Short Circuit

Java

Effective Java
by Joshua Bloch
Third Edition

Summary by Emiel Bos

1 Ecosystem

While C++ compiles to a native machine code binary that can be directly executed, Java compiles to a portable, intermediate format (bytecode) that requires a runtime container (JVM) to execute. Contrary to C++, Java is a *safe language*, which means that, in the absence of native methods, it is immune to buffer overruns, array overruns, wild pointers, and other memory corruption errors that are present in languages that treat all of memory as one giant array. Unlike C++, Java has automatic garbage collection.

The JVM + the Java Class Library (JCL; Java's standard library and analogue to C++'s standard library) are packed into the Java Runtime Environment (JRE). The JRE + development tools (javac, Javadoc, profilers, etc.) are packed into the Java Development Kit (JDK). Numerous features are added to the libraries in every major release, and it pays to keep abreast of these additions [Item 59]. Installing Java on Windows is as simple as downloading a preferred version of JDK, unzipping the compressed version to a directory of choice (e.g. `%PROGRAMFILES%/Java/jdk-1?.?.?`), setting that directory as `%JAVA_HOME%` environment variable, and adding `%JAVA_HOME%/bin` to `PATH`. You can use the installer version as well, but this approach is more convenient if you want to have and use multiple versions of JDK.

Java code uses *packages* to organize the namespace for classes and reduce the risk of name collisions. Classes are declared part of a package with the `package` keyword at the top of the corresponding file: `package <packagename>[.<subpackagename>....<sub`
They can be imported with the `import` keyword that must come after any `package` declarations: `import <packagename>[.<subpackagen`
When a class does not include a package declaration, it is considered to be in the "default package", but this should be avoided.[1] Package (and module) names should be hierarchical with the components separated by periods [Item 68]. Components consist of lowercase alphabetic characters and sometimes digits, and should be short and can be meaningful abbreviations or acronyms. The name of any package that will be used outside your organization should begin with your organization's domain name with the components reversed, e.g. `org.eff`. Many packages have names with just one component next to the domain name, but large facilities can have an entire hierarchy of components. The standard libraries and optional package begin with `java` or `javax`.
A class's *fully-qualified*/official/technical name includes its package name. You can use a class without `import`, but then you must refer to it by its fully-qualified name. The directory path of the file `classname.java` must correspond to the fully-qualified name. Use a `*` wildcard to import all classes in a package, e.g. `import java.util.*;`. `import java.*;` doesn't import anything because there are no classes directly in package `java`. Everything in the package `java.lang` is automatically/implicitly imported into every program. `import static <packagename>.<classname>.<membername>` to import a static member, which allows using that static member without specifying the class.

A *module* is a grouping of packages, like a package is a grouping of classes. If a library uses the module system, its exported API is the union of the exported APIs of all the packages explicitly exported by the library's module declaration, which conventionally is in file called `module-info.java`.

Java programs and third party libraries are typically packaged as Java Archive (JAR) files, which aggregate many Java class files and associated metadata and resources into one archive file for distribution. JAR is built on ZIP and is almost

[1] It can cause particular problems for Spring Boot applications that use the `@ComponentScan`, `@ConfigurationPropertiesScan`, `@EntityScan`, or `@SpringBootApplication` annotations, since every class from every JAR is read.

identical, but JAR adds a manifest file and signatures. Since Java is a dynamically bound language, when you run a Java application with library dependencies, the JVM needs to know where the dependencies are, which can be done either by repacking the application and its dependencies into a single JAR file, or telling the JVM where to find the dependent JAR files via the runtime classpath. Classpath is a parameter in the JVM or the Java compiler that specifies the location of user-defined classes and packages, and can be set either in the command line with the `-cp`/`-classpath` option: `java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp`, or with the `CLASSPATH` environment variable, which is best done in a batch file:

```
#!/bin/bash

export DIR=/usr/libexec/myApp
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar
java com.example.MyApp
```

Executable JAR files are the simplest way to assemble Java code into a single file that can be executed, which you can do with `java -jar <jar-path> [args...]`. Note that a `-cp`/`-classpath` option will be ignored if you use `-jar`, because the application's classpath is determined by the JAR file manifest. When an application has not been packaged as an executable JAR, you need to provide the name of an entry-point class on the java command line.

Java does not provide a standard way to load nested JARs (JARs in another JAR), which would allow for distributing a self-contained application, i.e. containing also all the JAR dependencies. To solve this problem, many developers use "uber" JARs that packages all the classes from all the application's dependencies into a single archive. The problem with this approach is that it becomes hard to see which libraries are in your application, and can cause name collisions. (Spring Boot takes a different approach and lets you actually nest JARs directly.)

1.1 Programs

Even though it is not enforced by the compiler, each `classname.java` file should contain at most one *top-level* (i.e. not nested) class or interface and the file must have the same name [Item 25]. Compile with `javac <classname>.java` (the Java compiler), which will compile it to Java bytecode, contained in a `classname.class` file (i.e., with the same name). You run this in the Java Virtual Machine (JVM) with `java <classname>` (no extension), which will call the `main()` of that class. `main()` has the following signature (unlike C++, it returns `void`):

```
public static void main(String[] args) {

}
```

A *JavaBean* is just a standard; it is a regular class except it follows the conventions that all properties are private and only accessed through getters and setters, has a public no argument ("nullary") constructor, and implements the `Serializable` interface. It's largely obsolete.

2 Syntax

Java's syntax is mostly derived from C/C++. Bitwise right shift operator has a signed and unsigned version. The signed version is the regular `>>` operator which fills the new leftmost bit with the sign (the former leftmost bit), i.e. negative values get `1` as new leftmost bit and positive values get `0` as new leftmost bit. The unsigned bitwise right shift `>>>` always shifts a `0` into the leftmost position. The bitwise complement operator is `~`.

The *scope* of a local variable extends from the point where it is declared to the end of the enclosing block. Minimize the scope of local variables [Item 57]. The easiest way of ensuring this is to declare them where they're first used. Initialize them at their declaration (except when the initializing expression can throw a checked exception and the variable needs to be used outside the enclosing try block). Because the for-loop allows declaring loop variables, you should prefer it to the while-loop. Furthermore, you should prefer the *for-each loop* (`for (Element e : c)`) to the (regular) for-loop, because it gets rid of the clutter and the opportunity for error by hiding the iterator or index variable [Item 58], although you can't do this for:

- Destructive filtering; you need to use an explicit iterator for removing elements, so that you can call its `remove()` method (although you can often avoid this using Collection's `removeIf()`). For example:

  ```
  for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
     Element e = i.next();
     if(e.equals(5)) i.remove();
  ```

```
    }
```

- Transforming; you need the list iterator or array index for replacing the values of elements.
- Parallel iteration; you need explicit control over the iterator or index variable for iterating multiple collections in parallel, because you need to advance them in lockstep.

In those cases, you need a regular for-loop. The benefit is that you can define multiple loop variables, e.g.:

