



Effective Java

01 Consider static factory methods over constructors

Pros

Cons

02 Consider a builder when faced with many constructor arguments

03 Enforce singleton property with a private constructor or enum type

Public final field

Static Factory

Enum Singleton

04 Enforce noninstantiability with a private constructor

10 Obey the general contract when overriding equals

11 Always override hashCode when you override equals

14 Consider implementing Comparable

17 Minimize mutability

18 Favor composition over inheritance

19 Design and document for inheritance or else prohibit it

20 Prefer interfaces to abstract classes

21 Design interfaces for posterity

28 Prefer lists to arrays

30 Favor generic methods

31 Use bounded wildcards to increase API flexibility

40 Consistently use the Override annotation

42 Prefer Lambdas to anonymous classes

43 Prefer method references to lambdas

44 Favor the use of standard functional interfaces

45 Use streams judiciously

46 Prefer side-effect-free functions in streams

47 Prefer Collection to Stream as return type

48 Use caution when making streams parallel

58 Prefer for-each loops to traditional for-loops

59 Know and use the libraries

60 Avoid float and double if exact answers are required

61 Prefer primitive types to boxed primitives

64 Refer to objects by their interfaces

69 Use exceptions only for exceptional conditions

70 Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

71 Avoid unnecessary use of checked exceptions

72 Favor the use of standard exceptions

73 Throw exceptions appropriate to the abstraction

74 Document all exceptions thrown by each method

75 Include failure-capture information in detail messages

76 Strive for failure atomicity

77 Don't ignore exceptions

01 Consider static factory methods over constructors

Consider to use one or more static factory methods in order to replace constructors in creating (complex) instances in a streamlines manner.

Pros

- Declarative name
 - More readable code for specific instances which are created without referral of the docs
- No enforcing to always create a new object
 - Allows immutable classes to use pre-constructed instances
 - Allows to cache instances when they are created (e.g. Singleton Pattern)
- Subtypes as return values
 - Flexibility in API design
 - Allows to hide *implementation*-classes
- Return value can vary from call to call
 - Any subtype of the return value is allowed \Rightarrow allowing different implementations being used according to the calling args
 - Can be used to gradually roll out different versions of code
- Class of returned object mustn't exist when the factory is written

Cons

- Classes without `public` or `protected` constructors can't be subclassed
 - Could be a "hidden" benefit thus developers are forced to favor composition over inheritance
- Factory methods are hard to discover
 - No standouts in docs and No standardized IDE support
 - Require team/project standards for the naming

02 Consider a builder when faced with many constructor arguments

Both constructors and static factories share the limitation that they don't scale well with large number of optional arguments. In addition constructors with a lot of numeric arguments can look messy since one has to consult the docs to understand for what each arguments stands for.

The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters. It simulates named optional parameters thus enhancing the readability of the construction of complex objects.

03 Enforce singleton property with a private constructor or enum type

| Singletons are now considered an anti-pattern in most cases

Singletons make a class rather difficult to test since its impossible to mock an implementation of a singleton unless it implements an interface that serves as a type. There are three different ways in order to ensure only a single instance will exist during in runtime and how to provide it.

Public final field

This approach provides a simple API which makes it obvious that this class is a singleton.

```
class ChuckNorris {
    public static final ChuckNorris INSTANCE = new ChuckNorris();

    // Use private ctor to prevent instantiation
    private ChuckNorris() { ... }

    public void blowStuffUp() { ... }
}
```

Static Factory

This approach gives one more flexibility in implementing the creation of the singleton instance. Additionally it allows in the future to change if the class is a singleton or not without breaking its API. It can be uses as a `Supplier<ChuckNorris>`. However this approach requires some additional logic for serialisation purposes.

```
class ChuckNorris {
    private static final ChuckNorris INSTANCE;

    // Use private ctor to prevent instantiation
    private ChuckNorris() { ... }

    public static ChuckNorris getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new ChuckNorris();
        }

        return INSTANCE;
    }

    public void blowStuffUp() { ... }
}
```

