameter appears only once in a method declaration, replace it with a wildcard

Item 29: Consider typesafe heterogeneous containers

- Single-element containers could be parametrized (`ThreadLocal`, `AtomicReference`)
- `String.class` is of type `Class<String>`
- Typesafe heterogeneous container pattern

```
public class Favorites {
  public <T> void putFavorite(Class<T> type, T instance);
  public <T> T getFavorite(Class<T> type);
}
```

- `String s = f.getFavorite(String.class)` is typesafe
- You can use `Class` objects as keys for typesafe heterogeneous containers.

Item 30: Use enums instead of int constants

- *int enum pattern* just a class with int constants
- Compiler won't complain if you pass one int constant, where another expected
- If int constant number is changed, clients should be recompiled
- There is no easy way to translate int enum constants into printable strings
- There is no reliable way to obtain size or iterate over all the int enum constants in a group

- *String enum pattern* is even worse
- String comparisons is expensive
- Error is string constant lead to runtime error
- Use enums!
- Each enum internally is `public static final int` field
- Enums provide compile-time type safety
- Enum types with identically named constants coexist peacefully because each type has its own namespace
- You can add or reorder constants in an enum type without recompiling clients
- Translate enums into printable strings by calling their `toString` method
- Enum types let you add arbitrary methods and fields and implement arbitrary interfaces
- If an enum is generally useful, it should be a top-level class; if its use is tied to a specific top-level class, it should be a member class of that top-level class
- To avoid switch on enum constant use *constant-specific method implementations*. Add abstract method to enum type, and override that method for each constant
- Enums have auto-generated methods `valueOf(String)`, `values()`
- Switches on enums are good if you are client user of that enum

Item 31: Use instance fields instead of ordinals

- Every enum constant associated with int value via `ordinal()` method
- Reordering, adding or deleting enum constant cause problems if you depend on `ordinal()`
- Use instance fields for enum (`APPLE(1)` instead of `APPLE.ordinal()`)

Item 32: Use EnumSet instead of bit fields

- *Bit int enum pattern* use int constants as a power of two (1,2,4,8, . . .) This lets you to perform union and intersection with bitwise operations efficiently (`apply(STYLE_ITALIC | STYLE_BOLD)`)
- Hard to interpret bit int constants
- Hard to iterate over bit int constants
- Just because an enumerated type will be used in sets, there is no reason to represent it with bit fields
- Use `EnumSet` instead (`apply(EnumSet.of(Style.ITALIC, Style.BOLD))`)

Item 33: Use EnumMap instead of ordinal indexing

- `ordinal()` for enums cause a lot of problems in array indexing

- ordinal indexing is not typesafe, may cause wrong associations or `IndexOutOfBoundsException`
- Use `EnumMap.get(APPLE)` instead of `array[APPLE.ordinal()]`
- If the relationship that you are representing is multidimensional, use `EnumMap<..., EnumMap<...>>`

Item 34: Emulate extensible enums with interfaces

- There is no much useful use cases to extend enum functionality
- `enum` could inplement interfaces, therefore allow extensibility
- While you cannot write an extensible enum type, you can emulate it by writing an interface to go with a basic enum type that implements the interface

Item 35: Prefer annotations to naming patterns

- Prior to release 1.5, it was common to use naming patterns to indicate that some program elements demanded special treatment by a tool or framework (name test methods beginning with `test` for JUnit)
- No warning about typos, no control over program elements, ugly and fragile approach
- Annotations solve *naming patterns* problems
- Define annotation Test `public @interface Test`
- `@Retention(RetentionPolicy.RUNTIME)` meta-annotation indicates that Test annotations should be retained at runtime
- `@Target(ElementType.METHOD)` meta-annotation indicates that the Test annotation is legal only on method declarations
- Process marker annotations `Method.isAnnotationPresent(Test.class)`
- With the exception of toolsmiths, most programmers will have no need to define annotation types
- Consider using any annotations provided by your IDE or static analysis tools