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    // Do something with i;
}
```

You can print stuff with `java.lang.System.out.println()`. Because all classes in `java.lang` are automatically imported, you can omit that part. `out` is a public static final field of type `PrintStream` that is an already opened output stream representing standard out, i.e. it typically corresponds to display output or another output destination specified by the host environment or user.

Datetimes are complicated, with multiple timezones, multiple offsets, and timezone regions that change over time. `java.util.Date` is obsolete and should no longer be used in new code; use `java.time.Instant`, `java.time.LocalDateTime` or `java.time.ZonedDateTime` instead.[2] `ThreadLocalRandom` should be used instead of `Random`, which generates higher quality random numbers and is faster.

`String`s are poor substitutes for other value types, enum types, aggregate types, and capabilities (which should be unique, unforgeable keys) [Item 62]. The string concatenation operator (+) is convenient but doesn't scale; concatenating n strings is $O(n^2)$, because `String`s are immutable and the operands are copied [Item 63]. `java.lang.String` has a mutable `StringBuilder` companion class, with `append()`, `insert()` and `toString()` as principal operations, which are overloaded so as to accept data of any type and then converts that data to a string.

Unlike C++, in Java there is no operator overloading, there are no unsigned integer types, and things like pointers and references are abstracted away.

3 Primitive types

Java has the following primitives that can be cast/converted: `byte`[3], `char`, `short`, `int`, `long`, `float`, and `double`. Implicit casting/conversion is possible when the source type has smaller range (called promotion), else you have to explicitly cast, which works the same as in C++ with parentheses. `float`s and `double`s are rounded down when cast to `int`s or `long`s. The `boolean` type cannot be cast to or from. `char` is a 16-bit integer representing Unicode. It can be cast to `byte`, which will drop the largest 8 bits, but this is safe for ASCII characters. Converting a `char` to an `int` indicating its position in the alphabet can be done with `c - 'a'`.

Each primitive has a corresponding *boxed-primitive* class, whose name is a capitalized, written-out version of its primitive (e.g. `java.lang.Integer`). Primitives can be automatically converted to its wrapper object and vice versa via *autoboxing*. Boxing is converting from a primitive to its reference type, and unboxing is the inverse. This blurs but does not erase the distinction between the primitive and boxed primitive types, and you should prefer primitive types to boxed primitives [Item 61]. Because boxed primitives are reference types, they have identities distinct from their values (whereas primitives have only their values). As a consequence, applying the `==` operator to boxed primitives (and most other reference value types like `String`s) is almost always wrong, because it compares references/addresses. Primitives can't be null, boxed primitives can, and are default initialized to null (like all nonconstant reference fields). In nearly every case when you mix primitives and boxed primitives in an operation, the boxed primitive is auto-unboxed, and this can lead to a `NullPointerException`. Last, primitives are more time- and space-efficient than boxed primitives, and accidental autoboxing can be a performance degrader if done repeatedly. However, you must use boxed primitives as type parameters in parameterized types and methods (e.g. any collection type), because the language does not permit you to use primitives.

The `float` and `double` types perform binary floating-point arithmetic, which was carefully designed to furnish accurate approximations quickly over a broad range of magnitudes. However, they do not provide exact results and should not be used where exact results are required, such as monetary calculations, because it is impossible to represent 0.1 (or any other negative power of ten) as a `float` or `double` exactly [Item 60]. Instead, use `BigDecimal` (gives you full control over rounding, letting you select from eight rounding modes, but is less convenient and slower than primitive arithmetic types),

[2]Refer to this StackOverflow answer for a thorough discussion.
[3]Unfortunately, Java doesn't have byte literals, and for some reason, the `byte` type is signed.

`int`, or `long` for monetary calculations. Use `BigDecimal`'s `String` constructor rather than its `double` constructor to avoid introducing inaccurate values into the computation.

Every primitive type has a default initialization value when not explicitly initialized. All number types are default initialized as 0, booleans are default initialized as `false`. Default initialization only occurs for class/instance fields; local variables aren't initialized, so they do not exist in a usable state in memory and using them results in a compilation error.

4 Reference types

Reference types include class types, interface types, and array types. Variables of reference type can be null. Java 10 introduced `var` as a type specifier for inferring types automatically, so it's like C++'s `auto`. While primitive values are stored directly in variables, instantiations of reference types (objects and arrays) are always stored as references in variables. Reference types are default initialized as null, but like with primitive types, this only holds for class/instance fields, and local variables have to be explicitly initialized to at least null in order for them to be used without a compilation error.

4.1 Classes

Unlike in C++, in Java there are no global functions or variables. All code belongs to a class (or interface with default methods), and all values are objects (except for primitive types). Methods and fields comprise the *members* of a class. `static` members of a class belong to the class itself and not to a specific instantiation, so they are called using the class name and do not require the creation of a class instance.

Classes can have a number of keywords/modifiers in their declaration. The access modifiers specify which other classes can access their members, allowing *information hiding* or *encapsulation* (an ideal in which components are oblivious to each others' inner workings and communicate only through their APIs):

- `private`; only own class.
- package-private (default); other classes inside same package.
- `protected`; other classes inside same package and extended classes in other packages.
- `public`; all classes.

Make each class or member as inaccessible as possible [Item 15]. public and package-private members are part of a class's implementation and not its exported API, while protected and public members are part of the class's exported API and must be maintained forever. For public classes, always make fields private and provide accessor methods (getters and setters) to preserve the flexibility to change the class's internal representation, even for immutable (final) fields, although it's slightly less harmful in that case [Item 16]. There is nothing inherently wrong with exposing the data fields of package private classes (provided they do an adequate job of describing the abstraction provided by the class), because client code is confined to the same package.
`abstract` specifies that a class can have zero or more abstract methods, which means it can only serves as a base class and cannot be instantiated; it needs to be extended in order for its properties to be used. `final` specifies that it cannot be extended from and cannot have any subclasses.

The dot `.` operator is used to access member fields and methods. Since it dereferences the object, it is actually basically the `->` operator in C++. Use the `this` keyword inside a class to refer to itself.

An object is instantiated with the `new` keyword like so: `Object obj = new Object();`. This reserves memory for the object, calls the class's constructor to initialize that memory, and returns the memory address. A reference variable is null when it does not reference any object. A constructor first calls the constructor of its class's superclass, then initializes its own member fields, and then executes its own code. You can initialize member fields inline, at their declaration, or inside the constructor, and this is largely a matter of preference. Rule of thumb: prefer initialization in constructor if you have a constructor parameter that changes the value of the field, and initialization in declaration otherwise. If no constructor is provided, Java generates a default constructor.
It's worth considering implementing static factory methods over constructors for various reasons [Item 1]. *Static factory methods* simply return an instance of the class, and in the JCL they can be recognized by names such as: `from()` for type conversion, `of()` for aggregation methods (e.g. `List.of(1, 2, 3)` for immutable `List` objects), `valueOf()`, `instance()`/`getInstance()`, etc.
If your constructor has many parameters and many of those optional, consider a *builder*, a public inner subclass that a client calls the constructor of and gets a builder object of, on which setter-like methods are called – only for the fields for

which the client has a value to specify – and which lastly returns a new instance of the actual object itself with a `build()` method [Item 2].
Constructors can be private, which is useful for *singleton* classes; of which only one instance is allowed. Besides a private constructor, you export a static member (either a public static field, or a private static field plus a public static `getInstance()` method) that holds the instance. However, best practice is to implement singletons as single-element enums, with one constant (e.g. `INSTANCE`) [Item 3]. Private constructors are also the only way to ensure non-instantiability [Item 4], e.g. for utility classes with only static methods such as `java.lang.Math` or `java.util.Arrays`.

When a variable of an object gets out of scope, the reference is broken. When there are no references left, the object gets marked as garbage. The garbage collector then collects and destroys it some time afterwards. Because of Java's garbage collection, a class has no destructor, though every object has a `finalize()` method called prior to garbage collection, which can be overridden to implement finalization. However, you should not do this, since it is unpredictable, slow and unnecessary [Item 8]. Cleaners are slightly better, but its better to still avoid those as well. Instead, for objects that encapsulate resources that require termination such as files, threads or socket handles, implement `AutoCloseable`. This interface declares the `close()` method, which is called automatically when exiting a try-with-resources block for which the object has been declared in the resource specification header. If used outside such a block, the user/client of the class must `close()` it themselves. An instance must keep track of whether it has been closed in a field and throw an `IllegalStateException` if one of its methods gets called when closed.

Implicit casting for objects happens when the source type extends or implements the target type (i.e. casting to a superclass or interface). Java provides the `instanceof` operator to test if an object is of a certain type, or a subclass of that type, useful for testing whether to cast.

Class, interface, enum and annotation type names should consist of one or more words with the first letter of each word capitalized [Item 68]. Abbreviations should be avoided. The generally accepted convention is to only capitalize the first letter of acronyms, e.g. `HttpUrl`.

4.1.1 Fields

Fields are variables belonging to a class. Fields can have keywords/modifiers: `static` fields aren't tied to an instance, `final` is for fields that can only be initialized once (so it's basically C++'s `const`) in a constructor or during its declaration (whichever is earlier), `transient` indicates that this field will not be stored during serialization, and `volatile` ensures all threads see a consistent value for the variable. Fields can also have the same access modifiers as classes.

A *constant field* is a static final field whose value is immutable, i.e. it holds a primitive type or an immutable reference type, e.g. enum constants. If a static final field has a mutable reference type, it can still be a constant field if the referenced object is immutable.

Method and field names follow the same typographical conventions as classes and interfaces, except that the first letter should be lowercase [Item 68]. Constant field names, however, should consist of one or more uppercase words separated by underscores. Ensure that public constant fields are either primitive values or references to immutable objects (so they can't be arrays, for example).

4.1.2 Methods

Methods are functions belonging to a class. Methods can have keywords/modifiers: `abstract` for *abstract methods* (methods with no implementation, so subclasses need to provide their implementation or be declared `abstract` themselves), `final` for methods that can't be overridden (methods are by default virtual, unlike C++), `static` methods (and fields) aren't tied to an instance (non-static methods are therefore also called *instance methods*), `native` for methods implemented in platform-dependent code (outside Java), and `synchronized` for methods which specifies that threads need to acquire monitor which is the class itself (or `java.lang.Object` for `static` classes). Methods in an interface can have the `default` modifier for a default (non-abstract) method (methods in an interface are by default abstract). Methods can also have the same access modifiers as classes. The `throws` keyword after the parameter list indicates that a method can throw exceptions, with those possible exceptions listed after the keyword. Methods in Java can't have default argument values, as in C++, and are usually overloaded instead.
The Java Native Interface (JNI) allows for *native methods* (specified with the `native` keyword), which are implemented in *native programming languages* such as C/C++. They provide access to platform-specific facilities (e.g. registries) and to existing (legacy) libraries of native code, and are used to write performance-critical parts of applications in native languages. However, is raraly necessary to use native methods to access platform-specific facilities, because Java already provides access to many features previously found only in host platforms, and it is rarely advisable to use native methods for improved performance, because JVMs have gotten much faster [Item 66]. Native methods are not immune to memory

corruption errors (because native languages are not safe), make programs less portable and harder to debug, can decrease performance because the garbage collector can't automate or track native memory usage (and going into and out of native code is expensive), and require "glue code" that is difficult to read and tedious to write.

You should document any restrictions that exist on a method's parameters and enforce them with explicit checks at the beginning of the method body (except when the validity check would be expensive/impractical and the check is performed implicitly in the process of doing the computation) [Item 49]. Failure to validate parameters can result in a violation of *failure atomicity*. It is particularly important to check the validity of parameters that are not used by a method but stored for later use, which is often the case for constructors and static factories. For public and protected (i.e. *exported*) methods, use the Javadoc `@throws` tag to document the exception that will occur upon parameter violation (typically `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException`). The `Objects.requireNonNull(T obj)` method checks whether the specified object reference is null; if it isn't, it returns its argument, if it is, it throws a `NullPointerException`.[4] Use this, as there's no reason to perform null checks manually anymore. Note that you don't have to explicitly check whether an argument is null if you use that argument in another function call that will do it. If every method in a class is at risk of throwing a certain exception, you can document it in the class-level doc comment for the enclosing class to avoid the clutter of documenting it on every method individually. For non-public/unexported methods, you can check their parameters using *assertions* with the `assert` keyword, e.g. `assert a != null`. When assertions are enabled, they throw `AssertionError` when they fail, and if they are not enabled, they have no effect and essentially no cost. `AssertionError` extends `Error`, which itself extends `Throwable`, meaning `AssertionError` is an unchecked exception, and you should not try to catch or handle it. You enable assertions by passing the `-ea`/`-enableassertions` flag to the `java` command (or in your IDE's project configuration).

You must program defensively, assuming that clients of your class will do their best to destroy its invariants. One such attack is changing/updating a mutable field in an immutable class. To prevent this, for immutable classes and often for mutable ones as well, you should make a (defensive) copy of each mutable parameter passed to the constructor and store and use this, as well as have accessors (getters) return (defensive) copies of mutable fields (rather than returning the actual reference) [Item 50]. What's more, these copies should be made before checking the validity of the parameters, and the validity check should be performed on the copies rather than the originals, which protects the class against changes to the parameters from another thread during the *window of vulnerability* between the time of check and the time of copy, known as a time-of-check/time-of-use (TOCTOU) attack. Do not use `clone()` to make a defensive copy of a parameter whose type is subclassable by untrusted parties (though `clone()` is fine for returning copies from accessor methods).

A method is *overloaded* when a class contains multiple different methods with the same name but different signatures (i.e. a different number of parameters or different type of input parameters, or both). A method is *overridden* when a subclass contains a method declaration with the same signature as a method declaration in an ancestor (class or interface). Overriding is the norm and overloading is the exception. Be careful with overloading methods, because the choice of which overloading to invoke is made at compile time, and arguments are passed with their compile-time type rather than their runtime type [Item 52]. However, while selection among overloaded methods is static, confusingly, selection among overridden methods is dynamic (i.e. during runtime, based on the runtime type of the object on which the method is invoked), and the "most specific" overriding method always gets executed. A safe, conservative policy is never to export two overloadings with the same number of parameters, except when at least one corresponding formal parameter in each pair of overloadings has a "radically different" type in the sense that it is impossible to cast any non-null expression to both types (because then it is always clear which overloading will apply to any set of actual parameters). As an example, the `List<E>` interface violates this, because it has `remove(E)` (removes the specified element) and `remove(int)` (removes the element at the specified index), so if you have a `List<Integer>` and call `remove()` with an `int`, it doesn't autobox it to an `Integer` (which does happen with `Set<Integer>`, because it doesn't have a `remove(int)`), resulting on wildly different behaviour.[5] Do not overload methods to take different functional interfaces in the same argument position. You can always give methods different names instead of overloading them. As for constructors, you always have the option of exporting static factories instead. Constructors can't be overridden.

The last argument of a method may be declared as a variable arity, or *varargs*, parameter, by adding ... after the type, in which case the method becomes a varargs method. This allows one to pass a variable number of values (including zero) of the declared type, which will be available inside the method as an array. (`main()` can therefore also have `String... args` as parameter.) Varargs are simply implemented as arrays under the hood. This also means that every invocation of a varargs method causes an array allocation and initialization, which is a performance consideration.[6] If you want your method to take strictly one or more arguments, you shouldn't check the array length (because it's ugly and only fails at

[4]You can also add a `String` message as second parameter to let it throw a customized `NullPointerException`.

[5]Prior to Java 5, the `List` interface was "generified" and had a `remove(Object)` method in place of `remove(E)`, and the corresponding parameter types, `Object` and `int`, were radically different. Adding generics and autoboxing to the language essentially damaged the `List` interface.

[6]To avoid the performance penalty, you could have overloadings for e.g. one, two, or three parameters and a varargs version for more than three arguments.

runtime), but instead take two parameters: a regular one for the necessary first argument, and a varargs one for the optional others [Item 53].

For methods returning arrays or collections, return empty arrays or collections instead of null [Item 54]. If you're worried about the performance overhead of repeatedly allocating the empty container or array, you can return `Collections.emptyList()/emptyS` which returns the same immutable empty container (because immutable objects may be shared freely), or the same zero-length array by allocating it somewhere once.
The `Optional<T>` class represents an immutable container that can hold either a single non-null `T` reference ("present"; use static factory `Optional.of(result)`) or nothing at all ("empty"; use static factory `Optional.empty()`). It is essentially an immutable collection that can hold at most one element (it could have implemented `Collection<T>`). It should be used as return type for methods that conceptually return `T` but may be unable to do so under certain circumstances, and clients will have to perform special processing if no result is returned [Item 55]. This is more flexible and user-friendly than throwing an exception, and is less error-prone than one that returns null. Never return null from an `Optional`-returning method, which defeats the entire purpose. Passing null to `Optional.of(value)` makes it throw a `NullPointerException`. The `Optional.ofNullable(value)` method does accept null and returns an empty optional if a null is passed in. Optionals are similar to checked exceptions in that they force the user of an API to handle the case when no value is returned. You can `get()` an `Optional`'s value at risk of an `NoSuchElementException`. `orElse(T other)` returns the value if present, or the specified `other` if not. `orElseThrow(Supplier exceptionSupplier)` throws an exception created by the specified supplier if it's empty. `orElseGet(Supplier)` invokes the supplier and returns that result if the `Optional` is empty (a better name would have been `orElseCompute()`). `Optional` also has instance methods `filter()`, `map()`, `flatMap()`, and `ifPresent()`. There's also `isPresent()`, but that can often be replaced by some other methods for shorter and clearer code. Container types (including collections, maps, streams, arrays, and optionals) should not be wrapped in optionals (instead you should return empty containers; see the beginning of this paragraph). Optionals aren't appropriate for some performance-critical situations. For performance, you should never return an optional of the boxed primitive types `Integer`, `Long`, and `Double`, because those have two layers of boxing; instead, use `OptionalInt`, `OptionalLong`, and `OptionalDouble`. It is almost never appropriate to use an optional as a key, value, or element in a collection or array. Storing an optional in an instance field is also often a bad code smell, but not always.

4.1.3 Inheritance

Classes inherit with the `extends` keyword, and can reference its direct superclass with the `super` keyword. Unlike C++, classes can only inherit from one class (but they can implement multiple interfaces). If the superclass does not have a constructor without parameters, the subclass must specify in its constructor(s) what constructor of the superclass to use: `super(<arguments...>)`. The extending class can override its parent's methods, but they cannot have a more restrictive access level than the superclass. Inheritance is appropriate only when the subclass really "is-a" superclass. Inheriting from concrete (non-abstract) classes (not interfaces) across package boundaries is dangerous, because inheritance violates encapsulation (a subclass depends on the implementation details of its superclass); instead, favor *composition*, where you give your (wrapper) class a private field that references an instance of the existing class (i.e. the existing class becomes a component) [Item 18]. This is also known as the *decorator pattern*. Each method that would normally be implicitly inherited simply invokes the corresponding method in the component, called *forwarding*; these methods must be explicitly defined and are known as *forwarding methods*. Often, all the forwarding methods are contained in a separate forwarding class, and that forwarding class can be extended by several wrapper classes. Google's Guava library provides forwarding classes for all of the `Collection` interfaces.

You should either design and document for inheritance or prohibit it (by declaring your class final or ensuring that there are no accessible constructors) [Item 19]. For each exported method, the documentation must indicate which overridable (i.e. exported, non-final) methods the method invokes, in what sequence, and how the results of each invocation affect subsequent processing. It should do this in the "Implementation Requirements" section of the specification, which is generated by the Javadoc tag `@implSpec`. An unfortunate consequence of the fact that inheritance violates encapsulation is that this need for implementation details violates the dictum that good API documentation should describe what a given method does and not how. Also, constructors must not invoke overridable methods, directly or indirectly, because the superclass constructor runs before the subclass constructor, so the overriding method in the subclass will get invoked before the subclass constructor has run. It is generally not a good idea for a class designed for inheritance to implement either the `Cloneable` and `Serializable` interfaces because they place a substantial burden on programmers who extend the class. If you decide to anyways, make sure `clone()` and/or `readObject()` do not invoke overridable methods, because they behave a lot like constructors. Also, make `readResolve()` or `writeReplace()` protected rather than private. The only way to test a class designed for inheritance is to write subclasses.

4.1.4 Nested classes

Most classes are top-level, though you can have nested classes, which are placed inside another class and which may access the private members of the enclosing class. There are four kinds of nested classes:

- Member classes
  - static member classes; ordinary classes that happens to be declared inside another class and accessed as `OuterClass.InnerClass`. You need to make the nested class a public static class if you want it's public fields and methods to be accessible outside the parent class. Does not qualify as an inner class (all the following types are inner classes, however).
  - non-static member classes; very different from the static variant, even though they only differ by one keyword. Each instance of a non-static member class is implicitly associated with an enclosing instance of its containing class, and this association is established when the member class instance is created, and it cannot be modified afterwards.
- Anonymous classes; essentially function literals. Rather than being declared along with other members, these are simultaneously declared and instantiated at the point of use. Anonymous classes have enclosing instances if and only if they occur in a non-static context, but even then they cannot have any static members other than constant variables. Their syntax is similar to a constructor invocation; it consists of the `new` keyword followed by the name of an class/interface to extend/implement followed by a class body, e.g.:

  ```
  ParentClass myObject = new ParentClass() {
     // Definition of a new anonymous class extending ParentClass
  }
  ```

  Anonymous classes were adequate for representing function objects to pass as arguments to functions, but because of their verbosity, you should prefer lambdas [Item 42]. For example, we can pass an anonymous class as stand-in for the `Comparator` argument of `Collections.sort(List<T>, Comparator<? super T>)`:

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

- Lambda expressions, or lambdas; shorthand instances of *functional interfaces*, which are interfaces with a single abstract method (but they can have other default methods). Because `Comparator` is a functional interface, the above snippet can be written more concisely with lambdas in the following ways:

  ```
  Collections.sort(words, (s1, s2) -> Integer.compare(s1.length(), s2.length())); // Using a
      lambda
  Collections.sort(words, comparingInt(String::length)); // Using a comparator construction
      method
  words.sort(comparingInt(String::length)); // Using List's sort() method
  ```

  The types of the lambda (`Comparator<String>`), its parameters (both `String`), and its return value (`int`) are deduced from the context with *type inference*. The rules for type inference are complex, and sometimes the compiler can't infer types. Specify types in that case, or if their presence makes your program clearer. Because lambdas lack names or documentation, if a computation isn't self-explanatory or exceeds a few lines, don't put it in a lambda. Also, if it more succinct to use *method references* without losing readability, use those [Item 43].

  Anonymous classes are still useful for creating instances of an abstract classes (since lambdas are limited to functional interfaces), creating instances of interfaces with multiple abstract methods, and accessing the function object from within its body (since lambdas cannot obtain a reference to itself; `this` in a lambda refers to the enclosing instance – which is typically what you want – while `this` in an anonymous class refers to the anonymous class instance).

- Local classes; can be declared practically anywhere a local variable can be declared and obeys the same scoping rules. Like member classes, they have names and can be used repeatedly; like anonymous classes, they have enclosing instances only if they are defined in a non-static context, and they cannot contain static members; and like anonymous classes, they should be kept short so as not to harm readability.

Here is an overview. Favor static member classes over non-static member classes, because each instance of the latter will have a hidden extraneous reference to its enclosing instance, wasting time and space and can be bad for garbage collection [Item 24].

4.1.5 Immutable classes

Immutable classes are those whose instances cannot be modified. The boxed primitive types are immutable; performing arithmetic on them creates new instances, a pattern known as the functional approach (cq. the imperative/procedural approach, in which methods apply a procedure to their operand, causing its state to change). To make a class immutable:

- don't provide mutator methods that modify the object's externally visible state (however, you can have non-final fields in which you cache expensive computations the first time they are needed, called *lazy initialization*).
- ensure that the class can't be extended (because subclasses can compromise immutability) by either making the class final or making all of its constructors (package) private and adding public static factories; the latter is the most flexible because it allows the use of multiple package-private implementation classes.
- make all fields final and private.
- ensure exclusive access to any mutable components.

Classes should be immutable wherever and as much as possible [Item 17]. The benefits of immutability are manifold: they're simple and can be in exactly one state (the one they were created with), they're thread-safe and require no synchronization, they can be shared, reused and cached freely (reducing memory footprint and garbage collection costs), they can share their internals, they make great building blocks for other objects, and they provide failure atomicity for free. The major disadvantage is that they require a separate object for each distinct value, which can be costly in terms of performance and memory, especially if you perform a multistep operation that generates a new temporary object at every step. You can mitigate this with a mutable companion class. For example, the immutable `java.lang.String` class has a mutable `StringBuilder` companion class, with `append()` and `insert()` as principal operations, which are overloaded so as to accept data of any type and then convert that data to a string. `StringBuilder` can be used to efficiently built sequences of characters, which can finally be converted to an immutable `String` with `StringBuilder`'s `toString()`.

4.2 Interfaces

Interfaces are like abstract classes which contain no fields and usually define a number of abstract methods (without an actual implementation). A class implements an interface with the `implements` keyword (rather than `extends`). A class may implement multiple interfaces, contrary to inheritance. When a class implements an interface, the interface serves as a type that can be used to refer to instances of the class, and it says something about what a client can do with instances of the class. It is inappropriate to define an interface for any other purpose [Item 22], like *constant interfaces* that contain only constants. Interfaces can extend other interfaces. Interfaces may have `default` methods (with a default implementation), `private` methods (used by default methods) or `static` methods. All of a class's methods that come from an interface must be declared `public` in the class. Default methods were introduced to allow the addition of methods to an existing interface without breaking its implementing classes. However, in that case, default methods are "injected" into existing implementations without the knowledge or consent of their implementors, so you should design interfaces for posterity and avoid retroactively adding default methods [Item 21].

Even though they're similar (especially since interfaces can have default methods), you should prefer interfaces to abstract classes [Item 20], for a number of reasons: existing classes can easily implement a new interface while they can only inherit from one (abstract) class, interfaces are not hierarchical (making them ideal for e.g. *mixins*, a sort of an "add-on"/"plugin" type that a class can implement in addition to its "primary type"). You can combine the advantages of interfaces and abstract classes with the *template method pattern*, in which an abstract skeletal implementation class complements an interface and this skeletal implementation class implements the remaining non-primitive interface methods [?] atop the primitive interface methods. Extending a skeletal implementation takes most of the work out of implementing an interface. By convention, skeletal implementation classes are called `Abstract<interfaceName>`, e.g. the Collections Framework provides a skeletal implementation to go along with each main collection interface: `AbstractCollection`, `AbstractSet`, `AbstractList`, and `AbstractMap`. A *simple implementation* is like a skeletal implementation in that it implements an interface and is designed for inheritance, but it differs in that it isn't abstract: it is the simplest possible working implementation.

Marker interfaces are interfaces that contains no (new) method declarations but merely designate (or "mark") a class that implements the interface as having some property. They define a type, with compile-time error detection as goal. An example is the `Serializable` interface, which indicates that its instances can be written to an `ObjectOutputStream` (or "serialized").

Functional interfaces are interfaces with a single abstract method (but they may have other default methods). They represent function types and their instances are *function objects* and represent concrete functions or actions. You can concisely create instances using lambdas. You shouldn't define your own functional interfaces unnecessarily; if one of the many standard functional interfaces in `java.util.function` does the job, you should generally instead use that, because it will make your API more readable, and many of the standard functional interfaces provide useful default methods [Item 44]. You should only consider violating this rule if the custom functional interface will be commonly used and could benefit from a descriptive name, or if it has a strong contract associated with it, or if it would benefit from custom default methods. `java.util.function` has 43 interfaces, but they can be derived from six basic ones, which operate on object reference types:

Interface	Function signature	Example	Description
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`	Result and argument types are the same
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`	Result and argument types are the same
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`	Takes an argument and returns a boolean
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`	Argument and return types differ
`Supplier<T>`	`T get()`	`Instant::now`	Takes no arguments and returns a value
`Consumer<T>`	`void accept(T t)`	`System.out::println`	Takes an argument and returns nothing

Variables, fields, parameters, and return values should all be declared using interface types rather than class types, if an appropriate interface type exists and if the program doesn't use methods of the implementing class not found in the interface [Item 64]. In such cases, the only time you need to refer to an object's class is when you're creating it with a constructor. If you write

```
Set<Integer> set = new LinkedHashSet<>();
```

rather than declaring it with `LinkedHashSet`, switching implementations is much easier because the surrounding code is unaware of the implementation type. If there is no appropriate interface, just use the least specific class in the class hierarchy that provides the required functionality.

4.3 Method references

Method references are a special type of lambda expression which are often more concise because they reference existing methods. The syntax is `ClassName::methodName`, e.g. instead of the lambda `(a, b)-> Integer::sum(a, b)` you can simply use the method reference `Integer::sum`. There are five types of method references:

Type	Description	Example	Lambda equivalent
Static	Most mehod references are of this type.	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	The receiving object is specified in the method reference. Similarly to static references, the function object takes the same arguments as the referenced method.	`Instant.now()::isAfter`	`Instant then = Instant.now(); t -> then.isAfter(t)`
Unbound	The receiving object is specified when the function object is applied, via an additional parameter before the method's declared parameters.	`String::toLowerCase`	`str -> str.toLowerCase()`
Class constructor	Serve as factory objects.	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>`
Array constructor	Serve as factory objects.	`int[]::new`	`len -> new int[len]`

4.4 Enums

Enum types (enumerated types), or enums, are special types whose legal values consist of a fixed set of enumerable constants, e.g. the seasons of the year:

```
public enum Season { SPRING, SUMMER, FALL, WINTER }
```

They're often used in switch statements. Contrary to C++ and C#, where enums are essentially `int` values, an enum in Java is a fully fledged class that exports each enumeration constant as a constant (i.e. public static final) field (written in uppercase letters). All advice for classes naturally hold for enums. They have no constructors and are therefore effectively final, cannot be instantiated or extended, and cannot extend other classes. They can have fields and methods and implement interfaces (with which you can emulate extensible interfaces allowing clients to write their own enums that implement the interface [Item 38]). They provide high-quality implementations of all the `Object` methods, they implement `Comparable` and `Serializable`, and their serialized form is designed to withstand most changes to the enum type. They are a generalization of singletons (in the form where a class has a constant field holding the singular instance), which are essentially single-element enums. Their `toString()` instance method translates a value into printable strings (which you van override). The static `values()` function returns an array with the enums values (in order of declaration). Enum types have an automatically generated `valueOf(String)` method that translates a constant's name into the constant itself. Use enums instead of `int` (or even worse, `String`) constants, because enums provide compile-time type safety and are more readable and powerful [Item 34]. It is not necessary that the set of constants in an enum type stay fixed for all time. The enum feature was specifically designed to allow for binary compatible evolution of enum types.

To associate data with enum constants, declare enum *instance fields* and write a constructor that takes the data and stores it in the fields, and then you can pass the data as parameters that are passed to its constructor between parentheses after the constant declaration:

```
public enum Season {
   SPRING(20, "easter"),
   SUMMER(30, "liberation day"),
   FALL(10, "thanksgiving"),
   WINTER(0, "christmas");

   private final int typicalTemperature;
   private final String typicalHoliday;

   Season(double typicalTemperature, String typicalHoliday) {
      this.typicalTemperature = typicalTemperature;
      this.typicalHoliday = typicalHoliday;
   }

   public int typicalTemperature() { return typicalTemperature; }
   public String typicalHoliday() { return typicalHoliday; }
}

int summerTemperature = Season.SUMMER.typicalTemperature();
```

All enums have an `ordinal()` method, which returns the declared position of each enum constant in its type as an `int`. Never derive a value associated with an enum from its ordinal; store it in an instance field instead [Item 35]. In fact, its best to avoid `ordinal()` entirely.

Each enum constant declaration can have its own class body, called a *constant-specific class body*, which are treated like anonymous classes extending the enum class. In rare cases, you might need different behaviour/code for different constants. This can be used in *constant-specific method implementations*, in which you declare an abstract method in the enum type and override it with a concrete method for each constant in its constant-specific class body:

```
public enum CalculatorOperation {
   PLUS { public double apply(double x, double y) { return x + y; } },
   MINUS { public double apply(double x, double y) { return x - y; } },
   TIMES { public double apply(double x, double y) { return x * y; } },
   DIVIDE { public double apply(double x, double y) { return x / y; } };

   public abstract double apply(double x, double y);
}
```

However, the above can be written much more concisely by passing a lambda to its constructor, which stores it in an instance field.

```
public enum Operation {
   PLUS((x, y) -> x + y),
   MINUS((x, y) -> x - y),
   TIMES((x, y) -> x * y),
   DIVIDE((x, y) -> x / y);

   private final DoubleBinaryOperator op; // DoubleBinaryOperator is one of many predefined
       functional interfaces, with applyAsDouble() as its only function

   Operation(DoubleBinaryOperator op) {
      this.op = op;
   }

   public double apply(double x, double y) {
      return op.applyAsDouble(x, y);
   }
}
```

You can combine constant-specific class body with constant-specific data, in which case first the parentheses with data is written and then the constant-specific class body. The above can result in a lot of boilerplate, since you can't readily share code between enum constants.

When you want to combine several enum constants, such as when passing flags to a method, a traditional and intuitive way is by using bit fields, where each bit represents an enum constant. However, you should avoid bit fields and instead use `java.util.EnumSet`s [Item 36], which implements the `Set` interface, providing richness, type safety, and interoperability while still being represented as a bit vector:

```
public class Text {
   public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

   public void applyStyles(Set<Style> styles) { ... } // Any Set could be passed in, but EnumSet is
       clearly best
}

text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

`EnumMap`s are, similarly, the enum-oriented implementation of the `Map` interface, mapping enum constants of some enum type to values. The `EnumMap` constructor takes the `Class` object of the key type (e.g. `EnumType.class`), which is a bounded type token providing runtime generic type information (the bound of the type token is `T extends Enum<T>`, i.e. `T` should be an enum). Instead of using ordinals to index into arrays, use `EnumMap`s [Item 37]. `Class<T>` has a method `getEnumConstant()`, which returns the type's constant as a `T[]` if `T` is an enum, and null otherwise.

4.5 `java.lang.Object`

`java.lang.Object` is Java's top type; the superclass to every class that doesn't declare a parent. All of its non-final methods (`equals()`, `hashCode()`, `toString()`, `clone()`, and `finalize()`) have explicit general contracts because they are designed to be overridden. The `==` operator compares references/addresses, while the `equals()` methods compare values/contents. Don't override `equals()` unless you have to. If a class does not override `equals()`, then by default it uses the `equals(Object o)` method of the closest parent class that does, in which case each instance of the class is equal only to itself. When overriding `equals()` you must ensure it is: reflexive, symmetric, transitive, consistent over time, and null must return `false`; violating this can result in nasty bugs, since containers like `HashSet` rely on `equals()`. There is no way to extend an instantiable class and add a value component while preserving the equals contract, unless you're willing to forgo the benefits of object-oriented abstraction [Item 10]. Rather than extending, you can use composition (where you write an unrelated class containing an instance of the first class and provide a "view" method that returns the contained instance). A good `equals()` body has, in order:

1. `==` operator check as performance optimization.
2. `instanceof` operator check for the class in which the method occurs (or maybe a shared interface).
3. argument cast to the correct type (possible because of the previous check).
4. check whether all relevant fields match.

The parameter type of `equals()` must always be `Object`, or else you overload instead of override. Always override `hashCode()` when you override `equals()`, or your class will violate the general contract for `hashCode()`, which will prevent it from functioning properly in collections such as `HashMap` and `HashSet` [Item 11]. The contract mainly states (besides other things) that equal objects must have equal hash codes. This entails that you must exclude any fields that are not used in `equals()`. Good `hashCode()` implementations return different hash codes for different instances, but that's not required.

A class that implements `Comparator<T>` represents a comparison function, which can be passed to e.g. `Arrays.sort(arr, comp)`/`Collections.sort(col, comp)`. `Comparator<T>` is a functional interface that has a `compare(T o1, T o2)` abstract method, a default `reversed()` instance method that returns a comparator that imposes the reverse ordering of this comparator. It also has a set of comparator construction methods, for example, for `int`s, there are `comparingInt()` and `thenComparingInt()`. `comparingInt()` is a static method that takes a key extractor function that maps an object reference to a key of type `int` and returns a comparator that orders instances according to that key. `thenComparingInt()` also takes a key extractor function, but is an instance method and is instead called on a `Comparator`, returning a comparator that first applies that comparator and then uses the extracted key to break ties. Both methods can be strung together – `comparingInt()` followed by zero or more `thenComparingInt()` – which can be strung together into a `Comparable`-implementing class's static final `COMPARATOR`, which in turn can be used in a much less verbose implementation of that class's `compareTo()`.

compareTo(T o) is similar to equals(), but should return a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object. It is not declared in Object, but is the sole method in the Comparable<T> interface (not the be confused with Comparator<T>, discussed above). T is the type to which elements of the type implementing Comparable<T> can be compared. In practice, nearly all types can be compared only to elements of their own type, so you'd get e.g. String implements Comparable<String>. Implementing it indicates that instances of the implementing class have a natural ordering, which allows them to be sorted with Arrays.sort(arr)/Collections.sort(col) or used in a TreeSet/TreeMap. Implementing Comparable is recommended [Item 14]. A class that implements Comparable does not need a Comparator anymore. compareTo() doesn't have to work between different types and is permitted to throw ClassCastException if types are different, which it usually does. The same un-extensibility caveat and the same workaround as with equals() applies here. Because Comparable<T> is parameterized, compareTo(T obj) is statically typed, so you don't need to type check or cast its argument; that's all caught during compilation. A compareTo() implementation recursively calls compareTo() on its object reference fields. Starting with the most significant field, for each field, only work your way to the nextmost significant field in a nested if-block if the field is equal (else return the result), until you find an unequal field or compare the least significant field. Use of the relational operators < and > is verbose, error-prone and not recommended.

Object's implementation of toString() returns <ClassName>@<HashCode>. Always override toString() and return a concise, informative, self-explanatory representation [Item 12]. toString() is automatically invoked when an object is passed to println(), printf(), the string concatenation operator, assert, or is printed by a debugger.

The assignment operator = copies references. To copy an object itself, use the clone() method. Override clone() judiciously [Item 13]. The Cloneable interface is implemented by a class to indicate to Object.clone() that it is legal for that method to make a field-for-field copy of instances of that class. If a class doesn't implement Cloneable, Object.clone() throws CloneNotSupportedException. Annoyingly enough, Cloneable does not itself have a clone() method.[7] By convention, classes that implement this interface should override Object.clone() (which is protected) with a properly functioning public method. The general contract for clone() is very weak and vague: "Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, x.clone()!= x and x.clone().getClass()== x.getClass() and x.clone().equals(x), but these are not absolute requirements. By convention, the object returned by this method should be obtained by calling super.clone(), and the returned object should be independent of the object being cloned, so it may be necessary to modify one or more fields of the object returned by super.clone() before returning it." Java supports *covariant return types*, which means that an overriding method's return type can be a subclass of the overridden method's return type, which eliminates the need for casting in the client. Because of this, overridden clone()'s may return the type of their encompassing class. Even though Object's clone() is declared to throw CloneNotSupportedException, public clone methods need and should not do this, as methods that don't throw checked exceptions are easier to use. If your class does not have reference fields, this is what overriding clone() should look like:

```
@Override public PhoneNumber clone() {
  try {
    return (PhoneNumber) super.clone(); // Returns a field-by-field copy of this PhoneNumber
        instance, somehow
  } catch (CloneNotSupportedException e) {
    throw new AssertionError(); // Can't happen, but this try-catch boilerplate is necessary
        because annoyingly Object declares its clone() to throw CloneNotSupportedException, which
        is checked
  }
}
```

However, if an object contains fields that refer to mutable objects, these must be recursively cloned as well (and that may not even be enough for a deep copy). clone() functions as a constructor, and like any constructor you must ensure that it does no harm to the original object and that it properly establishes invariants on the clone. Like serialization, the Cloneable architecture is incompatible with normal use of final fields referring to mutable objects (except in cases where the mutable objects may be safely shared between an object and its clone). All in all, a better approach to object copying is to provide a copy constructor or copy factory, but sometimes you have no choice, such as when you extend a class that already implements Cloneable. However, arrays are best copied with clone().

[7] This use of interfaces is not one to be emulated. Normally, implementing an interface says something about what a class can do for its clients, but here it modifies the behavior of a protected method of a superclass. The Cloneable interface and Object.clone() are an obvious case of object-oriented design gone wrong.

4.6 Generics

Generics are Java's version of C++'s templates. A class or interface whose declaration has one or more *(formal) type parameters* (e.g. `List<E>`) is a *generic* class or interface, and they're collectively knows as *generic types*. A *parameterized type* is a generic type instantiated with an *actual type parameter* (e.g. `List<String>`). Each generic type defines a *raw type*, which is the name of the generic type used without any accompanying type parameters in angle brackets ((e.g. `List`). Raw types behave as if all of the generic type information were erased from the type declaration. They exist for compatibility with pre-generics code. Don't use them, because the compiler can't catch type errors, and you instead risk `ClassCastException`s at runtime [Item 26].[8] It is fine to use types parameterized to allow insertion of arbitrary objects (e.g. `List<Object>`), because you still retain type safety. For example, `List<String>` is a subtype of the raw type `List`, but not of `List<Object>` (so you can pass a `List<String>` to a parameter of type `List` but not to a parameter of type `List<Object>`).
When using generic `Collection`s, the compiler inserts invisible casts (to the actual type parameter) for you when retrieving elements from collections and guarantees that they won't fail. For this reason, when designing new types, favor generic types if it would otherwise require explicit casts (from type `Object`) in client code [Item 29]. You can also *generify*/parameterize a type after the fact without harming clients of the original non-generic version. Just as classes, you should parameterize methods when clients would otherwise have to put explicit casts on input parameters and/or return values [Item 30]. The type parameter list of a method goes before a method's return type in its signature (so that return types can use the parameter(s)).

Type parameter names usually consist of a single letter; common examples are `T` for an arbitrary type, `E` for the element type of a collection, `K` and `V` for the key and value types of a map, `X` for an exception, and `R` for the return type of a function.

