Final Last Updated August 9, 2013

Effective Java Reading Notes

Ilya Nepomnyashchiy

I received this book as part of my "welcome package" at work and my mentor regularly references items from it in code reviews, so I felt like it was a good idea to read through this book. I also thought it would make a good starting point for my reading notes. Apparently even James Gosling, the creator of Java, thinks this book is quite legit. I don't know how much better of an endorsement you could have.

These are notes for myself. If you have any questions, the book itself is much more detailed and is a must-have for any Java developer anyway. That being said, I'd be happy to clarify, debate, or correct any observations that I make here if you e-mail me from my website.

Contents

Chapter 2: Creating and Destroying Objects

Item 1: Consider static factory methods instead of constructors

When creating a new class, one should consider using a static factory method (not to be confused with the common Factory Method pattern) instead of a constructor. This is a function, along the lines of `Boolean.valueOf`, that returns an instance of the class. There are several advantages to these methods: they have names (`Number.getPrimeNumber(3, 5.4)` conveys much more information than `new Number(3, 5.4)` and you can have several with the same signature), they aren't required to create a new instance of the object (good for classes where you can cache instances or singletons), they can return subtypes of the class they're in, and they allow for type inference when parameterized types are involved (you can't write `MyClass<List<X>, Y> thing = new MyClass()`, which is pretty silly, in my opinion).

Their two disadvantages are that it's hard to distinguish them from other static methods, and that you cannot subclass these methods. The latter is because subclasses must be able to call a constructor of their super class.

Item 2: Consider the builder pattern

Try to avoid telescoping contructors, like

```
public MyClass(int a) {
    MyClass(a, 0, 0);
}

public MyClass(int a, int b) {
    MyClass(a, b, 0);
}

public MyClass(int a, int b, int c) {
    // Do some stuff
}
```

, which are bad because they cause unreadable code (try debugging what `new MyClass(1, 2, 3, 4, 5, 6, 7)` is supposed to mean). Also try to avoid using the JavaBeans Pattern, which has the downside of letting your class potentially remain uninitialized and doesn't allow you to have an immutable class. Instead, consider forming a Builder class. This sort of pattern has the safety of the telescoping constructor and the readability of the JavaBeans pattern: `new MyClass.Builder(1, 2).donuts(3). retries(4).secondsToRetry(5).build()`. It almost reads like some sort of functional language :)

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

If your class should only have one instance, either make the constructor private and have a public static instance in the class, use a public static factory method to return a private static instance, or make the class into an enum. The very last option has the upside of providing a lot of machinery that is necessary to ensure singletons remain singletons when serializing objects.

Item 4: Enforce noninstantiability with a private constructor

The main idea here is that if you don't have an explicitly defined constructor, Java will automatically create a public constructor which does nothing. This is bad because you can suddenly instantiate classes that are bags of utilities or something like that. Therefore, make a private constructor that does nothing or throws an exception and never call it.

Item 5: Avoid creating unnecessary objects

In general, this is pretty clear. If your class creates the same objects, which are treated like constants, every time some method is invoked, consider making a private static final member and then using a `static` { } block to initialize them.

Another pitfall to avoid is accidentally unboxing and reboxing primitive types. The following code creates an unnecessary object:

```
Long a = 5L;
long b = 5L;
a = a + b;
```

Item 6: Eliminate obsolete object references

When you have a class that manages its own memory (e.g. an implementation of a stack), you have to null out references or the Garbage Collector will believe that those references are still in use and won't collect said objects. However, don't become paranoid and null out references that you don't need to null out, because that's just ugly.

Who knew that a garbage collected language could have memory leaks?

Item 7: Avoid finalizers

Pretty much never use finalizers. They greatly slow down your code and are not guaranteed to ever run. The only time you ever want to use them is as a safety net (although a terminator method is better) or when terminating primitive objects.

Finalizers are apparently a sort-of analogue to destructors in C++ except that they're often completely unnecessary (because garbage collectors are magical) and can make your program perform very poorly.