Item 36: Consistently use the Override annotation

- `@Override` can only be used on method declarations
- `@Override` indicates that the annotated method declaration overrides a declaration in a supertype
- `@Override` helps to catch tricky bugs (overloaded `equals`, `hashcode`)
- Use the `@Override` annotation on every method declaration that you believe to override a superclass declaration

Item 37: Use marker interfaces to define types

- A marker interface is an interface that contains no method declarations (`Serializable`, `Cloneable`)
- Marker interfaces are not marker annotations
- Marker interfaces define a type that is implemented by instances of the marked class; marker annotations do not
- Marker interfaces can be targeted more precisely by extending that interface
- `Set` is *restricted marker interface*. It extends `Collection` but does not add new methods. It only refines contract for some methods to be more targeted.
- The chief advantage of marker annotations over marker interfaces is that it is possible to add more information to an annotation type after it is already in use, by adding one or more annotation type elements with defaults (Java8 default methods)
- If you find yourself writing a marker annotation type whose target is `ElementType.TYPE`, take the time to figure out whether it really should be an annotation type, or whether a marker interface would be more appropriate.

Item 38: Check parameters for validity

- Detect errors and wrong values as soon as possible
- For public methods, use the Javadoc `@throws` tag to document the exception that will be thrown if a restriction on parameter values is violated
- nonpublic methods should generally check their parameters using assertions
- Unlike normal validity checks, they have no effect and essentially no cost unless you enable them, which you do by passing the `-ea` (or `-enableassertions`) flag to the java interpreter
- Do not use validity check if it is impractical or performed implicitly in the process of doing the computation

Item 39: Make defensive copies when needed

- You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants
- If you return mutable reference from class, then class is also mutable
- To make immutable class, make a defensive copy of each mutable parameter
- Defensive copies are made before checking the validity of the parameters and the validity check is performed on the copies rather than on the originals (protect against changes from another thread, TOCTOU *time-of-check/time-of-use* attack)
- Do not use the clone method to make a defensive copy of a parameter whose type is subclassable by untrusted parties
- Defensive copying of parameters is not just for immutable classes, think twice before returning a reference

- Arrays are always mutable
- Defensive copying can have a performance penalty associated with it and isn't always justified
- If the cost of the defensive copy would be prohibitive and the class trusts its clients not to modify the components inappropriately, then the defensive copy may be replaced by documentation outlining the client's responsibility not to modify the affected components

Item 40: Design method signatures carefully

- Choose method names carefully
- Follow the code conventions
- Too many methods make a class difficult to learn, use, document, test, and maintain
- Avoid long parameter lists (four parameters or fewer)
- Use builder pattern, helper classes or helper methods to avoid long parameter lists
- For parameter types, favor interfaces over classes
- Prefer two-element enum types to boolean parameters

Item 41: Use overloading judiciously

- The choice of which overloaded method to invoke is made at **compile time**
- Selection among overloaded methods is static, while selection among overridden methods is dynamic
- A safe, conservative policy is never to export two overloadings with the same number of parameters
- The rules that determine which overloading is selected are extremely complex. They take up thirty-three pages in the language specification [JLS, 15.12.1-3]

Item 42: Use varargs judiciously

- Use `call(int...)` when you need zero or more arguments
- Use `call(int, int...)` when you need one or more arguments
- Don't use varargs for every method that has a final array parameter; use varargs only when a call really operates on a variable-length sequence of values
- Every invocation of a varargs method causes an array allocation and initialization
- Use overloaded methods with 2, 3, 4 params to cover most use-cases, otherwise use varargs

Item 43: Return empty arrays or collections, not nulls

- Do not return `nulls`!
- Return empty collection (`Collections.emptyList()`), or zero-length array (`new int[0]`) instead of nulls
- Zero-length arrays and empty collections are not performance problems, because they are immutable and only one instance could be used