If you want to use a generic type, but you don't know or care what the actual type parameter is, you can use a question mark instead to get a *wildcard type*. A *bounded type parameter* is a type parameter followed by `extends TypeName` (in which case the actual type parameter is restricted to be a subtype of the specified type) or `super TypeName` (for supertypes of the specified type), else it is *unbounded*.[9] This allows the type at hand and its clients to take advantage of methods defined by the extended type without the need for explicit casting or the risk of a `ClassCastException`. The subtype relation is defined so that every type is a subtype of itself. A *recursive type bound* happens when a type parameter is bounded by some expression involving that type parameter itself, which most often happens in connection with the `Comparable<T>` interface, e.g.:

```
public static <E extends Comparable<E>> E max(Collection<E> c); // A recursive type bound to
    express mutual comparability. "Any type E that can be compared to itself"
```

Generics are implemented by *erasure*: they enforce their type constraints only at compile time and discard (erase) their element type information at runtime (e.g. the runtime type of a `List<Integer>` instance is simply `List`). This is what allowed generic types to be used freely with legacy code. Another term for it is that generics are non-reified. A *non-reifiable* type is one whose runtime representation contains less information than its compile-time representation. The only parameterized types that are reifiable are unbounded wildcard types such as `List<?>`.
Generic types are *invariant*: for any two distinct types `TypeA` and `TypeB`, `List<TypeA>` is neither a subtype nor a supertype of `List<TypeB>`, e.g. `List<String>` is not a subtype of `List<Object>`, because `List<String>` can't do everything a `List<Object>` can (namely store anything that isn't a `String`). Wildcards are meant to alleviate this restriction; `Generic<?>` is a supertype of all parameterizations of the generic type `Generic`. They can be used for maximum API flexibility, by using bounded wildcard types on generic/parameterized input parameters (of a method) that represent producers, in which case you use `extends` (e.g. a source `Iterable<? extends E>` from which to obtain elements of some subtype of your class's type parameter `E`), or consumers, in which case you use `super` (e.g. a destination `Collection<? super E>` of some supertype of your class's type parameter `E`) [Item 31]. Do not use bounded wildcard types as return types, as that would force clients to use wildcard types in client code (reducing flexibility). `Comparable`s/`Comparator`s are always consumers, so you should generally use `Comparable<? super T>`/`Comparator<? super T>` in preference to `Comparable<T>`/`Comparator<T>`, since both of them consume `T` instances (and produces integers indicating order rela-

[8]Exceptions are the use of raw types in class literals (`List.class`, `String[].class`, and `int.class` are all legal, but `List<String>.class` and `List<?>.class` are not) and in conjunction with the `instanceof` operator (whose behaviour is unaffected by unbounded wildcard types), e.g.

```
if (o instanceof Set) { // Raw type
   Set<?> s = (Set<?>) o; // Wildcard type
   ...
}
```

[9]`TypeName` can be an interface, so "`extends`" can also mean "implements"; there isn't a separate keyword for that.

tions). For example:

```
public static <T extends Comparable<T>> T max(List<T> list) // Return the max element of a list
public static <T extends Comparable<? super T>> T max(List<? extends T> list); // Revised
    declaration that uses wildcard types
```

The second declaration accepts `List<ScheduledFuture<?>>` while the first doesn't. `ScheduledFuture` does not directly implement `Comparable<ScheduledFuture>`, but instead implements `Delayed` which does implement `Comparable<Delayed>`, so a `ScheduledFuture` instance is comparable to any `Delayed` instance. The wildcard is required to support types that do not implement `Comparable` directly but extend a type that does. The `list` argument is a producer, and should therefore use `extends`, because the list should be able to contain any subtype of the type parameter.

As a rule, if a type parameter appears only once in a method declaration, replace it with a wildcard, which gets rid of the type parameter and yields simpler APIs, but this may entail writing a helper function (with the more convoluted parameterized signature) to capture the wildcard type (as wildcards only allow nulls as values), e.g.:

```
public static void swap(List<?> list, int i, int j) {
   swapHelper(list, i, j);
}

private static <E> void swapHelper(List<E> list, int i, int j) {
   list.set(i, list.set(j, list.get(i)));
}
```

When you program with generics, you will see many unchecked compiler warnings. Eliminate every unchecked warning that you can, because then you know that your code is typesafe (i.e. no `ClassCastException`s) [Item 27]. If you can't eliminate a warning but can prove that the code is typesafe, then (and only then) suppress the warning with an `@SuppressWarnings("unchecked")` annotation. Always use the `@SuppressWarnings` annotation on the smallest scope possible, and always add a comment explaining why it is safe to do so.

Even though generics and varargs parameters were both introduced in Java 5, they do not play along well. Varargs parameters are actually arrays[10], and arrays and generics have different type rules. Even though it is illegal to create a generic array explicitly, it is legal to declare a method with a generic varargs parameter, because such methods can be very useful in practice (and so the language designers opted to live with this inconsistency). However, you should be careful [Item 32], because this combination can cause *heap pollution*, which occurs when a variable of a parameterized type refers to an object that is not of that type, which in turn can cause `ClassCastException`, e.g.:

```
static void dangerous(List<String>... stringLists) {
   List<Integer> intList = List.of(42);
   Object[] objects = stringLists;
   objects[0] = intList; // Heap pollution
   String s = stringLists[0].get(0); // ClassCastException due to compiler generated cast
}
```

A method with generic varargs parameter is typesafe if the method doesn't store/write anything into the underlying array and doesn't allow a reference to the array to escape to other code, either by returning a reference or passing a reference to another method (though it's fine if that method is another varargs method that is correctly annotated with `@SafeVarargs` or is a non-varargs method that merely computes some function of the contents of the array). If these conditions are met, always use `@SafeVarargs` to suppress client warnings automatically, which represents a promise by the author of a method that it is typesafe.

4.7 Arrays

Arrays are reference types, or objects, and are created at runtime, just like class instances. Array length is defined at creation and cannot be changed. The length can be queried in constant time with `array.length`, because Java stores it. Unlike C++, sub-arrays can vary in length, i.e. multi-dimensional arrays are not bound to be rectangular. Nonzero-length arrays are always mutable. Arrays can be created in any of the following ways:

```
int[] numbers = new int[5]; // Initializes all elements to the default value (boolean: false, int:
    0, double: 0.0, etc.)
int[] numbers = new int[] {20, 1, 42, 15, 34}; // Initialization; long version
int[] numbers = {20, 1, 42, 15, 34}; // Initialization; short version
```

[10] *Leaky abstraction* happens when an abstraction leaks details that it is supposed to abstract away. The array that is created to hold varargs parameters should be an implementation detail, but is visible.

```
System.out.println(Arrays.toString(array)); // Print array (else you get gibberish like [I@5fdef03a)

int[][] numbers = new int[3][3]; // Multidimensional array declaration
int[][] numbers = {{2, 3, 2}, {1, 2, 6}, {2, 4, 5}}; Multidimensional array initialization
System.out.println(Arrays.deepToString(array)); // Print multidimensional array (else it still
    prints the inner arrays as gibberish)

int array[][] = new int[2][]; // Jagged/ragged/skewed multidimensional array declaration (different
    row lengths)
array[0] = new int[8];
array[1] = new int[3];

String[] = new String[5]; // Arrays of reference type are not initialized in the same way, not with
    a constructor
```

`java.util.Arrays` is a JCL utility class containing various static methods for manipulating arrays (e.g. sorting and searching) and a static factory that allows arrays to be viewed as lists. Copy-assigning arrays (`int[] listB = listA`) makes a shallow copy, i.e. the array reference is copied and both variables point to the same array and the elements still refer to the same things. Use `int[] listB = listA.clone()` or `int[] listB = Arrays.copyOf(listA, listA.length);` to make a deep copy.

Arrays are *covariant* (if `SubClass` is a subtype of `SuperClass`, then `SubClass[]` is a subtype of `SuperClass[]`) and *reified*[11][12] (they know and enforce their element type at runtime). These are disadvantages, as it can lead to runtime exceptions like `ArrayStoreException`, and for these reasons you should prefer `List`s [Item 28]. Arrays provide runtime type safety but not compile-time type safety, while generics (such as `List`s) provide compile-time safety but not runtime safety (because generics are implemented with erasure). It is illegal to create an array of a generic type, a parameterized type, or a type parameter, because generic arrays aren't typesafe; they'll lead to *generic array creation* errors at runtime (but you can cast them after creating them, i.e. `E[] elems = (E[])new Object[]`). It is legal, though rarely useful, to create arrays of unbounded wildcard types. You can use raw types, but this breaks multiple rules and is not recommended:

```
HashSet<Integer>[] sets = new HashSet[5];
for (int i = 0; i < 5; i++) {
   sets[i] = new HashSet<Integer>();
}
```

4.8 Collections

The JCL supports a variety of data structures in the `java.util` package, organized into a big hierarchy. `Collection` is the root interface for all of these. A `Collection` represents a group of objects called *elements*, and is type parameterized by one or more element types. The `Collection` interface isn't directly implemented by anything, but instead the JCL has subinterfaces like `List` and `Set` that are implemented. Since interfaces can't declare constructors there is no way to enforce this, but by convention all general-purpose `Collection` implementation have two constructors: a default constructor that creates an empty collection and a constructor that takes another collection and creates a new one with the same elements, which is very convenient. For example, because a collection is an object, a variable of that type is actually a reference, and copy assignment (`ArrayList<Integer> listA = listB`) will only copy the reference. A copy constructor (`ArrayList<Integer> listA = new ArrayList<>(listB)`) makes a deep copy, i.e. copies the `ArrayList` object and copies all element values (not just the references). Collections can only store elements of reference types, so no primitives, or else you get a compile-time error; a fundamental limitation of Java's generic type system. You have to use the boxed primitive types. The most common data structures are:

- `ArrayList<E>`: dynamic array, and the most straightforward implementation of the `List` interface. Add elements to the end with `add(E element)`. You can't index with `[]` but have to use `get(int index)`. You assign to an index with `set(int index, E element)`. Allows nulls. `List.of()` is a (default) static method that returns an immutable list of the specified elements that doesn't allow nulls, while `Arrays.toList()` is a static method that returns a fixed-size "wrapper"/view list (i.e. no data is copied) backed by the supplied array (so adding and removing elements throws an `UnsupportedOperationException`, but you can change elements of the list with `set()`, which changes the elements of the underlying array) that does allow null. Both can be used to inline initialize an `ArrayList` (and any other `List`) with the copy constructor.
- `Deque<E>`: double-ended queue interface, which you can use as a (LIFO) stack with `push()`/`addFirst()` and

[11] reify, verb, formal: make (something abstract) more concrete or real. "These instincts are, in man, reified as verbal constructs."

[12] Cq. generics, which are non-reified, because they enforce their type constraints only at compile time and discard (erase) their element type information at runtime.

`pop()`/`removeFirst()`/`remove()` for adding/removing on top/at the front, or as a (FIFO) queue with `add()`/`addLast()` (at the end) and `remove()`/`removeFirst()`/`pop()` (at the front). Implemented by `ArrayDeque`, which doesn't allow nulls, and `LinkedList`, which does allows nulls and also supports adding at and getting arbitrary indices. `offer()` (from the `Queue` parent interface) methods insert an element if possible, otherwise returning `false`, is designed for use when failure is a normal (e.g. in fixed-capacity queues) and differs from `add()` methods, which can fail to add an element only by throwing an unchecked exception. Similarly, `remove()` throws `NoSuchElementException` when the queue is empty, while `poll()` returns null.

- `TreeSet<E>`/`TreeMap<K,V>`: binary search tree (BST) based sets and maps, i.e. the items are unique and sorted. They define `floor()`/`floorKey()` and `ceiling()`/`ceilingKey()` for finding entries that are closest to the input.
- `HashSet<E>`/`HashMap<K,V>`: hash table based sets and maps. Use `add(E e)`/`get()`/`contains()`/`remove(E e)` for `HashSet` and `put(K k, V v)`/`get(K key)`/`containsKey(K k`/`remove(K k)` for `HashMap`. Not sorted.
- `PriorityQueue<E>`: priority queue/heap data structure. Min heap by default; use `new PriorityQueue<>(Comparator.reverseOrd` for a max heap. Supports `add()`, `remove()` and `peek()`. The `Map` interface has a convenient `merge()` method (so `TreeMap` can also make use of it), which simply inserts the given value if the key is not present, and else applies the given function (method reference or lambda) to the current value and the given value and overwrites the current value with the result, e.g. `map.merge(key, 1, Integer::sum)` for incrementing by one if the key is present. The `Map` interface also has the similar `computeIfAbsent()` method, which looks up the given key and simply returns its associated value if it's present, and else the method computes a value by applying the given function object to the key and stores and returns this value. The method simplifies the implementation of maps that associate multiple values with each key, e.g. `groups.computeIfAbsent(alphabetize(word), (unused)-> new TreeSet<>()).add(word)`.

Here is another nice overview.