Chapter 3: Methods Common to All Objects

This chapter mostly concerns itself with those common methods we're all used to, like `toString` and `equals`

Item 8: Obey the general contract when overriding equals

The general idea here is that `equals` should behave mathematically like the equals operator, with the additional constraints that no object should be equal to null, and objects that are equal should remain equal unless something is changed. The latter of these constraints is easily violated if your `equals` method depends on an unreliable resource, which is generally a bad idea. It is also very easy to violate symmetry or transitivity if you try to make `equals` work on subclasses or something like that. In cases where you have subclasses, it is better to use composition rather than inheritance if you want the equality to work out. This chapter also provides a nice recipe for good `equals` functions.

Item 9: Always override hashCode when you override equals

The basic idea is that many library structures require that objects have the same `hashCode` when they are equal (the converse need not be true, but should generally be true for performance reasons). This chapter provides a recipe for a `hashCode` function that works well in practice, although it may not be cryptographically strong or anything.

Item 10: Always override toString

People will often want to convert your object into a string representation, and the default method is pretty useless in most cases, so you should make a useful version. However, information that's displayed in the string representation should be accessible by other means, or else developers will be forced to parse strings to use your object, which results in fragile systems and unhappy clients.

Item 11: Override clone judiciously

The tl;dr here is that `clone` and `Clonable` are terrible and you should avoid using them whenever possible.

Item 12: Consider implementing Comparable

If your object admits some sort of natural ordering, you should implement `Comparable` because it allows you to use lots of library functions that will be very useful, such as `Arrays.sort`. Make sure that your `compareTo` function functions like a mathematical comparison operator. It is not required that `compareTo` agrees with `equals` when it comes to equality, but it is highly recommended.

Chapter 4: Classes and Interfaces

This is the longest chapter so far! Most of these items are effectively "use classes and interfaces in the way they were designed so that the compiler can check for things rather than your code giving you lots of runtime errors."

Item 13: Minimize the accessibility of classes and members

This item advises following the idea of encapsulation by closing off (making private, package-private or protected) as many implementation details as possible. In general, you should use the lowest access level that you can for your code to still work. This will often give you nice things like thread-safety, preservation of invariants, etc. One thing to note is that having a final reference to a mutable object is as bad as having a mutable field and that arrays can never be immutable. This can cause potential security holes.

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

For private classes this matters little, but in public classes, anything that is exposed becomes a permanent part of the class' public API and can never be changed for fear of breaking compatibility. Thus, you should provide accessor methods and make the fields private so that you can change them at any point in time. One potential exception is if the fields are immutable, but this is described as "questionable."

Item 15: Minimize mutability

Make as few parts of your class mutable as possible. Unless otherwise necessary, do not provide mutators, prevent subclassing by declaring the class final (so that subclasses can't compromise immutability), make all fields final, make all fields private, and make sure no mutable components can be accessed by others. This, like item 13, gives you nice things like thread-safety and the ability to share your objects without needing to reason about their safety. One can also cache certain instances of the class and/or certain fields that one does not want to immediately calculate.

The biggest disadvantage of immutable classes is that you have to create a new instance to make a small change. This can be fixed by making multistep operations available as a primitive operation that only creates one new instance. It can also be fixed by making a mutable utility class (such as `StringBuilder` for `String`, or to some extent `BitSet` for `BigInteger`).

Classes that cannot be fully immutable should limit mutability as much as possible and have no reinitialization methods.

Item 16: Favor composition over inheritance

Instead of subclassing (which violates encapsulation, because the subclass needs to be aware of the internals of the superclass and exactly how all overridden methods are used), consider adding the class you want to inherit from as a private field. The book provides an example of where subclassing fails really hard. This generally only applies to subclassing across packages, since changing implementation details of a superclass within the same package means that changes to the subclass are local and can be done quickly (and you're likely breaking your own code rather than someone else's).

As the book describes, subclassing should only be used for "is-a" relationships. A Stack is not a Vector, but a Circle is a Figure.

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

In general, due to the previous item, one should prevent one's classes from being subclassed, to avoid having to do the following: If you wish your class to be subclassed, you must document exactly when and how any overridable method is used in the documentation. The book's example in item 16 illustrates why this is the case. Since one would typically wish to avoid having to expose internals in documentation, it is generally best to avoid encouraging subclassing of your classes. This is especially true because anything you put in your documentation becomes permanently part of your class' API for other developers subclassing.