Enum Singleton

Out of the three this is the recommended approach if the singleton class mustn't inherit from a specific superclass (except interfaces). It is more concise, provides the serialisation machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialisation or reflection attacks.

```
enum ChuckNorris {
    INSTANCE;

    public void blowStuffUp() { ... }
}
```

04 Enforce noninstantiability with a private constructor

When faced with some sort of utility- or other class which shouldn't be instantiated its best to prevent it altogether in the first place to create a clear API definition and reduce side-effects. The preferred way in doing so is a private constructor with a `AssertionError` inside it. The thrown error isn't strictly necessary but provides a handy safety net.

```
class SomeUtilityClass {  
    private SomeUtilityClass() {  
        throw new AssertionError();  
    }  
}
```

10 Obey the general contract when overriding equals

Per default every object is only *equal* to itself. There are valid reasons to change this by a custom `equals`-implementation. But there are also some cases where one shouldn't:

- Each instance of the class is inherently unique
- There is no need for the class to provide a *logical equality* test (e.g. `java.util.regex.Pattern`)
- A superclass has already overridden `equals` and the superclass behaviour is appropriate for this class
- The class is private or package-private and one is certain that its `equals`-method will never be invoked

If one wants an additional safety net one could go forward and implement the `equals`-method by throwing an `AssertionError`

Every implementation of the `equals`-method must adhere to the following four rules. If it doesn't this can lead to some nasty side effects and unexpected behaviour during runtime.

- **Reflexive**

For any non-null reference value `x`, `x.equals(x)` must return `true`

- **Symmetric**

For any non-null reference values `x` and `y` following conditions apply

- `x.equals(y) ⇒ true`
- `y.equals(x) ⇒ true`

- **Transitive**

For any non-null values `x`, `y`, `z` following conditions apply:

- `x.equals(y) ⇒ true`
- `y.equals(z) ⇒ true`
- `x.equals(z) ⇒ true`

- **Consistent**

For any non-null values `x`, `y` multiple invocations of `x.equals(y)` the return values must be consistent

- **Non-Null**

For any non-null reference value `x`, `x.equals(null)` must return `false`

When combining the rules above one comes to the following sample implementation for the class `Point`.

Note: that one has to check each relevant attributes of the class in the custom equals implementation (in the example these are the fields `x` and `y`).

```
class Point {
    private int x;
    private int y;

    public int getX() { ... }
    public void setX(int x) { ... }

    public int getY() { ... }
    public void setY(int y) { ... }

    @Override
    public final boolean equals(final Object object) {
        // Check for own reference
        if (object == this) {
            return true;
        }

        // Check if not-null and has correct instance type
        if (!(object instanceof final Point other)) {
            return false;
        }

        // Compare all relevant class attributes
        return this.x == other.x && this.y == other.y;
    }
}
```

Be vary when comparing native values with floating points (`float` and `double`). One should use the according compare utilities of the native type (e.g. `Float.compare()`) in order to prevent floating point errors. If one compares attributes which are objects, use their according `equals` -implementation.

11 Always override hashCode when you override equals



`equals` and `hashCode` must always be implemented and modified together

If the equals-implementation differs from the `hashCode` it can lead to some nasty behaviour during runtime and vice versa. The following rules state the contract when implementing `hashCode`:

- The return values of `hashCode` must be consistently the same when called multiple times (until the `equals` method is changed)
- If two objects are equal according to the `equals`-method, their respective return value of `hashCode` must be the same \Rightarrow all fields checked in the `equals`-implementation must be included in the hash-code computation
- If two objects aren't equal according to the `equals`-method, their respective hash mustn't be different, however it would be beneficial for performance reasons



If you compute a hash never come up with an own implementation \Rightarrow use available helpers from the standard libs

As stated above one should include all significant attributes of a class in its hash-code computation. For primitive values just use their according hashing implementation on the respective type (e.g. for `float` use `Float.hashCode(value)`). If you have a complex type (e.g. an object) use its respective implementation of `hashCode`.