Item 44: Write doc comments for all exposed API elements

- Document API with the `javadoc` utility
- To document your API properly, you must precede every exported class, interface, constructor, method, and field declaration with a doc comment
- If a class is serializable, you should also document its serialized form
- The doc comment for a method should describe succinctly the contract between the method and its client
- With the exception of methods in classes designed for inheritance, the contract should say **what** the method does rather than **how** it does its job.
- Methods should document pre- and postconditions, side effects, thread safety, exceptions
- The first "sentence" of each doc comment should be the summary description
- When documenting a generic type or method, be sure to document all type parameters
- When documenting an enum type, be sure to document the constants
- When documenting an annotation type, be sure to document any members

Item 45: Minimize the scope of local variables

- The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used
- Nearly every local variable declaration should contain an initializer (`try-catch` block is an exception)
- `for` loop allows to declare loop variable, prefer it over `while`
- Keep methods small and focused

Item 46: Prefer for-each loops to traditional for loops

- `foreach` saves from subtle bugs, copy-paste errors, improves readability and maintainability
- `foreach` introduces no performance penalty
- Implement `Iterable` interface to use custom class in `foreach` loop
- `foreach` is not used in

- filtering (no access to iterator to call `remove`)
- transforming (no access to index element to apply change)
- parallel iteration (needed few iterators and control locks)

Item 47: Know and use the libraries

- By using a standard library, you take advantage of the knowledge of the experts who wrote it and the experience of those who used it before you
- If a flaw were to be discovered, it would be fixed in the next release
- With using libraries you don't have to waste your time writing ad hoc solutions to problems that are only marginally related to your work
- Performance of standard libraries tends to improve over time, with no effort on your part
- Libraries tend to gain new functionality over time

Item 48: Avoid float and double if exact answers are required

- The `float` and `double` types are not suited for monetary calculations because it is impossible to represent 0.1 (or any other negative power of ten) as a `float` or `double` exactly
- Use `BigDecimal`, `int`, or `long` for monetary calculations
- `BigDecimal` has full control over rounding
- If performance is crucial, you don't mind keeping track of the decimal point yourself, and the quantities aren't too big, use `int` or `long`

Item 49: Prefer primitive types to boxed primitives

- Primitives have only their values, whereas boxed primitives have identities distinct from their values.
- Boxed primitive may have `null` value
- Primitives more time and space-efficient than boxed primitives
- Applying the == operator to boxed primitives is almost always wrong
- When you mix primitives and boxed primitives in a single operation, the boxed primitive is auto-unboxed,
- Use boxed primitives as type parameters in parameterized types
- Use boxed primitives when making reflective method invocations

Item 50: Avoid strings where other types are more appropriate

- Strings are poor substitutes for other value types (`5` is better than `"5"`)
- Strings are poor substitutes for enum types
- Strings are poor substitutes for aggregate types; to access individual fields you must parse string

Item 51: Beware the performance of string concatenation

- Using the string concatenation operator repeatedly to concatenate n strings requires O(n^2) time
- To achieve acceptable performance, use a `StringBuilder` instead

Item 52: Refer to objects by their interfaces

- If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types
- If you depend on any special properties of an implementation, document these requirements where you declare the variable

Item 53: Prefer interfaces to reflection

- Reflection provides programmatic access to the class's member names, field types, method signatures, etc.
- Using reflection have disadvantages
    - No compile-time type checking
    - Code verbosity
    - Performance suffers
- As a rule, objects should not be accessed reflectively in normal applications at runtime
- Create instances reflectively and access them normally via their interface or superclass
- Using reflection cause compiler warnings

Item 54: Use native methods judiciously

- The Java Native Interface (JNI) allows Java applications to call native methods, which are special methods written in native programming languages such as C or C++
- JNI has three main uses
    - Access to platform-specific facilities such as registries and file locks
    - Access to libraries of legacy code, which could in turn provide access to legacy data

    - Used to write performance-critical parts of applications

- Do not use native methods for improved performance
- Applications using native methods have disadvantages

    - programs are not immune to memory corruption errors
    - less portable
    - difficult to debug

Item 55: Optimize judiciously

- Premature optimization is the root of all evil
- Strive to write good programs rather than fast ones
- Strive to avoid design decisions that limit performance
- Consider the performance consequences of your API design decisions
- Measure performance before and after each attempted optimization