Furthermore, when developing a class for subclassing, one should test that the interface is both as big and as small as it should be by writing several subclasses.

Item 18: Prefer interfaces to abstract classes

Java does not support multiple inheritance and it is easier to retrofit older classes to support a new interface, so interfaces are generally a better option. If one wishes to implement some of the functions, one can create an `Abstract*` skelletal class and then use anonymous classes to extend these, or have an instance of the skelletal class as a private field and forward requests to that field.

On the other hand, interfaces are harder to update with new methods / fields.

Item 19: Use interfaces only to define types

Interfaces that contain constants, for example, should be avoided (instead create a class that includes these constants and use a static import). This makes sense because an interface describes a type that can be accepted as a parameter, for example, and it makes no sense to accept as a parameter any class that "uses these physical constants." It also makes an internal implementation detail part of your class' public signature, which means you cannot switch away from using this interface and preserve binary compatibility.

Item 20: Prefer class heirarchies to tagged classes

This one, I think, should be straightforward to anyone experienced with object-oriented programming. Instead of using a class like:

```
class Figure {
    enum Shape { RECTANGLE, CIRCLE };
    \\ ...
}
```

and having tons of methods and fields that are each only applicable to one of the types, create an abstract class that implements all methods and fields that are common, and then use subclassing. Classes and subclassing are cheap! Tagged classes are just poor reimplementations of subclassing that introduce lots of boilerplate code and does not easily allow for immutable classes.

Item 21: Use function objects to represent strategies

In C++, when one wants to, say, pass a comparator to a sorting algorithm, one sends a function pointer:

```
int compare_function(const void *a, const void *b) {
    return ( a - b );
}

qsort(array, num, size, compare_function);
```

This is known as the strategy pattern.

In Java, the idiomatic way to do this is by creating an interface for the type of thing you want to pass (in our example a comparator, so we might define `Comparator<T>`) and then creating a class for each strategy. In many cases you will be able to use an anonymous class extending the interface (assuming you don't have to do it many times: a new instance is created each time). If you do need to pass this class in many times, you should use a private static final field.

This should be fairly straightforward, as object references in Java are the closest things to pointers.

Item 22: Favor static member classes over nonstatic

The basic idea here is that nonstatic member classes must be attached to an instance of your class and have a reference to said instance. Even if the member class is used in an instance of your class, if it doesn't need to reference said instance, it is best to leave it static. For example, the `Entry` subclass of the `Map` class is static. This saves an unnecessary reference from existing.

Chapter 5: Generics

Hooray for generics! I honestly can't imagine a language like Java without some sort of generic type system. Once again, most of these items are just about using generics the way they're designed and ensuring that errors are spotted at compile time rather than runtime. However, some of these things were new to me.

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

An example of a raw type is `List` rather than `List<String>`. The reason you shouldn't use them is that, while `List<E>` is not a subclass of `List<F>` for any `E` or `F` (except, of course, when they're equal), we have that `List<E>` is a subclass of `List`, ∀ `E`. This leads to awful things like:

```
void addItem(List x, Object o) {
    x.add(o);
}

public static void main(String[] args) {
    List<String> strings = new ArrayList<String>();
    strings.addItem(strings, new Integer(5));
    String s = strings.get(0);
}
```

The above code compiles and then throws a runtime exception. The only reason raw types exist in the first place is for backwards compatibility of old code.

There are two exceptions: When getting class literals (e.g. `List.class` is legal, but `List<String>.class` is not), and when using `instanceOf`. Both of these stem from the fact that the type parameter is erased at runtime (type checking is just performed at compile time). One should make sure to use the unbounded wildcard type in the latter case when doing something with said items (the unbounded wildcard parameter, as in `List<?>`, tells us that what we have is a `List` of some type but we don't care which – most notably it means you cannot add things into said object).