For the example class `Point` from Effective Java Item 10 this would be the according `hashCode`-implementation:

```
@Override
public final int hashCode() {
    return Objects.hash(this.x, this.y);
}
```

In this example `Objects.hash(values...)` provides a simple utility for computing the hash-code from multiple attributes. However it isn't really performant since the

arguments array must be created and the arguments must be potentially boxed and unboxed. A more performant but also more complex implementation would be following:

```
@Override
public final int hashCode() {
    // Compute first hash
    int result = Integer.hashCode(this.x);

    // Add every other args hash code
    result = 31 * result + Integer.hashCode(this.y);

    return result;
}
```

If the class is immutable one could consider caching the hash-code after the first computation for more performance gains.

14 Consider implementing Comparable

The interface `Comparable<T>` allows to specify a *natural* order for instances of a certain class. It enables the class to operate with a lot of prebuilt (sorting-)algorithms of the standard-libs. Its implementation returns a numeric value which implies the order for the two objects being compared:

- return value is negative \Rightarrow this object is *smaller* compared to the given one
- return value is `0` \Rightarrow compared objects are *equal*
- return value is positive \Rightarrow this object is *greater* compared to the given one

In general it is recommended that the object-equality matches up with the one defined by the equals-implementation. If this isn't the case it should be mentioned explicitly to inform the consumer about potential behaviour differences.

The following two rules apply additionally for the implementation:

- if `x.compareTo(y)` throws an exception, `y.compareTo(x)` must throw an exception as well
- transitivity:
`x.compareTo(y) > 0 && y.compareTo(z) > 0 && x.compareTo(z) > 0`
- for all `z` the following statement must be true if `x.compareTo(y) == 0` :
`signum(x.compareTo(z)) == signum(y.compareTo(z))`

When one implements `compareTo`, the comparison order is critical. Start with the most significant attribute and continue your way down. If the comparison of each attribute resulted in `0` one can be certain that these objects are identical.

When comparing native types one should use the corresponding comparison helper of the type (e.g. for floats use `Float.compare(a, b)`). When comparing objects use either their `Comparable`-implementation or a suiting `Comparator`.

Remember that when you implement the `Comparable`-interface you specify the *natural order*. However usually this isn't the only order to exist for a type. If one needs another, more specific order, one can go forward and implement a respective `Comparator`.

17 Minimize mutability



Classes should be immutable unless there is a very good reason to make it mutable

Immutable classes are easier to implement and to maintain since one doesn't have to worry about changing state and the objects' reaction to that change. In addition they are inherently thread-safe and require no synchronisation and can be shared freely. Therefore if possible try to make classes immutable. To do so the following rules must be met:

- Don't provide methods that modify the object's state (mutators)
- Ensure that the class can't be extended
⇒ usage of `sealed` or `final`
- Make all fields `final`
- Make all fields `private`
- Ensure exclusive access to any mutable components
if the class uses other mutable objects ensure that their references aren't leaked to a consumer

When implementing an immutable class its constructors should create fully initialised objects with all their invariants established. If the class can't be fully immutable try to restrict its mutability as much as possible.

18 Favor composition over inheritance

Favor Composition over Inheritance (FCOI)



In case of doubt: Prefer composition (part-of) over inheritance (is-a)


Don't specialise any classes if they aren't specifically built for this purpose (e.g. mentioned in the docs) even if they don't restrict it (non-`final` class)

Composition leads to more flexible and maintainable code in the long run. Parts of the class / functionalities can be exchanged more easily with less side effects.