When writing a `Collection` implementation, you can extend from the skeletal `AbstractCollection`. For an unmodifiable collection, you need implement only `contains()` and `size()`. For a modifiable collection, you need to additionally override `add()` (which otherwise throws an `UnsupportedOperationException`), and the `Iterator` returned by `iterator()` must implement `remove()`.

`Collection`s are limited to a fixed number of types per container, often one or two. For an arbitrary number of types, you can parameterize the container's key rather than the container itself, which is called a *typesafe heterogeneous container* [Item 33]. One way of doing this is by using `Class` objects as keys. That is, the container has a `HashMap` with `Class<?>` as key type and `Object` as value type.[13] This allows storing and retrieving instances of arbitrarily many types, and the generic type system is used to guarantee that the type of the value agrees with its key. When a class literal is passed among methods to communicate both compile-time and runtime type information (such as when putting and getting items from the `HashMap`), it is called a *type token*. You can also use a custom key type, e.g. a `DatabaseRow` type representing a database row (the container), and a generic type `Column<T>` as its key.

The `Collections` class is a utility class with helper functions. It contains immutable empty containers `EMPTY_LIST`, `EMPTY_SET`, and `EMPTY_MAP`, as well a other helpful functions.

Java doesn't have a built in `Tuple` or `Pair` object, so if you need to store a list of pairs, consider doing in in two lists with the same layout.

5 Exceptions

try-catch-finally blocks looks like this:

```
try {
   methodThrowingExceptions(); // Statements that may throw exceptions
} catch (IOException | IllegalArgumentException ex) { // Informally known as multi-catch
   reportException(ex); // Both IOException and IllegalArgumentException will be caught and handled
       here, after which the finally-block will execute if it's present, followed by the code below
       it. Other exceptions are thrown further up the call stack
} finally {
   freeResources(); // Always executed after the try or catch blocks. Useful for providing clean-up
       code that is guaranteed to always be executed
}
```

[13] You might think that you couldn't put anything in it because of the unbounded wildcard type, but it's not the type of the map that's a wildcard type but the type of its key. This means that every key can have a different parameterized type: one can be `Class<String>`, the next `Class<Integer>`, and so on. That's where the heterogeneity comes from.

but you should prefer try-with-resources blocks when working with resources that should be closed (because it gets messy with multiple resources and because finally blocks can still throw) [Item 9]:

```
FileOutputStream fos = new FileOutputStream("filename"); // Resources can also be declared before
    they're used in a try-with-resources block
try (fos; XMLEncoder xEnc = new XMLEncoder(fos)) { // Initialization of one or more resources that
    are released automatically when the try block execution is finished.
   xEnc.writeObject(object);
} catch (IOException ex) {
   Logger.getLogger(Serializer.class.getName()).log(Level.SEVERE, null, ex);
}
```

Resources must implement `AutoClosable`, and any class that does can be used as a resource. You'll often see only the try-with-resources block, because a finally block is no longer needed.

You can throw exceptions with the `throw` keyword: `throw new NullPointerException();`.

`java.lang.Throwable` is supertype of everything that can be thrown or caught. A `Throwable` contains a snapshot of the execution stack of its thread at the time it was created and may contain a message string that gives more information about the error. Its two direct subclasses are:

- `Exception` contains *checked exceptions*, which are checked at compile time, and client code that calls methods throwing such exceptions are forced to either handle the exception in a `catch`-clause or it must propagate it outward and specify the type using the `throws` clause. This enhances reliability. Use checked exceptions for conditions from which the caller can reasonably be expected to recover [Item 70]. A *fully checked exception* is a checked exception where all its child classes are also checked, while a *partially checked exception* is a checked exception where some of its child classes are unchecked. `Exception`s are full-fledged objects on which arbitrary methods can be defined, with the primary goal to catch the exception with additional information.

  Don't overuse checked exceptions, because they do place burden on the user of the API and checked-exception-throwing methods can't be used directly in streams [Item 71]. Unless the exceptional condition cannot be prevented by proper use of the API and the programmer using the API can take some useful action if it happens, an unchecked exception is more appropriate. It is especially important to consider whether a method should throw a checked exception if it's gonna be the first or only one, because from then on the method has to be wrapped in a `try` block and can't be used directly in streams. The easiest way to eliminate a checked exception is to return an `Optional`, but then the method can't return any additional information on why it can't compute the result. You can also break up the checked-exception throwing method into a "state-dependent" method that throws an unchecked exception (e.g. `Iterator`'s `next()`) and a "state-testing" method (e.g. `hasNext()`) indicating whether the preconditions are satisfied.

  All exceptions under `Exception` are checked, except those in subclass:

  – `RuntimeException` contains *unchecked exceptions*, which are not checked at compile time and are not required to – and generally shouldn't – be caught or declared in a `throws` clause.[14] Recovery is often impossible and continued execution would do more harm than good. *Runtime exceptions* should be used to indicate programming errors, most of which violate precondition violations (e.g. `IndexOutOfBoundsException`).

- `Error` contains the other other kind of unchecked throwable: *errors*. There is a strong convention that errors are reserved for use by the JVM to indicate resource deficiencies, invariant failures, or other conditions that make it impossible to continue execution. For this reason, it's best not to throw `Error` or any subclass (except `AssertionError`) or implement any new `Error` subclasses.

All of the unchecked throwables you implement should subclass (directly or indirectly) `RuntimeException`. However, you should favor the use of standard exceptions [Item 72]. The most commonly reused exceptions are:

- `IllegalArgumentException`; non-null parameter value is inappropriate
- `IllegalStateException`; receiving object's state is inappropriate for method invocation
- `NullPointerException`; parameter value is null where prohibited
- `IndexOutOfBoundsException`; index parameter value is out of range
- `ConcurrentModificationException`; concurrent modification of an object has been detected where it is prohibited
- `UnsupportedOperationException`; object does not support method. Rare, because most objects support all of their

[14]In C++, all exceptions are unchecked, where it is up to the programmers to specify or catch the exceptions.

methods, but its used by classes that fail to implement one or more optional operations defined by an interface they implement, e.g. an append-only `List` implementation where someone would delete an element.

Occasions for use in the table above do not appear to be mutually exclusive. Throw `IllegalStateException` if no argument values would have worked, otherwise throw `IllegalArgumentException`.

It is possible to define a `Throwable` that is not a subclass of `Exception`, `RuntimeException`, or `Error` – the JLS doesn't address such throwables directly but specifies implicitly that they behave as ordinary checked exceptions – but you shouldn't do that. Do not reuse `Exception`, `RuntimeException`, `Throwable`, or `Error` directly; treat them as abstract classes.

Use exceptions only for exceptional conditions and not for ordinary control flow (in the name of misguided attempts to improve performance) [Item 69]. Also, a well-designed API must not force its clients to use exceptions for ordinary control flow.

When a method propagates an exception thrown by a lower-level abstraction that it calls, the propagated exception may not make sense for the propagating method, which is disconcerting and pollutes the API [Item 73]. To avoid this, higher layers should catch lower-level exceptions and throw exceptions that can be explained in terms of the higher-level abstraction, known as *exception translation*. Often, you want the higher layer exception and also the lower-layer exception, e.g. for purposes of debugging. A special form of exception translation called *exception chaining* additionally passes the lower-level exception to the higher level exception as constructor argument of type `Throwable`, from whence it can be retrieved with `Throwable.getCause()` and which integrates the cause's stack trace into that of the higher-level exception. Most standard exceptions have chaining-aware constructors, and if not, you can set the cause using `Throwable.initCause()`. Exception translation should not be overused; it's best to avoid exceptions from lower layers or to silently handle them.

When a program fails due to an uncaught exception, the system automatically prints the exception's stack trace with the exception's string representation (gotten with its `toString()`), which often contains the exception's class name followed by its detail message. The detail message should contain as much information as possible, namely the values of all parameters and fields that contributed to the exception [Item 75]. For example, the detail message of an `IndexOutOfBoundsException` should contain the lower bound, the upper bound, and the index value that failed. One way to ensure this is to require this information in their constructors (instead of a the regular string detail message). The detail message can then be generated automatically from the parameters (even though the Java libraries don't make heavy use of this idiom).

A *failure atomic* method leave the object in the state that it was in prior to the invocation, which you should strive for [Item 76]. This can be achieved by working with immutable objects (which give failure atomicity for free), by checking parameters for validity before performing the operation or ordering any part of the computation that may fail before any parts that modify the object, by modifying a temporary copy of the object and replacing it afterwards, or by recovery code that intercepts a failure and rolls back the object's state. Failure atomicity is not always possible or practical. API documentation should clearly indicate what state the object will be left in after a certain exception.

It is easy to ignore exceptions by surrounding a method invocation with a try statement whose catch block is empty, but this defeats the purpose of exceptions [Item 77]. If you still do, put a comment in the catch block explaining why it is appropriate, and the exception variable should be named `ignored`.

6 Reflection

Reflection allows us to inspect and manipulate classes, interfaces, constructors, methods, and fields at run time. Each `.class` file refers to the types it depends upon, so that the hierarchy of classes and interfaces is embedded in the collection of `.class` files. The JVM assembles this data into a representation of an inheritance and (interface) implementation graph during the class loading process. This inheritance metadata is always present for every type during runtime. In the HotSpot JVM, the metadata is held in an object header, which separates the metadata into instance-specific metadata (e.g. the object's intrinsic monitor/synchronization lock, information for garbage collection, etc.) and type-specific metadata (a pointer to the metadata for the class that the object belongs to). The JVM exposes this runtime metadata and type information and allows Java programmers to access and use it.

`java.lang.reflect` is the package with all of Java's reflection APIs, called the Core Reflection API. For every type of object, the JVM instantiates an immutable instance of `java.lang.Class` – which predates `java.lang.reflect` and therefore is not in it – when they are loaded, providing methods to examine the runtime properties of the object including its members and type information. `Class` also provides the ability to create new classes and objects. `Class` is the entry point for all operations in the Core Reflection API. Because the Core Reflection API predates the `Collection` API, types such as `List` do not appear in the Core Reflection API; instead, arrays are used, making some parts of the API rather more awkward to use. `Class<T>` is parameterized, with `T` the type that it represents. There are several ways to get a `Class`:

- `Object.getClass()`. If an instance of an object is available, then the simplest way is to invoke `Object.getClass()`, e.g. `"foo".getClass()` returns `Class<String>`.

```
    Set<String> s = new HashSet<String>();
    Class c = s.getClass(); // Returns a Class<java.util.HashSet>
```

- `.class`. If the type is available but there is no instance then you can append `.class` to the name of the type. This is also the only way to obtain the `Class` for a primitive type[15], on which `Object.getClass()` yields a compile-time error (since primitive types cannot be dereferenced).
- `Class.forName()`. If the fully-qualified name of a class is available as a `String`, you can use the `static` method `Class.forName()`. Arrays have their own special syntax: `Class.forName("[D")` returns the `Class` corresponding to an array of primitive type `double` (that is, the same as `double[].class`), and `Class.forName("[[Ljava.lang.String;")` returns the `Class` corresponding to a two-dimensional array of `String` (that is, identical to `String[][].class`).

There are several Reflection API methods that return `Class`es, but they take `Class`es as well:

- `Class.getSuperclass()` returns the super class for the given class.
- `Class.getClasses()` returns all the public classes, interfaces, and enums that are members of the class as an array `Class<?>[]` (including inherited members).
- `Class.getDeclaredClasses()` returns all of the classes, interfaces, and enums that are explicitly declared in this class as an array `Class<?>[]` (excluding inherited members).
- `Class/Field/Method/Constructor.getDeclaringClass()` return the Class in which these members were declared.
- `Class.getEnclosingClass()` returns the immediately enclosing class of the class.

`Class.getModifiers()` returns an `int` in which each bit represents a class modifier (basically keywords and annotations) of this object's class. `java.lang.reflect.Modifier` provides `static` methods and constants to decode such `int`s, e.g. `Modifier.toString(int m)`, `Modifier.isInterface(int m)`, `Modifier.isStatic(int m)`, etc. `Class.getTypeParameters()` returns an array of `TypeVariable<Class<T>>`s of this object's class, where a `TypeVariable<Class<T>>` represents a generic type declaration and `T` is the generic type that declares the type variable. `TypeVariable.getName()` returns the type name, e.g. `T`, `K`, or `V`. `Class.getGenericInterfaces()` returns the an array of `Type`s representing the interfaces directly implemented by the class or interface represented by this object. You can print the name of the interface with `Type.toString()`. `Class.getAnnotations()` returns `Annotation`s that are present on this element.