Item 24: Eliminate unchecked warnings

When you try to cast something to a parameterized type, the compiler will try to type check and fail. It is worth thinking long and hard about whether you need to cast in such a way. If you do, and you can *prove* that said code is type safe, then you can use `@suppressWarnings("unchecked")` before the relevant expression. It is important that this is used in as small a scope as possible and that you add a comment explaining why this expression is definitely type safe.

Item 25: Prefer lists to arrays

Arrays are worse because: `E[]` is in fact a subclass of `F[]` if `E` is a subclass of `F`. This relation is called covariance (I assume because of some sort of category theoretical connection to type theory?) which would allow you to do things like cast an `E[]` array to an `Object[]` array and then put things into the array that don't belong there. On the other hand, parameterized lists are type checked at compile time.

However, due to the fact that arrays keep their type information at runtime, but parameterized types are "erased", it is generally best to not mix the two. It is also worth noting that the compiler will not let you create a generically typed array for this reason.

Item 26: Favor generic types

It's sort of sad that this needed to even be said, but generic types are great, if only because they provide for better type checking than, say code that operates on `Objects`. The general recipe for generifying classes is to replace all instances of `Object` with a formal type parameter and then resolve all warnings.

Item 27: Favor generic methods

Same idea as above. This section also describes recursive type bounds such as `public static <T extends Comparable<T>> T max(list<T> list)` which takes any class that can be compared to itself.

Item 28: Use bounded wildcards to increase API flexibility

For example, a union method that returns a `Set<E>` should take two parameters of `Set<? extends E>`, which is a set of a definite type that is a subtype of `E`. This allows you to take the union of a `Set<Number>` and a `Set<Integer>`, for instance.

The other bounded wildcard is `Iterator<? super E>`, which takes any type that is a superclass of `E`. The mnemonic given in this section is "PECS", or Producer-extends Consumer-super. The idea is that if the parameter will give you `E`s then you use the `extends` keyword, and if the parameter takes the `E`s then it should take any superclass of the `E`s. You should take some time to think about this if it's not intuitive, because I feel that you don't really need the mnemonic if you have the intuition on why this is true. Here's a great Stack Overflow answer illustrating this.

Item 29: Consider typesafe heterogeneous containers

By noting that `Class` is actually a parameterized type, one can make container classes and methods that take a `Class<T>` as a parameter and thus ensure type safety for keeping objects of varying classes.

Chapter 6: Enums and Annotations

Item 30: Use enums instead of int constants

Before Java had enums, Java programmers had to manually declare enumerated types as ints. This is silly because these types did not have their own namespace and also because they behaved as ints everywhere, allowing for some terrible code with many downsides (described in the section). Enums fix this and allow for multiple nice things: You can associate data and behaviors with the various constants, as enums are fully-fledged classes and each enum constant is an instance of this class.

When associating a method with each enum constant, you may be tempted to use a switch statement inside the method to decide which code path it follows based on which enum constant it is in. It is better to declare an abstract method in the class and then override it within each constant, or use a nested strategy enum and then have a method in the main enum that calls it. This is because you may forget to add a case to the switch statement when adding a new constant, and part of the upside of enums is that they can be easily evolved.

Item 31: Use instance fields instead of ordinals

It may be tempting to use the `ordinal()` method, which returns the numerical position of a constant in an enum. This is a bad idea for two reasons: the constants can be re-ordered, and you may have intervening values that are not there or want to have two enum constants share the same value at some point. Instead associate a private int with each constant.

Item 32: Use EnumSet instead of bit fields

It may also be tempting to use a bit field (much like in C) when you want to pass a set of enum constants to a function, but this has all the same downsides of using int constants. It is best to use `EnumSet`, which is implemented using a bit field as well, but is abstracted away all nicely.

Item 33: Use EnumMap instead of ordinal indexing

Once again, one shouldn't use the `ordinal()` method for pretty much anything (unless doing some sort of very internal work). When one wants to index some data by elements of an enum type, one should use an `EnumMap`, which is doing the same thing underneath, but will catch more errors at compile time and abstracts the implementation away. Thus, there is no danger of doing unchecked casts or messing up the int index. For multidimensional associations, one can use a nested `EnumMap`.