Item 56: Adhere to generally accepted naming conventions

- Typographical

    - Package: `com.google.inject`, `org.joda.time.format`
    - Class/Interface: `Timer`, `FutureTask`, `LinkedHashMap`, `HttpServlet`
    - Method/Field: `remove`, `ensureCapacity`, `getSrc`
    - Constants: `MIN_VALUE`, `NEGATIVE_INFINITY`
    - Local variable: `i`, `href`, `houseNumber`
    - Type parameter: `T`, `E`, `K`, `V`, `T1`, `T2`

- Grammatical

    - Class - *noun*: `Timer`, `Task`
    - Interface - *noun*, *adjective* ends with *able*: `Comparator`, `Comparable`
    - Annotation - *noun*, *verb*, *preposition*, *adjective*: `@Test`, `@Autowired`, `@ImplementedBy`, `@ThreadSafe`
    - Method - *verb*, rarely *noun*: `drawImage`, `getDimension`, `isInterrupted`, `size`
    - Field - *noun*: `height`, `capacity`

- Conventions should not be followed blinfly if long-held conventional usage dictates otherwise

Item 57: Use exceptions only for exceptional conditions

- Exceptions slower than normal checks
- Placing code inside a try-catch block inhibits certain optimizations that modern JVM implementations might otherwise perform
- A well-designed API must not force its clients to use exceptions for ordinary control flow

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

- Java provides three kinds of throwables: checked exceptions, runtime exceptions, and errors
- Checked exceptions *force* the caller to handle them
- Use checked exceptions for conditions from which the caller can reasonably be expected to recover
- Unchecked exceptions indicate programming error and not needed to be handled *almost* always
- Errors indicated JVM problems and not needed to be handled at all

Item 59: Avoid unnecessary use of checked exceptions

- Checked exceptions *force* the caller to handle them in `try-catch` block, or propagate outward
- If catching exception provide no benefit (*recovery*, *logging*) use unchecked exception
- One technique for turning a checked exception into an unchecked exception is to break the method that throws the exception into two methods, additional method returns a boolean that indicates whether the exception would be thrown

Item 60: Favor the use of standard exceptions

- `IllegalArgumentException` caller passes in an argument whose value is inappropriate (e.g. negative value for square root)
- `IllegalStateException` invocation is illegal because of the state of the receiving object (e.g. partially initialized object)
- `NullPointerException` and `IndexOutOfBoundsException` are more specific versions of `IllegalArgumentException`
- `ConcurrentModificationException` object that was designed for use by a single thread or with external synchronization detects that it is being (or has been) concurrently modified.

- `UnsupportedOperationException` used by implementations that fail to implement one or more optional operations defined by an interface

Item 61: Throw exceptions appropriate to the abstraction

- **Exception translation**. Higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction

```
try {
  // Use lower-level abstraction to do our bidding
} catch(LowerLevelException e) {
  throw new HigherLevelException();
}
```

- **Exception chaining**. Higher-level exception could contain reference to lower-level exception (e.g for debugging)
- While exception translation is superior to mindless propagation of exceptions from lower layers, it should not be overused

Item 62: Document all exceptions thrown by each method

- Always document checked exceptions with javadoc `@throws` tag
- Do not document multiple excetions by their common superclass (`@throws Exception` is bad)
- Document *expected* unchecked exceptions with javadoc `@throws` tag
- Do not include unchecked exceptions in method `throws` declaration
- If an exception is thrown by many methods in a class for the same reason, it is acceptable to document the exception in the class's documentation comment

Item 63: Include failure-capture information in detail messages

- To capture the failure, the detail message of an exception should contain the values of all parameters and fields that "contributed to the exception"
- To avoid verbosity, include only *useful* information to exception message
- Exception detail message for programmers, not for users

Item 64: Strive for failure atomicity

- **Failure atomicity** (failed method invocation should leave the object in the state that it was in prior to the invocation)