This page gives an overview of all the member-locating methods and their characteristics. The Reflection API defines an interface `java.lang.reflect.Member`, which is implemented by `java.lang.reflect.Field`, `java.lang.reflect.Method`, and `java.lang.reflect.Constructor`. Just like `Class`, you can programmatically parse and retrieve modifiers for all three types of member, and member names, field types, method signatures, etc. `Constructor`, `Method`, and `Field` instances let you manipulate their underlying counterparts reflectively. `Field`s can be `Field.get(Object o)` and `Field.set(Object o, Object value)`. `Method`s can be invoked with `Method.invoke()`; the first argument is the object on which its method is invoked (and this is null is the method is `statis`, and subsequent arguments are the method's parameters. If the underlying method throws an exception, it will be wrapped by an `java.lang.reflect.InvocationTargetException`. The API for `Constructor` is similar to `Method`, but constructors have no return values, and the invocation of a constructor creates a new instance of an object for a given class.

From the JVM's perspective, arrays and enums are just classes. Many of the methods in `Class` may be used on them. Reflection provides a few specific APIs for arrays and enums.

`Class<T>` has a `T cast(Object obj)` instance method, which dynamically casts the object reference to the type represented by the `Class` object on which it is called. It simply checks that its argument is an instance of the type represented by the `Class` object, in which case it returns the argument, otherwise it throws a `ClassCastException`. `Class<T>` also has a `<U> Class<? extends U> asSubclass(Class<U> clazz)` instance method, which attempts to cast the `Class` object on which it is called to represent a subclass of the class represented by its argument.

Reflection allows one class to use another, even if the latter class did not exist when the former was compiled. There are a few sophisticated applications that require reflection, such as code analysis tools and dependency injection frameworks, but even such tools have been moving away from reflection of late because of its disadvantages [Item 65]:

[15]That's not really true, as the boxed wrapper classes of the primitive types and `void` have a `static` field `TYPE` which contains the `Class` for the primitive type being wrapped.

- You lose all the benefits of compile-time type checking, including exception checking. A program invoking a nonexistent or inaccessible method reflectively will fail at runtime (unless you've taken special precautions).
- Reflective access is clumsy and verbose.
- Reflective method invocation is much slower than normal method invocation.

If you have any doubts as to whether your application requires reflection, it probably doesn't. You can obtain many of the benefits with few of its costs by using it only in a very limited form. For many programs that must use a class that is unavailable at compile time, you can use reflection only to instantiate objects, and access the objects using some interface or superclass that is known at compile time.

7 Annotations

Historically, *naming patterns* were used to indicate that some program elements demanded special treatment by a tool or framework, e.g. JUnit required test methods to have their names start with `test`. This is problematic, because typographical errors result in silent failures and you can't associate parameter values with program elements. Annotations solve these [Item 39].
Annotations are a form of syntactic metadata that can be added to Java source code. They can be embedded in and read from Java class files generated by the Java compiler, allowing them to be retained by the Java virtual machine at runtime and read via reflection. The compiler can detect errors or suppress warnings, software tools can process annotation information to generate code, XML files, and so forth, and some annotations are available to be examined at runtime. Annotations start with the `@` sign (`@` = AT, as in annotation type) and may precede declarations of classes, fields, methods, variables, parameters and packages to annotate them, each of with may have multiple annotations as well, and you can even repeat the same annotation.

An annotation can include parameters, either named or unnamed, with values:

```
@Author(
  name = "Benjamin Franklin",
  date = "3/27/2003"
)
class MyClass { ... }
```

If there is just one named parameters, then the name can be omitted, as in:

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

And if there are no parameters, you only need to give the name of the annotation, in which case it is called a *marker annotations*, because it only "marks" the annotated element:

```
@Override
void methodFromSuper() { ... }
```

You should generally prefer marker interfaces over marker annotations, especially if you want to write methods that accept only objects that have this marking, in which case you can use the marker interface type as parameter type [Item 41]. If this is not the case, and additionally, the marking is part of a framework that makes heavy use of annotations, then a marker annotation is the clear choice.

Custom annotation types can be defined as well. This is mainly useful in case a repository has many comments containing similar structure – in which case you can replace them with an annotation – or for developing tools like IDEs, or test or IoC frameworks. An annotation type declaration is a special type of an interface declaration. They are declared in the same way as the interfaces, except the interface keyword is preceded by the `@` sign:

```
@Documented // In case you want this information appear in Javadoc-generated documentation
@interface ClassPreamble {
  String author();
  int currentRevision() default 1; // Default value is 1
  String[] reviewers(); // Note use of array
}
```

The last declared parameter is an array. The syntax for array parameters in annotations is flexible, and is optimized for single-element arrays so that you can pass one value to the annotation, which populate the array. To specify a multiple-element array, surround the elements with curly braces and separate them with commas.

Java provides the following built-in annotations:

- Annotations applied to Java code (supplied in `java.lang`):
  - `@Override`; checks that the method is an override. Causes a compilation error if the method is not found in one of the parent classes or implemented interfaces. Probably the most important annotation for the typical programmer, and will avoid a large class of nefarious bugs if used consistently on every method declaration that you intend to override a superclass declaration [Item 40]. A common example of such a bug is accidentally using a different signature than the method you want to override, causing you to overload it instead.
    In concrete classes, you need not annotate methods that you believe to override abstract method declarations (though it is not harmful to do so). If you know that an interface does not have default methods, you may choose to omit the annotation on concrete implementations of interface methods to reduce clutter. It is worth annotating all methods that you believe to override superclass or superinterface methods in an abstract class or an interface, however.
  - `@Deprecated`; causes a compile warning if the method is used.
  - `@SuppressWarnings`; instructs the compiler to suppress the compile time warnings specified in the annotation parameters. Always use the `@SuppressWarnings` annotation on the smallest scope possible.
- Annotations applied to other annotations, aka *meta-annotations* (supplied in `java.lang.annotation`):
  - `@Retention`; specifies how the marked annotation is stored, whether in code only, compiled into the class, or available at runtime through reflection. For example, if you want to define an annotation type to designate tests methods, you'd use `@Retention(RetentionPolicy.RUNTIME)` to indicate that the `@Test` annotations should be retained at runtime, because that's when the test tool needs it and by default annotations are only available during compile time.
  - `@Documented`; marks another annotation for inclusion in the documentation.
  - `@Target`; marks another annotation to restrict what kind of Java elements the annotation may be applied to, e.g. `@Target(ElementType.METHOD)` meta-annotation indicates that the annotation is legal only on method declarations.
  - `@Inherited`; marks another annotation to be inherited to subclasses of annotated class (by default annotations are not inherited by subclasses).
  - `@SafeVarargs`; suppress warnings for all callers of a method or constructor with a generics varargs parameter (that is typesafe).
  - `@FunctionalInterface`; specifies that the type declaration is intended to be a functional interface. Similar to `@Override`, it is a statement of programmer intent that tells readers of the class and its documentation that the interface was designed to enable lambdas, it keeps you honest because the interface won't compile unless it has exactly one abstract method; and it prevents maintainers from accidentally adding abstract methods to the interface as it evolves. Always annotate your functional interfaces with the `@FunctionalInterface` annotation.
  - `@Repeatable`; specifies that the annotation can be applied more than once to the same declaration. It takes a single parameter, which is the class object of a *containing annotation type*, which is another annotation with as sole parameter/value an array of the annotation type, e.g. `AnnotationType[]`, which holds the multiple annotations if it is used more than once. A repeated annotation generates a synthetic annotation of the containing annotation type. The `getAnnotationsByType()` method of the Reflection API is agnostic to this and can be used to access both repeated and non-repeated annotations, but `isAnnotationPresent()` does distinguish between an annotation type and its containing annotation type, so you much check for both the annotation type and its containing annotation type if you don't want to have one of them silently ignored, e.g. `m.isAnnotationPresent(AnnotationType.class)|| m.isAnnotationPresent(AnnotationTypeContainer.class)`.
    Repeatable annotations were added to improve the readability of source code that logically applies multiple instances of the same annotation type to a given program element, but there is more boilerplate in declaring and processing repeatable annotations, and their processing is error-prone.

8 Concurrency

A *process* has a self-contained execution environment with a complete, private set of basic run-time resources, like its own memory space. They're often seen as synonymous with programs/applications, but what the user sees as a single application may be a set of cooperating processes. To facilitate communication between processes (that can also be on different systems), most operating systems support Inter Process Communication (IPC) resources, like pipes and sockets. Most implementations of the JVM run as a single process.
Every process consists of one or more *threads*, which share the process's resources, including memory and open files. This makes communication easier, but also more dangerous. Threads are sometimes called lightweight processes, because they requires fewer resources and are faster to create and abort.

Each thread is associated with a `Thread` instance. `Thread` implements the `Runnable` interface, which defines a single method, `run()`, meant to contain the code executed in the thread. `Thread` implements `run()`, but it does nothing. You can create a `Thread` with actual code by either passing a `Runnable` object to the `Thread` constructor or by subclassing `Thread` with a `run()` implementation and creating that subclass (easier to use in simple applications, but your task class must be a descendant of `Thread`). `Runnable` is a functional interface, so you can create and start a `Threads` inline like this:

```
Thread thread = new Thread(() -> { System.out.println("Hello world"); });
thread.start();
```

`Thread` defines a number of methods for thread management, which include static methods (used by `Thread`s for querying or altering their own status) and instance methods (used by `Thread`s for querying other `Thread`s). The static `Thread.sleep()` causes the current thread to suspend execution for the specified time in milliseconds (and optionally nanoseconds), which allows for pacing and making processor time available to other threads. The sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS, and because the sleep period can be terminated by interrupts. `sleep()` throws `InterruptedException` when another thread interrupts the current thread while it is sleeping.

An `interrupt` is an indication to a thread that it should stop and do something else (it commonly just terminates). The `Thread.interrupt()` instance method sends an *interrupt* to the thread, which sets an internal flag known as the interrupt status. To support its own interruption, a thread must periodically call the static `Thread.interrupted()`, and if it returns `true`, the most common reactions are to `return` or to throw a `InterruptedException`. If it frequently invokes (a) `InterruptedException`-throwing exception(s), it can also support its own interruption by catching that. By convention, any method that exits by throwing an `InterruptedException` clears interrupt status when it does so.

The `join()` instance method allows the invoking thread to wait for the completion of the thread on which it is called. Like `sleep()`, a (no-guarantees) waiting period can be specified, and it responds to an interrupt by exiting with an `InterruptedException`.

When multiple threads share mutable data, each thread that reads or writes the data must perform synchronization to prevent thread interference (the operations of different threads on the same data interleave) and memory consistency errors (different threads have inconsistent views of what should be the same data). You can use the `synchronize` keywords on methods or blocks of code to ensure that only a single thread at a time can execute it:

```
public synchronized void method() { // Acquires lock on the object on which this method is called
   // Synchronized statements
}

synchronized(someObject) { // Acquires lock on someObject, which must be a reference type and
    non-null
   // Synchronized statements
}
```

When a thread executes a synchronized method for an object, other threads that invoke synchronized methods on the same object block until the first thread exits the method. If you can't have an entire method synchronized, you can synchronize only a part of its by wrapping those statements with the second method using `synchronized(this){ }`. Locks in Java are *reentrant*, meaning a thread can reacquire a lock that it already currently owns, which allows synchronized code to (directly or indirectly) invoke other synchronized code on the same lock. Reentrant locks simplify the construction of multithreaded programs, but they can turn liveness failures into safety failures.

Synchronization ensures *mutual exclusion* (code is locked and mutable data is transformed from one consistent state to another, such that no method will ever observe the object in an inconsistent state) and reliable communication between threads (each thread entering a synchronized method or block sees the effects of all previous modifications that were guarded by the same lock) [Item 78]. Programmers often only consider the first aspect. The language specification

guarantees that reading or writing a variable is atomic (except for `long` or `double`), but it does not guarantee that a value written by one thread will be visible to another.
Do not use the unsafe and deprecated `Thread.stop()`, which can result in data corruption. A recommended way to have one thread stop another is to have the thread-to-stop poll a `boolean` field that can be set by the second thread to indicate that the first thread is to stop itself. In order to avoid *liveness failure* (when a program fails to make progress), you should either poll the field with a `synchronized` getter and set the field with a `synchronized` setter, or better, you should simply specify the `boolean` field with the `volatile` keyword. `volatile` performs no mutual exclusion, but guarantees that any thread that reads the field will see the most recently written value. Note that an unsynchronized method can update a `volatile` field non-atomically, and that two threads may interleave their read and write operations, which may cause *safety failure* (when a program computes the wrong results). If a field is exclusively read and written to by synchronized methods, you can and should remove the `volatile` modifier. While `volatile` provides only the communication effects of synchronization, the package `java.util.concurrent.atomic` has primitives which also provides atomicity for lock-free, thread-safe programming on single variables. `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, and `AtomicReference` each provide access and updates to a single variable of the corresponding type, together with relevant atomic operations. For example, the `++` operator is not atomic – it reads the value, and in a separate step it writes back a the old value plus one – but `AtomicInteger.getAndDecrement()` does this atomically. However, it is always best to confine mutable data to a single thread. *Safe publication* is also acceptable, in which one thread creates and modifies a data object and then shares it with other threads, synchronizing only the act of sharing the object reference. Other threads can then read the object without further synchronization as long as it isn't modified again.