Patterns leveraging composition:

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. Imagine that you're creating a stock market monitoring app. The app downloads the


 <https://refactoring.guru/design-patterns/adapter>



REFACTORING
· GURU ·

Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors. Imagine that you're

 <https://refactoring.guru/design-patterns/decorator>



REFACTORING
· GURU ·

19 Design and document for inheritance or else prohibit it

If a class allows inheritance it should be designed for it and have the according docs which specify how the sub-class must be implemented. If this isn't the case prevent inheritance in the first place (`final` class, `private` -ctor)

If possible don't call methods which can be overridden, constructors or methods with similar behaviour (`clone()`) \Rightarrow danger of prenatal communication.

Only allow the implementation of abstract methods.

20 Prefer interfaces to abstract classes

In Java both abstract classes as well as interfaces allow to implement instance methods for their implementing/extending classes. In general interfaces should be preferred to abstract classes, when it comes to define a contract or base functionality.

- Existing classes can easily be retrofitted to implement a new interface, however changing the superclass can lead to a massive rattail of additional complexity or can be virtually impossible at a later stage.
- Interfaces are ideal for defining mix-ins. Therefore they allow one to easily extend a classes behaviour without having to modify its inheritance chain ⇒ Favor Composition over Inheritance
- Interfaces allow for the construction of hierarchical type frameworks

21 Design interfaces for posterity

Prior to Java 8, it was impossible to add methods to interfaces without breaking its implementations. They would lack the added method and therefore cause a compile-time error. With Java 8 the possibility to declare *default methods* in interfaces was added to fight this issue.

The default implementation of an interface method is used by all classes which implement the interface but don't override it (e.g. declare their own implementation). One could be mislead into thinking that this solves the issue at hand and allows for extending interfaces without breaking their implementing classes.



It isn't always possible to implement a general default method which takes care of all invariants of every conceivable implementation

There won't be any compile-time errors, way worse, there can be runtime- or behavioural-errors. In conclusion one should design interfaces still with great care. Default methods can save one when one has to correct an interface later on but one shouldn't count on it.

28 Prefer lists to arrays



If you find yourself mixing them and getting compile-time errors or warnings, your first impulse should be to replace the arrays with lists.

Arrays and generics have very different type rules. Arrays are covariant and reified, generics on the other hand are invariant and erased. In plain english this means that arrays provide runtime type safety but not compile-time safety and vice versa. Therefore arrays and generics don't mix well.

30 Favor generic methods

Generic methods, like generic types, are safer and easier to use than methods requiring their clients to put explicit casts on input parameters and return values. Like types, you should make sure that your methods can be used without casts, which often means making them generic. And like types, you should generify existing methods whose use requires casts. This makes life easier for new users without breaking existing clients. The only exception could be explicit implementations where each type requires a specific implementation in order to achieve the functionality or better performance.

31 Use bounded wildcards to increase API flexibility

When using generics in methods or types one faces the issue that the type `List<Integer>` is not a subtype of `List<Number>`. If we consider the following method:

```
public void doStuff(List<Number> numbers) { ... }
```

It will only accept arguments which implement `List<Number>`. In order to write a generic component for all subtypes of a type one can use a bounded wildcard. As the name states it isn't a full wildcard - it has a boundary in place. For the method above to accept all `List`s which house a subtype of `Number` it would look like the following:

```
public void doStuff(List<? extends Number> numbers) { ... }
```

One can take the following mnemonic to decide on which wildcard type to use



PECS → **P**roducer-**e**xtends, **c**onsumer-**s**uper

Following statements aid you further with the implementation of bounded wildcards:

- for maximum flexibility, use wildcard types on input parameters that represent producers or consumers
- do not use bounded wildcard types as return types
- if a type parameter appears only once in a method declaration, replace it with a wildcard

40 Consistently use the Override annotation



Use the Override annotation on every method declaration that you believe to override a superclass declaration

The `@Override` annotation tells the compiler that one wants to override a certain method with the given implementation, thus enabling the compiler or IDE to give one hints and warnings about broken implementation. Given the following example:

```
class Bigram {  
    // ...  
    public final boolean equals(Bigram b) {  
        return b.first == first && b.second == second  
    }  
}
```

In this example it is obvious that this is a broken `equals` -implementation. However it is a perfectly valid instance method from the perspective of the Java compiler. Thus the program will compile without any issue. These issue will arrive during runtime when the *erratic* behaviour starts occurring based on the faulty equals implementation. In the end this is a simple pitfall with potentially huge effects which could have been avoided if the developer would have added the `@Override` annotation. In this case the compiler would have recognised the faulty method signature and hinted this to the developer.