- If an object is immutable, failure atomicity is free
- If ab object is mutable

    - Check parameters for validity before performing the operation
    - Perform partial computations and then check for validity
    - Write recovery code to return object to its original state after exception
    - Make temporary code, apply changes and then replace original object if no exceptions are thrown

- Failure atomicity is not always desirable (implementation complexity, performance)
- If method is not failure atomic, reflect that in documentation

Item 65: Don't ignore exceptions

- Don't ignore exceptions!
- An empty catch block defeats the purpose of exceptions
- At the very least, the catch block should contain a comment explaining why it is appropriate to ignore the exception

Item 66: Synchronize access to shared mutable data

- Synchronization prevent a thread from observing an object in an inconsistent state
- Synchronization ensures that each thread entering a synchronized method or block sees the effects of all previous modifications that were guarded by the same lock
- Reading or writing a variable is atomic unless the variable is of type `long` or `double`
- Do not use `Thread.stop`
- A recommended way to stop one thread from another is to have the first thread poll a `boolean` field that is initially `false` but can be set to `true` by the second thread to indicate that the first thread is to stop itself
- **Liveness failure** - the program fails to make progress
- Synchronization has no effect unless both read and write operations are synchronized
- Increment operator (++) is not atomic
- **Safety failure** - the program computes the wrong results
- Strive to assign mutable data to a single thread

Item 67: Avoid excessive synchronization

- To avoid liveness and safety failures, never give control to the client within a synchronized method or block
- *Alien* methods may cause data corruption, deadlocks

- Java locks are reentrant
- `CopyOnWriteArrayList` is a variant of `ArrayList` in which all write operations are implemented by making a fresh copy of the entire underlying array
- Do as little work as possible inside synchronized regions
- When in doubt, do not synchronize your class, but document that it is not thread-safe

Item 68: Prefer executors and tasks to threads

- Executor framework separates unit of work (task) and mechanism (thread creation)
- `Thread` is no longer a key abstraction, use `Runnable` or `Callable`
- `Executors.newCachedThreadPool` good choice for lightly-loaded server, if no threads available for submitted task, new one will be created
- `Executors.newFixedThreadPool(n)` good choice for heavily-loaded server, Only `n` threads will be created

Item 69: Prefer concurrency utilities to wait and notify

- Prefer higher-level concurrency utilities (Executor Framework, concurrent collections and synchronizers) to `wait` and `notify`
- `ConcurrentHashMap` is optimized for retrieval operations
- Use `ConcurrentHashMap` in preference to `Collections.synchronizedMap` or `Hashtable`
- `BlockingQueue` used for producer-consumer queues
- `CountdownLatch` is a single-use barrier that allow one or more threads to wait for one or more other threads to do something
- For interval timing, always use `System.nanoTime` in preference to `System.currentTimeMillis`. `System.nanoTime` is both more accurate and more precise, and it is not affected by adjustments to the system's real-time clock.
- `CyclicBarrier` could be used if you need multiple `CountDownLatch` objects
- Always use the *wait loop* idiom to invoke the `wait` method; never invoke it outside of a loop

Item 70: Document thread safety

- The presence of the synchronized modifier in a method declaration is an implementation detail, not a part of its exported API
- To enable safe concurrent use, a class must clearly document what level of thread safety it supports

- **immutable** - Instances of this class appear constant. No external synchronization is necessary
- **unconditionally thread-safe** - Instances of this class are mutable, but the class has sufficient internal synchronization that its instances can be used concurrently without the need for any external synchronization
- **conditionally thread-safe** - Like unconditionally thread-safe, except that some methods require external synchronization for safe concurrent use. Examples include the collections returned by the Collections.synchronized wrappers, whose iterators require external synchronization
- **not thread-safe** - Instances of this class are mutable. To use them concurrently, clients must surround each method invocation (or invocation sequence) with external synchronization of the clients' choosing
- **thread-hostile** - This class is not safe for concurrent use even if all method invocations are surrounded by external synchronization. Thread hostility usually results from modifying static data without synchronization. No one writes a thread-hostile class on purpose; such classes result from the failure to consider concurrency