Item 34: Emulate extensible enums with interfaces

Unfortunately, one cannot subclass an enum. However, this can be emulated by creating an interface with the methods that are desired in the enum and extending it. Then, all APIs should accept elements that are of the type of the interface.

Item 35: Prefer annotations to naming patterns

Before annotations, methods and classes that were to be denoted as special (for tools or IDEs) needed to be named in a certain way (for example, tests would be named with a name that started with `test`). For obvious reasons, this is very fragile (what if you make a typo?) Instead, newer versions of Java allow developers to use annotations (those tags before methods and classes that start with `@`). Annotations can also have an associated value or array of values.

Item 36: Consistently use the Override annotation

To oversimplify what the section writes a little bit, always use the `@Override` annotation if your method is intended to override something in a superclass or interface. This is intended to catch typos or accidentally messing up the method signature.

Item 37: Use marker interfaces to define types

When one wants to mark some sort of class or method as special, one can either use an annotation or make an interface with no methods (to simply define a type). In general, interfaces are for when one wants to have methods that only take things marked with said interface.

Chapter 7: Methods

Item 38: Check parameters for validity

Unless it is costly to do so, or implicitly checked in the operation of the method, the method should always document the proper form of arguments passed in and check them before doing anything. This will ensure that malformed arguments raise an exception early in the function's execution rather than mysteriously in the middle of a computation. This is especially true in constructors and other methods that save data for later execution based on that data.

To illustrate the latter clase of the first sentence above, a sorting class will be comparing elements to each other, so there is no need to check that all elements are mutually comprable. Furthermore, the exception raised here will be correct. In the case that the exception is not correct, there is an idiom (described later) that allows one to change the exception raised.

In private and helper classes that are never accessed by outside packages, it is sufficient to use asserts rather than exceptions, because the package developer controls all data passed in.

Another interesting thing noted in this section is the `@throws` tag. This is similar to `@return` but describes when exceptions are thrown.

Item 39: Make defensive copies when needed

When your class has mutable member elements that are passed in or returned, always make a copy when saving or returning. This prevents outside (malicious or incompetent) clients from breaking invariants on your classes. In doing this, avoid using the `clone` method, as clients could pass in malicious subclasses of the parameter you want, and `clone` would create one of those (which potentially give your attacker a reference to the object).

I think what this item illustrates is that it's helpful to remember that Objects are all references in Java, which means that any code with Objects is basically just carrying around a bunch of pointers (without most of the memory management issues).

Item 40: Design method signatures carefully

This section has a lot of miscellaneous method design suggestions: choose method names carefully, don't go overboard in providing convenience methods, avoid long parameter lists (I know of companies that actually ban methods where two booleans show up next to each other in the parameter list), favor interfaces over classes that implement them for parameter lists (for maximum extensibility), and prefer two-element enum types to boolean parameters (for readability).

Item 41: Use overloading judiciously

Overloaded methods are dispatched at compile time, not runtime (like overridden classes), which can lead to confusing and disasterous effects. In general (with possible exceptions for constructors), one should avoid having overloaded methods with the same number of parameters unless those parameters can clearly never be casted to each other (be careful of auto-unboxing and generics when determining this).

Apparently the rules for picking which overloaded function to use take thirty-three pages of the language specification. Whee.

Item 42: Use varargs judiciously

The gist here is that varargs become an array, but sometimes in ways you don't expect. It seems hard to say what the difference between “wanting an array input” and “wanting a varargs input” is. My personal inclination (not the book's) would be to stick with array inputs in general.

Item 43: Return empty arrays or collections, not nulls

Returning a `null` value when you wish to return an empty array requires your client to have special code for handling those values. In order to get around having to allocate a new array each time, you can save an instance of an empty array or list and return something that looks like this:

```
private static final Thing[] EMPTY_ARRAY = new Thing[0];
private final List<Thing> ThingList;
...
return ThingList.toArray(EMPTY_ARRAY);
```

which gives the `toArray` call the type you wish to return, and if the list is empty just returns the empty array (go read the doc to see why this is the case).

For returning a list, one can use `Collections.emptyList()` or the analogues for sets and maps.