42 Prefer Lambdas to anonymous classes

For a long time, anonymous classes were the trick to implement functional interfaces in Java-land. A sort implementation based on the string length would look like this with an anonymous class:

```
Collections.sort(words, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return Integer.compare(s1.length(), s2.length());  
    }  
});
```

In this example it is fairly concise, however quite a lot of code for such a *simple* task at hand. A simple drop-in replacement would be a lambda-expression:

```
Collections.sort(words, (s1, s2) -> Integer.compare(s1.length, s2.length));
```

With this simple refactoring one reduced five lines of code to a single one without the loss of any functionality nor readability - it is even more readable now. The following rules can assist one when implementing lambdas:

- Omit the types of all lambda parameters unless their presence makes your program clearer
- Lambdas lack names and documentation; if a computation isn't self-explanatory, or exceeds a few lines, don't put it in a lambda
- One should rarely, if ever, serialise a lambda
- Don't use anonymous classes for function objects unless you have to create instances of types that aren't functional interfaces

43 Prefer method references to lambdas



Where method references are shorter and clearer, use them; where they aren't, stick with lambdas

Lambdas can help us make our code more concise. For example if we want to convert a `String` to its lowercase form the following lambda would do the trick:

```
str -> str.toLowerCase()
```

However we can do better than that. The lambda above just *calls* the according method but doesn't do anything additionally. Hence we can replace it using a *method reference* like so:

```
String::toLowerCase
```

44 Favor the use of standard functional interfaces

Now that Java has lambdas, it is imperative that you design your APIs with lambdas in mind. Accept functional interface types on input and return them on output. It is generally best to use the standard interfaces provided in `java.util.function.Function`, but keep your eyes open for the relatively rare cases where you would be better off writing your own functional interface.

Standard interfaces

Interface	Functional Signature	Example
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T, R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

45 Use streams judiciously



Overusing streams makes programs hard to read and maintain

When we look at the following implementation - is it easy to read?

```
class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(groupingBy(word -> word.chars()
                .sorted()
                .collect(StringBuilder::new,
                    (sb, c) -> sb.append((char) c),
                    StringBuilder::append)
                .toString()))
                .values()
                .stream()
                .filter(group -> group.size() >= minGroupSize)
                .map(group -> group.size() + ": " + group)
                .forEach(System.out::println);
        }
    }
}
```

Certainly not. However if we tone it down a bit the usage of a stream can start to shine:

```
public class Anagrams {
    public static void main(String[] args) throws IOException {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary)) {
            words.collect(groupingBy(word -> alphabetize(word)))
                .values().stream()
                .filter(group -> group.size() >= minGroupSize)
                .forEach(g -> System.out.println(g.size() + ": " + g));
        }
    }
}
```

The following statements assist you when working with streams:

- In the absence of explicit types, careful naming of lambda parameters is essential to the readability of stream pipelines
- Using helper methods is even more important for readability in stream pipelines than in iterative code
- Refrain from using streams to process `char`-values
- Refactor existing code to use streams and use the in new code only where it makes sense to do so

Good use cases for streams:

- Uniformly transform sequences of elements • Filter sequences of elements
- Combine sequences of elements using a single operation (for example to add them, concatenate them, or compute their minimum)
- Accumulate sequences of elements into a collection, perhaps grouping them by some common attribute
- Search a sequence of elements for an element satisfying some criterion

Some tasks are best accomplished with streams, and others with iteration. Many tasks are best accomplished by combining the two approaches. There are no hard and fast rules for choosing which approach to use for a task, but there are some useful heuristics. In many cases, it will be clear which approach to use; in some cases, it won't. If you're not sure whether a task is better served by streams or iteration, try both and see which works better.