- To prevent DOS attack, you can use a private lock object instead of using synchronized methods

Item 71: Use lazy initialization judiciously

- Best advice for lazy initialization is "don't do it unless you need to"
- If you use lazy initialization to break an initialization circularity, use a synchronized accessor
- If you need to use lazy initialization for performance on a static field, use the *lazy initialization holder class* idiom
- If you need to use lazy initialization for performance on an instance field, use the *double-check* idiom

Item 72: Don't depend on the thread scheduler

- Any program that relies on the thread scheduler for correctness or performance is likely to be nonportable.
- The best way to write a robust, responsive, portable program is to ensure that the average number of *runnable* threads is not significantly greater than the number of processors
- Do not rely on `Thread.yield` because it has no testable semantics
- Use `Thread.sleep(1)` instead of `Thread.yield()` for concurrency testing
- Thread priorities are among the least portable features of the Java platform

Item 73: Avoid thread groups

- Thread groups do not provide any security functionality
- Thread API is weak
- Prior to release 1.5, there was one small piece of functionality that was available only with the ThreadGroup API the `ThreadGroup.uncaughtException` method was the only way to gain control when a thread threw an uncaught exception. As of release 1.5, however, the same functionality is available with `Thread.setUncaughtExceptionHandler` method.

Item 74: Implement Serializable judiciously

- Implementing `Serializable` decreases the flexibility to change a class's implementation once it has been released
    - `private` and `package-private` fields become part of its exported API
    - incompatible changes after deserialization lead to failure
- Define `serialVersionUID` to avoid `InvalidClassException`
- Because there is no explicit constructor associated with deserialization, it is easy to forget that you must ensure that it guarantees all of the invariants established by the constructors
- Releasing new version of serialized class greatly improves number of test-cases need to be verified
- Classes designed for inheritance should rarely implement `Serializable`
- You should consider providing a parameterless constructor on nonserializable classes designed for inheritance
- Use *thread-safe state machine* pattern (atomic-reference to enum) to implement a serializable superclass
- Inner classes should not implement `Serializable`

Item 75: Consider using a custom serialized form

- Do not accept the default serialized form without first considering whether it is appropriate
- The default serialized form is likely to be appropriate if an object's physical representation is identical to its logical content
- Even if you decide that the default serialized form is appropriate, you often must provide a readObject method to ensure invariants and security

- Before deciding to make a field nontransient, convince yourself that its value is part of the logical state of the object
- Declare an explicit serial version UID in every serializable class you write

Item 76: Write readObject methods defensively

- For classes with object reference fields that must remain private, defensively copy each object in such a field. Mutable components of immutable classes fall into this category.
- Check any invariants and throw an `InvalidObjectException` if a check fails. The checks should follow any defensive copying.
- If an entire object graph must be validated after it is deserialized, use the `ObjectInputValidation` interface
- Do not invoke any overridable methods in the class, directly or indirectly.

Item 77: For instance control, prefer enum types to readResolve

- To satisfy singleton property for serializable object, implement `readResolve`
- If you depend on `readResolve` for instance control, all instance fields with object reference types must be declared `transient`
- `readResolve` is not obsolete. It is needed for writing a serializable instance-controlled class whose instances are not known at compile time
- Use enum types to enforce instance control invariants wherever possible

Item 78: Consider serialization proxies instead of serialized instances

- Serialization proxy is a private static inner class implements Serializable and reflects serializable data for original object
- Add `writeReplace` method to proxy class. Serialization system emits a proxy instance instead of an instance of the enclosing class.
- Add `readObject` method to proxy class. Attacker wouldn't be able to violate class invariants.
- Add `readResolve` method to proxy class that returns logically equialent instance of the enclosing class.
- The serialization proxy pattern has some limitations:

    - It is not compatible with classes that are extendable by their clients
    - It is not compatible with some classes whose object graphs contain circularities
    - Serialization is slower than standard approach