Item 44: Write doc comments for all exposed API elements

This should be obvious since you have probably used APIs with these docs and had an IDE like Eclipse give you nice descriptions, or seen a generated JavaDoc. Basically, if the API is exposed, include a doc comment with things like `@param`, `@throws` and `@return`. Make sure to follow established conventions for these comments.

Chapter 8: General Programming

Item 45: Minimize the scope of local variables

In order to avoid confusion and mistakes when variables are accidentally misused or cannot be located, declare variables right before you need them, initialize them immediately (unless you can't), and prefer for-like loops to while-like loops.

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

For-each loops are much easier to read and avoid several common mistakes that anyone who has programmed in a language like Java or C++ has made 1000 times before. There are a few cases where one cannot use for-each loops, though: filtering, transforming, and parallel iteration of several arrays.

Item 47: Know and use the libraries

Using library functions means using better maintained and quicker updated code because of how many people maintain the Java libraries. Plus, you can avoid incorrectly implementing something that you don't have enough domain knowledge to implement.

The author recommends that every programmer be familiar with `java.lang`, `java.util`, and `java.io`.

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

As any programmer ought to know, floats and doubles are really bad at keeping certain exact values. One such example is monetary amounts. A better solution is to use ints or longs and keep track of the decimal point, or use `BigDecimal`.

One of my friends disregarded this when writing something that kept track of money, and now my balance is something like $1.689999999.

Item 49: Prefer primitive types to boxed primitives

For various performance and correctness reasons, one should prefer primitive types like int, float, and double to their boxed types Integer, Float, and Double, unless it is an application where primitives are unacceptable (such as in type parameters).

Item 50: Avoid strings where other types are more appropriate

There are lots of library functions that operate on strings and they're a first-class part of the language, so it's tempting to use them for things that they really aren't suited for. For example, you may want to use them for enum types, aggregate types, or capabilities. However, the problem is that you lose a lot of semantic checking that you could get if you wrote or used classes better suited for these operations, and you also suffer in performance when you have to spend your time parsing strings.

Item 51: Beware the performance of string concatenation

Strings are immutable, so every time you use the `+` operator on a String, you create a copy of the two strings. This results in an $O(n^2)$ operation for concatenating many strings. Instead, use `StringBuilder` if you have to perform a large number of string concatenations.

Item 52: Refer to objects by their interfaces

When declaring a new object, you might typically write something like:

```
ArrayList<Blah> list = new ArrayList<blah>();
```

However, it is a better idea to declare it as:

```
List<Blah> list = new ArrayList<blah>();
```

The reason for this is that if you later decide to use `Vector` or another datatype implementing the `List` interface, it only takes changing this one line to potentially increase the performance of your program, or otherwise better it (whatever the reason is that you're changing the type).

In some cases, you cannot do this: if there is no interface or if you require methods available in a specific type that are not present in the interface contract. In general, the suggestion is to use the most general type when referring to an object.

Item 53: Prefer interfaces to reflection

Reflection allows programmers to programmatically obtain classes, constructors, and fields. As you might expect, this leads to a slew of problems (lots of code to actually do, no compile time checking, etc.) If reflection is absolutely required (and it usually isn't), refer to the object by an interface it implements, so that you can pretend for the rest of the code that there is no reflection present.

Item 54: Use native methods judiciously

Native methods (e.g. calling out to a C library) are used for three reasons: access to legacy code, platform-specific features, and performance. It is appropriate to use native methods for the first use always, sometimes for the second use (although in many cases there are Java libraries that can perform the same operation), and rarely for the third use. Native methods are not as safe as Java code (for example, they are not memory managed), and small mistakes can corrupt the entire application. Furthermore, there is a fixed overhead to calling these libraries. Finally, apparently many features run just as fast in Java as an equivalent implementation in C would. I suppose it depends on the feature and application.

Item 55: Optimize Judiciously

The basic premise here is to never let optimization and performance harm the design of your program, because a poor design just means that the program will be terrible to maintain and further optimize. Also, do not optimize prematurely: focus on writing a good program before thinking about performance, unless it is part of a persistent data format or API.