46 Prefer side-effect-free functions in streams

The essence of programming stream pipelines is side-effect-free function objects. This applies to all of the many function objects passed to streams and related objects. The terminal operation `forEach` should only be used to report the result of a computation performed by a stream, not to perform the computation. In order to use streams properly, you have to know about collectors. The most important collector factories are `toList`, `toSet`, `toMap`, `groupingBy`, and `joining`.

47 Prefer Collection to Stream as return type

When writing a method that returns a sequence of elements, remember that some of your users may want to process them as a stream while others may want to iterate over them. Try to accommodate both groups. If it's feasible to return a collection, do so. If you already have the elements in a collection or the number of elements in the sequence is small enough to justify creating a new one, return a standard collection such as `ArrayList`. Otherwise, consider implementing a custom collection. If it isn't feasible to return a collection, return a stream or iterable, whichever seems more natural. If, in a future Java release, the Stream interface declaration is modified to extend `Iterable`, then you should feel free to return streams because they will allow for both stream processing and iteration.

48 Use caution when making streams parallel

Parallelising a pipeline is unlikely to increase its performance if the source is from `Stream.iterate`, or the intermediate operation `limit` is used. Therefore do not parallelise stream pipelines indiscriminately. Not only can parallelising a stream lead to poor performance, including liveness failures; it can lead to incorrect results and unpredictable behaviour (safety failures).

Performance gains from parallelism are best on streams over `ArrayList`, `HashMap`, `HashSet` and `ConcurrentHashMap` instances; arrays; `int` ranges; and `long` ranges. All of these datatypes can be split accurately and cheaply into subranges of any desired size which makes it easy to divide them among parallel tasks. In order to do so the `splitter`-method on a `Stream` or `Iterable` can be used.

Do not even attempt to parallelise a stream pipeline unless you have good reason to believe that it will preserve the correctness of the computation and increase its speed. The cost of inappropriately parallelising a stream can be a program failure or performance disaster. If you believe that parallelism may be justified, ensure that your code remains correct when run in parallel, and do careful performance measurements under realistic conditions. If your code remains correct and these experiments bear out your suspicion of increased performance, then and only then parallelise the stream in production code.

58 Prefer for-each loops to traditional for-loops

Dated approach for iterating a collection, using a `for`-loop:

```
for (Iterator<Element> i = elements.iterator(); i.hasNext();) {  
    Element e = i.next();  
    // Do stuff...  
}
```

More concise, preferred way for iterating a collection, using the `foreach`-loop:

```
for (Element e : elements) {  
    // Do stuff...  
}
```

However this isn't a silver bullet that can be generally applied anywhere. In the following situations one needs a dedicated iterator:

- **Destructive filtering**
- **Transforming**
- **Parallel iteration**

The `foreach`-loop provides compelling advantages over the traditional for loop in clarity, flexibility, and bug prevention, with no performance penalty. Use for-each loops in preference to for loops wherever you can.

59 Know and use the libraries

By using the standard library, you take advantage of the knowledge of the experts who wrote it and the experience of those who used it before you. Numerous features are added to the libraries in every major release and it pays off to stay informed about these additions.



Every (Java-)programmer should be familiar with the basics of `java.lang`, `java.util` and `java.io` and their sub-packages

Don't reinvent the wheel. If you need to do something that seems like it should be reasonably common, there may already be a facility in the libraries that does what you want. If there is, use it; if you don't know, check. Generally speaking, library code is likely to be better than code that you'd write yourself and is likely to improve over time. This is no reflection on your abilities as a programmer. Economies of scale dictate that library code receives far more attention than most developers could afford to devote to the same functionality.

60 Avoid float and double if exact answers are required

The `float` and `double` types are designed primarily for scientific and engineering calculations. They perform *binary floating-point arithmetic*, which was carefully designed to furnish accurate approximations quickly over a broad range of magnitudes. However based on that every calculation is only accurate to a certain point and therefore can have a floating-point rest, which can lead to unwanted behaviour → especially when dealing with supposedly *exact* numbers (such as in the finance domain).