Item 56: Adhere to generally accepted naming conventions

This item describes several naming conventions for classes, methods, fields, and local variables.

Chapter 9: Exceptions

You should thank me for refraining from making any bad puns here.

Item 57: Use exceptions only for exceptional conditions

The following is awful code and you should never do it:

```
try {
    int i = 0;
    while(true)
        array[i++].thing();
} catch(ArrayIndexOutOfBoundsException e) {
}
```

It's bad because it makes the code much less readable (fewer developers will be able to just look at the code and tell what it does) and it could accidentally ignore an exception legitimately thrown. It may seem like doing the code this way may provide performance benefits (because the loop end condition is not checked on every iteration of the loop), but the JVM optimizes that away in normal loops anyway. In fact, by executing the code in a `try` block, which is treated specially, this version winds up performing worse.

Furthermore, APIs should not force developers to use exceptions. If an object can only be used in a certain state, there should instead be state-testing methods like `Iterator.hasNext()`. Alternatively, return a distinguished value (this is only preferred if thread-safety is required, as an object can change in between the state-testing method call and the actual call, and if there is a distinguished value).

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

Checked exceptions should be used to signal conditions that a client can recover from, whereas unchecked exceptions and errors are ones that cannot be recovered from (and thus rightfully end the program). Most user-defined exceptions should not be errors. The choice for whether an exception should be checked or unchecked is sometimes more of an art than a science and should be handled on a case-by-case basis, considering how the API will be used.

(Not from the book): Some people think that developers should avoid checked exceptions altogether. Without having given it enough thought, I'm inclined to agree with this. Checked exceptions are basically like goto statements in my mind. I'm sure there are some exceptions (hah) to this.

Item 59: Avoid unnecessary use of checked exceptions

If a client can never recover from the error, the exception should just be unchecked. Making it checked is just cluttering the code.

Item 60: Favor the use of standard exceptions

The standard Java libraries include many standard exceptions and one should try to use those whenever the semantics match, as developers will be much more familiar with these exceptions, so it will increase code readability.

Item 61: Throw exceptions appropriate to the abstraction

Sometimes lower level code will throw an exception, which will propagate to higher level code and then make no sense to clients (for example, one could imagine an `IndexOutOfBoundsException` being thrown by a class with an array backing it but that exposes no array-like semantics). Use the techniques of exception translation and exception chaining to return something that fits your API.

Item 62: Document all exceptions thrown by each method

Make your `throws` clause as specific as possible (*never* use `throws Exception` or `throws Throwable`), only put checked exceptions in that clause, and document all unchecked exceptions with `@throws` doc comments. Unchecked exceptions generally correspond to preconditions of your method, so documenting any unchecked exception you are aware of also doubles as documenting the preconditions for your method.

Item 63: Include failure-capture information in detail messages

Try to include as much information about the failure inside the exception object, and especially its string representation, so that developers will be able to more effectively debug the issue.

Item 64: Strive for failure atomicity

Always make sure the object that was trying to be modified when a checked exception is thrown is still valid after the exception. This can be done by checking the exceptional condition before modifying anything, write recovery code that can rollback the object, or perform any mutations on a temporary copy of the object. Or just use immutable objects.

Item 65: Don't ignore exceptions

Ignoring an exception is like ignoring a fire alarm and you are a terrible person for doing so. Unless you live in a dorm where the fire alarm is way too sensitive. Then you need to realize that you are programming the equivalent of a college dorm, and should probably fix the exceptions.

Chapter 10: Concurrency

Item 66: Synchronize access to shared mutable data

The `synchronized` keyword provides two, both very important, things: mutual exclusion and communication between threads. Without it, threads may not see updates to variables that they are watching. You may also use the `volatile` keyword if you do not need mutual exclusion. The best thing to do, though, is to avoid having shared mutable data.

Item 67: Avoid excessive synchronization

Using excessive synchronization can hurt performance, cause incorrect code, cause runtime exceptions, or cause deadlocks. Never run code provided by a client inside a synchronized block. Do as little work inside a synchronized block as possible. Prefer not to synchronize an object internally unless you believe there are gains to be made by doing so.

Item 68: Prefer executors and tasks to threads

The Java libraries provide several executor objects, which take a `Runnable` and execute them asynchronously. Thus, instead of using threads, which are conceptually muddy to deal with, use tasks and executors.

Item 69: Prefer concurrency utilities to wait and notify

`wait` and `notify` were used in threads prior to Java 1.5 (when Java wasn't a real language) to implement various concurrency idioms. However, in modern Java, many of these are implemented for you and you should use them instead. For example, `CountDownLatch`, `Semaphore`, and `CyclicBarrier`.

For interval timing, always use `System.nanoTime` in preference to `System.currentTimeMillis`. It is more accurate and more precise.

Always invoke the `wait` method inside a while loop that checks the condition the thread is waiting on. Prefer `notifyAll` to `notify`. Only use these two in legacy code and prefer the utilities in `Java.util.concurrent`.

Item 70: Document thread safety

Document how much internal synchronization is performed in an object in its document comment (whether it is immutable, is fully thread-safe, conditionally thread-safe, not thread-safe or actively thread-hostile). Consider using a private lock if the class is unconditionally thread-safe, to protect from synchronization interference.

Item 71: Use lazy initialization judiciously

As with all optimizations, lazy initialization may sometimes harm performance. Only use it if you need to. It is even harder when you attempt to do it with classes intended to be used concurrently. The double-check idiom can be used for instance field lazy initialization and the lazy holder idiom can be used for static field lazy initialization to make them safe.

Item 72: Don't depend on the thread scheduler

Do not depend on thread priorities or `Thread.yield` as this can yield to incorrectness. Never busy wait.

Item 73: Avoid thread groups

Thread groups were a feature added for a particular purpose that the lanugage designers decided they didn't need them for. Now thread groups serve no function, and many of their associated methods are unsafe.

Chapter 11: Serialization

Except in some particular cases, I feel like it'd be a better idea to use something like a Protocol Buffer, but when using Java Serialization, it is best to follow these practices:

Item 74: Implement Serializable judiciously

Implementing `Serializable` seems like it's easy because on its face, it just means adding another thing to the class declaration, but the default serialization is usually terrible and no matter what serialization you use, you often have to export internal pieces of the class, thus exposing them and removing all of the advantages of information hiding (such as freedom to change the internals of the class).

Furthermore, serializing and deserializing opens up the class to a new slew of security holes and possible bugs, many of which are very difficult to test. Deserialization is pretty much another constructor, which is often overlooked but needs to enforce the invariants of the class. Classes designed for inheritance and interfaces should rarely implement `Serializable` because this imposes the requirement to be careful about serialization on anyone who extends them.

Inner classes should never implement `Serializable`

Item 75: Consider using a custom serialized form

The default serialized form contains a lot of information about the inner workings of a class that does not belong in the serialized form. In general, the serialized form should contain only the logical part of a class (for example, externally available values but not the form that those values are stored in) and values that cannot be computed from other fields. The section also provides some other suggestions about the form of a class that implements `Serializable`.

Item 76: Write readObject methods defensively

The `readObject` (deserialization) method is basically another constructor. This section mentions several things to do, which are basically the same as mentioned above for public constructors. It also mentions several attacks that can be made due to faulty deserialization.

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

Enum types are guaranteed to have a limited number of instances, whereas `readResolve` can be attacked.

Item 78: Consider serialization proxies instead of serialized instances

In some cases, a nested class acting as a proxy for the serialized form can provide many of the benefits of the above approaches with less effort.

In conclusion: Some basic principles

Just a few short items that espouse basic principles which recur often in this book and are useful to have in mind for other languages too:

- Always design to have errors show up at compile time rather than runtime (taking advantage of type checking and generics is a good way to do this). Aside: This is one of the reasons I love Haskell. It seems like 80% of the idiomatic programs that typecheck are correct (or put another way: I fixed so many errors in my programs just because they wouldn't type check when incorrect).
- If you can not find the error at compile time, design to have them be found as soon as possible.
- Design classes and APIs so that even a malicious client could not break invariants and so that a malicious client could only harm itself.