Don't use `float` or `double` for any calculations that require an exact answer. Use `BigDecimal` if you want the system to keep track of the decimal point and you don't mind the inconvenience and cost of not using a primitive type. Using `BigDecimal` has the added advantage that it gives you full control over rounding, letting you select from eight rounding modes whenever an operation that entails rounding is performed. This comes in handy if you're performing business calculations with legally mandated rounding behaviour. If performance is of the essence, you don't mind keeping track of the decimal point yourself, and the quantities aren't too big, use `int` or `long`. If the quantities don't exceed nine decimal digits, you can use `int`; if they don't exceed eighteen digits, you can use `long`. If the quantities might exceed eighteen digits, use `BigDecimal`.

61 Prefer primitive types to boxed primitives



Applying the == operator to boxed primitives is almost always wrong

Use primitives in preference to boxed primitives whenever you have the choice. Primitive types are simpler and faster. If you must use boxed primitives, be careful! Autoboxing reduces the verbosity, but not the danger, of using boxed primitives. When your program compares two boxed primitives with the == operator, it does an identity comparison, which is almost certainly *not* what you want. When your program does mixed-type computations involving boxed and unboxed primitives, it does unboxing, and when your program does unboxing, it can throw a `NullPointerException`. Finally, when your program boxes primitive values, it can result in costly and unnecessary object creations.

64 Refer to objects by their interfaces



If you get into the habit of using interfaces as types, your program will be much more flexible

If appropriate interface types exist, then parameters, return values, variables and fields should all be declared using these interface types.

```
// Use interface as var type ...
Set<Integer> uniqueNumbers = new LinkedHashSet<>();

// ... rather than the explicit implementation type
LinkedHashSet<Integer> uniqueNumbers = new LinkedHashSet<>();
```

However it is entirely appropriate to refer to an object by a class rather than an interface if no appropriate interface exists \Rightarrow if there is no appropriate interface, just use the least specific class in the class hierarchy that provides the required functionality.

69 Use exceptions only for exceptional conditions



Exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow

A horrific abuse would be the following example which leverages an exception for controlling logic:

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

These kind of statements are not only slower, they provide really bad readability. If this code would have been written with the proper control structures it is faster and far more legible:

```
for (Mountain mountain : range) {
    mountain.climb();
}
```

A well-designed API must not force its clients to use exceptions for ordinary control flow. Therefore use exceptions only for their designed purpose and don't write APIs which force others to abuse them.

70 Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

Checked exceptions should be used for conditions from which the caller can reasonably be expected to recover.

Unchecked exceptions are either runtime exceptions or errors. In general both of these types shouldn't be caught since the recovery from such an error is generally impossible. Use runtime exceptions for programming errors which can occur. For example an `ArrayOutOfBoundsException` would be a typical runtime exception. If you implement your own runtime exceptions make sure that they inherit from `RuntimeException` (directly or indirectly) and never directly from `Error`.

Throw checked exceptions for recoverable conditions and unchecked exceptions for programming errors. When in doubt, throw unchecked exceptions. Don't define any throwables that are neither checked exceptions nor runtime exceptions. Provide methods on your checked exceptions to aid in recovery.

71 Avoid unnecessary use of checked exceptions

When used sparingly, checked exceptions can increase the reliability of programs; when overused, they make APIs painful to use. If callers won't be able to recover from failures, throw unchecked exceptions. If recovery may be possible and you want to *force* callers to handle exceptional conditions, first consider returning an optional. Only if this would provide insufficient information in the case of failure should you throw a checked exception.

72 Favor the use of standard exceptions



The DRY-principle applies for an entire codebase and certainly doesn't stop with exceptions

The goal should be to rely on standard exceptions as much as possible. However the classes `Exception`, `RuntimeException`, `Throwable` or `Error` should never be used directly. The following exceptions are some of the most used ones:

Exception	Usage
<code>IllegalArgumentException</code>	Non-null parameter value is inappropriate
<code>IllegalStateException</code>	Object state is inappropriate for method invocation
<code>NullPointerException</code>	Parameter value is null but is expected to have a value
<code>IndexOutOfBoundsException</code>	Index parameter value is out of range
<code>ConcurrentModificationException</code>	Concurrent modification of an object has been detected where it is prohibited
<code>UnsupportedOperationException</code>	Object doesn't support method

Choosing which exception to reuse can be tricky because the "occasions for use" in the table above do not appear to be mutually exclusive. Consider the case of an object representing a deck of cards, and suppose there were a method to deal a hand from the deck that took as an argument the size of the hand. If the caller passed a value larger than the number of cards remaining in the deck, it could be

construed as an `IllegalArgumentException` (the `handSize` parameter value is too high) or an `IllegalStateException` (the deck contains too few cards). Under these circumstances, the rule is to throw `IllegalStateException` if no argument values would have worked, otherwise throw `IllegalArgumentException`.

73 Throw exceptions appropriate to the abstraction



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

This means, that lower-level exceptions shouldn't be exposed since it pollutes the calling code because it has to deal with exceptions from a context it doesn't know. The higher-level code should either replace or wrap any eventual lower-level exceptions in order to provide an expectable and streamlined API.

However while exception translation is superior to mindless propagation of exceptions from lower layers, it shouldn't be overused. If possible try to prevent lower-layer exceptions in the first place by ensuring everything is in correct shape and form.

If it isn't feasible to prevent or to handle exceptions from lower layers, use exception translation, unless the lower-level method happens to guarantee that all of its exceptions are appropriate to the higher level. *Chaining* provides the best of both worlds: it allows you to throw an appropriate higher-level exception, while capturing the underlying cause for failure analysis.

74 Document all exceptions thrown by each method

Document every exception that can be thrown by each method that you write. This is true for unchecked as well as checked exceptions, and for abstract as well as concrete methods. This documentation should take the form of `@throws` -tags in doc comments. Declare each checked exception individually in a method's throws clause, but do not declare unchecked exceptions. If you fail to document the exceptions that your methods can throw, it will be difficult or impossible for others to make effective use of your classes and interfaces.

75 Include failure-capture information in detail messages

To capture a failure, the detail message of an exception should contain the values of all parameters and fields that contributed to the exception. However be vary to not leak any sensitive information (credentials, encryption keys etc.) in the logs.

Since this idiom is beneficial for debugging and understanding the issue at hand, the standard libraries don't make heavy use of this idiom. For example the following implementation would be a better alternative to the standard string-constructor of the `IndexOutOfBoundsException` :

```
/**
 * Constructs an IndexOutOfBoundsException.
 *
 * @param lowerBound the lowest legal index value
 * @param upperBound the highest legal index value plus one
 * @param index the actual index value
 */
public IndexOutOfBoundsException(int lowerBound, int upperBound, int index) {
    // Generate a detail message that captures the failure
    super(String.format("Lower bound: %d, Upper bound: %d, Index: %d",
        lowerBound,
        upperBound,
        index)
    );

    // Save failure information for programmatic access
    this.lowerBound = lowerBound;
    this.upperBound = upperBound;
    this.index = index;
}
```

76 Strive for failure atomicity



A failed method invocation should leave the object in the state that it was prior to the invocation.

As a rule, any generated exception that is part of a method's specification should leave the object in the same state it was in prior to the method invocation. Where this rule is violated, the API documentation should clearly indicate what state the object will be left in. Unfortunately, plenty of existing API documentation fails to live up to this ideal.

77 Don't ignore exceptions

As the name states, exceptions should not be ignored in general \Rightarrow an empty `catch` - block defeats its purpose! However if there aren't any recovery measures to be taken, at least write a comment to document why this may be the case.