You should avoid excessive synchronization and do as little work as possible inside synchronized regions [Item 79]. Synchronization can cause *thread contention*, which reduces performance or halts progress (like with *starvation* or *livelock*). A class can disregard synchronization and allow the client to synchronize externally if it needs to (which is generally the best option – but do document that is not thread-safe), or you can synchronize internally to make the class *thread-safe* (which you should only do if you can achieve significantly higher concurrency). Two JCL classes that synchronize unnecessarily are `StringBuffer`, of which instances are almost always used by a single thread and which is therefore supplanted by `StringBuilder` (which is just an unsynchronized `StringBuffer`); and `java.util.Random`, which was supplanted by the unsynchronized `java.util.concurrent.ThreadLocalRandom` for the same reason. Sometimes you must synchronize internally, such as when a method modifies a static field, because a multithreaded client cannot perform external synchronization on such a method.
To avoid liveness and safety failures, never cede control to the client within a synchronized method or block by invoking, inside a synchronized region, *alien* methods that are designed to be overridden or are provided by a client as a function object.

When a uses a publicly accessible lock, it enables clients to execute method invocations atomically, but this is incompatible with high-performance internal concurrency control and allows client to mount a denial-of-service attack. Unconditionally thread-safe classes can prevent this with a *private lock object* rather than using synchronized methods (which imply a publicly accessible lock). Such lock fields should always be `final`, which prevents you from inadvertently changing its contents that can cause catastrophic unsynchronized access. Conditionally thread-safe classes can't use this idiom because they must document which lock their clients are to acquire when performing certain method invocation sequences. This idiom is well-suited to classes designed for inheritance and avoids the risk of a subclass and base class using the same lock for different purposes and interfering with each others operations.

When many threads are *runnable* (i.e. not waiting), the thread scheduler determines which ones get to run and for how long. It's an OS-specific policy, and any program that relies on the thread scheduler for correctness or performance is likely to be non-portable [Item 84]. Ensure that the average number of runnable threads is not significantly greater than the number of processors for robust, responsive and portable programs. The main technique for this is to have each thread do either useful work or wait for some (e.g. threads should not busy-wait). In terms of the Executor Framework, this means sizing thread pools appropriately and keeping tasks short, but not too short, or dispatching overhead will harm performance.
As a corollary, do not rely on `Thread.yield()` – which no testable semantics, and using it to "fix" a program that barely works because some threads aren't getting enough CPU time relative to others, because even though it may improve performance on one JVM implementation, it might make it worse on a second and have no effect on a third – or thread priorities, which are among the least portable features of Java.

`java.lang.Object` has three instance methods for concurrency. `wait()` causes the current thread to release the lock of the object it holds wait until it is awakened (i.e. notified or interrupted), and reacquires it when it wakes. This means that `wait()` must be invoked inside a synchronized region that locks the object on which it is invoked. Always use the following standard wait loop idiom to invoke the wait method (never invoke it outside of a loop) [Item 81]:

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait(); // Releases lock, and reacquires on wakeup
    // Perform action appropriate to condition
}
```

Testing the condition before waiting and potentially skipping the wait are necessary to ensure liveness, and testing the condition after waiting and potentially waiting again are necessary to ensure safety. `notify()`/`notifyAll()` wakes up a single/all thread(s) that is/are waiting on this object's monitor. Using `wait()` and `notify()` directly is like programming in "concurrency assembly language". Given the difficulty of using `wait()` and `notify()` correctly, you should use the following higher-level concurrency utilities instead.

8.1 Concurrent collections

The *concurrent collections* are high-performance concurrent implementations of standard `Collection` interfaces, which manage their own synchronization internally, making it is impossible to exclude concurrent activity from a concurrent collection; locking it will only slow the program. Because of this, you can't atomically compose method invocations on them either. Therefore, concurrent collection interfaces were outfitted with *state-dependent modify operations*, which combine several primitives into a single atomic operation. These operations proved sufficiently useful on concurrent collections that they were added to the corresponding collection interfaces as default methods. For example, `Map`'s `putIfAbsent(key, value)` associates the given key with the given value and returns null if the specified key is not already present, else it returns the current value. Concurrent collections make synchronized collections (e.g. `Collections.synchronizedMap(synchronizedMap(Map<K,V> map)`, which returns a synchronized (thread-safe) map backed by the given map) largely obsolete, and replacing them with concurrent maps can dramatically increase the performance of concurrent applications.

Some of the collection interfaces were extended with blocking operations, which wait/block until they can be performed. For example, `BlockingQueue` extends `Queue` and adds methods like `take()`, which waits until it can remove and returns the head element. `BlockingQueue` can be used as work queues (also known as producer-consumer queues), to which one or more producer threads enqueue work items and from which one or more consumer threads dequeue and process items as they become available. As you'd expect, most ExecutorService imple- mentations, including ThreadPoolExecutor, use a BlockingQueue (Item 80)

`CopyOnWriteArrayList` is a variant of `ArrayList` in which all modifications are done on a copy of the underlying array. Because the underlying array is never modified, iteration requires no locking and is very fast. For most uses, the performance would be atrocious, but it's perfect for observer lists, which are rarely modified and often traversed.

8.2 Executors

In large-scale applications, it makes sense to separate thread creation/management from the rest of the application. A `Thread` represents/serves as both a unit of work and the mechanism for executing it, while an *executor* (basically a work queue) abstracts the execution mechanism away from the unit of work, which in the (high-level) executor framework is referred to as a *task*. A task is either a `Runnable` or a `Callable<V>`, which are similar to `Runnable`s but can return a result (of type `V`) and throw arbitrary exceptions. This all gives the flexibility to select different execution policies at different times. The `java.util.concurrent` package defines three executor interfaces, each subsequent one is a subinterface of the previous one:

- `Executor` is a simple interface that supports launching new `Runnable`s. It provides a single method, `execute()`, designed to replace `(new Thread(runnable)).start()` with `executer.execute(runnable)`. The low-level idiom creates a new thread and launches it immediately, but the latter is more likely to use an existing worker thread to run `runnable`, or to place `runnable` in a queue to wait for an available worker thread. `executer.shutdown()` tells the executer to terminate gracefully (if you fail to do this, it is likely that your VM will not exit).
- `ExecutorService` adds features that help manage the life cycle, both of the individual tasks and of the executor itself. It adds `submit()`, which accepts both `Runnable` and `Callable<V>` objects and which returns a `Future`, which represents the result of an asynchronous computation. `Future` has methods to check if the computation `isDone()`, to `cancel()` its computation, to determine if the task completed normally or `isCancelled()`, and to `get()` the result of the computation, blocking if necessary until it is ready. `ExecutorService` also provides methods for submitting large collections of `Callable` objects and a number of methods for managing the shutdown of the executor.
- `ScheduledExecutorService` supports future and/or periodic execution of tasks. It adds `schedule()`, which executes a `Runnable` or `Callable` task after a specified delay, and `scheduleAtFixedRate()` and `scheduleWithFixedDelay()`,

which executes specified tasks repeatedly at defined intervals.

The executor implementations in `java.util.concurrent` are designed to make full use of the more advanced `ExecutorService` and `ScheduledExecutorService` interfaces, although they also work with the base `Executor` interface. The `java.util.concurrent.Exe` class provides (static) factory methods:

- `newSingleThreadExecutor()`, for a `ExecutorService` which executes a single task at a time with a single worker thread. A *worker thread* exists separately from the `Runnable` or `Callable` tasks it executes and is often used to execute multiple of those tasks, which minimizes the overhead due to thread creation, since `Thread`s use a significant amount of memory and are costly to (de)allocate.
- `newCachedThreadPool()`, for a `ExecutorService` that uses a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available, such that submitted tasks are not queued but immediately handed off. Generally a good choice for a small program or lightly loaded server, because it demands no configuration and generally "does the right thing", but if a server already utilizes all its CPUs and more tasks arrive, more threads will be created, which will only make matters worse.
- `newFixedThreadPool()`, for a `ExecutorService` that uses a *fixed thread pool*, which always has a specified number of worker threads running, and terminated threads are automatically replaced with new ones. Tasks are submitted to the pool via an internal queue where they wait for threads. Applications using a fixed thread pool degrade gracefully; rather than shutting down when overloading, the system just takes longer per request. Best for heavily loaded production servers.
- `ScheduledExecutorService` versions of the above.
- etc.

These provide most of the executors you'll ever need. If you need something more custom, you can use the `ThreadPoolExecutor` class directly, which lets you configure nearly every aspect of a thread pool's operation.
You should prefer executors, tasks, and streams to threads [Item 80].

8.3 Synchronizers

Synchronizers are objects that enable threads to wait for one another, allowing them to coordinate their activities:

- `CountDownLatch`; single-use/one-shot barriers that allow one or more threads to wait for n other threads to do something. Its sole constructor takes an `int` n, the number of times the `countDown()` method must be invoked on the latch before all `await()`ing threads are allowed to proceed.
- `Semaphore`
- `CyclicBarrier`
- `Exchanger`
- `Phaser`

9 Streams

The streams API is meant to ease the task of performing bulk operations, sequentially or in parallel. A *stream* is a finite or infinite sequence of data elements which can be object references (`Stream<T>`), `int`s (`IntStream`), `long`s (`LongStream`), and `double`s (`DoubleStream`) and which can come from anywhere (e.g. collections, arrays, files, regular expression pattern matchers, pseudorandom number generators, other streams). A *stream pipeline* is a multistage computation on these elements consisting of a source stream followed by zero or more intermediate operations and one terminal operation, each of which transforms the stream in some way (e.g. mapping, filtering, etc.). Stream pipelines are evaluated lazily: evaluation doesn't start until the terminal operation is invoked, and data elements that aren't required in order to complete the terminal operation are never computed. Terminal operations are *eager*, completing their traversal of the data source and processing of the pipeline before returning. This is efficient, as multiple operation can be fused into a single pass on the data with minimal intermediate state, and because laziness allows avoiding examining all the data when it is not necessary.

The streams API is *fluent*: it is designed to allow all of the calls that comprise a pipeline to be chained into a single expression. In fact, multiple pipelines can be chained together into a single expression. An example of a stream pipeline:

```
int sum = widgets.stream() // widgets is a Collection<Widget>. Collection.stream() returns a
```

```
Stream<Widget>
          .filter(b -> b.getColor() == RED) // Filter it to produce a new stream of red widgets
          .mapToInt(b -> b.getWeight()) // Transform it into a stream of ints
          .sum(); // Sum to produce a total weight
```

Streams sources can be any of

- a collection with `Collection.stream()` or `Collection.parallelStream()`
- an array via `Arrays.stream(Object[])`
- a static factory methods on the stream classes, such as `Stream.of(Object[])`, `IntStream.range(int, int)` or `Stream.iterate(Object, UnaryOperator)`. The last method generates an infinite stream by taking the first element in the stream and continuously applying the operator on the last element in the stream
- a file with `BufferedReader.lines()` or the static `Files.lines(Path path)`
- a `Scanner` with `Scanner.tokens()`, which is a simple text scanner which can parse primitive types and strings using regular expressions. The `Scanner` constructor takes a `File`, `Path`, `InputStream`, or `String`, and can then iterate over the delimited content with `next()` and `hasNext()`, but, as we said, can also generate a stream.
- an infinite stream of random numbers, e.g. generated with `Random.ints()`
- `IntStream.range()` and `IntStream.rangeClosed()` are basically the stream equivalent of the standard for-loop on integer indices.
- etc.

Intermediate operations can be any of

- `Stream.filter()`, which does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.
- `Stream.map()` applies the given function to the elements of this stream, where the function simply produces one result value per input value consumed.
- `Stream.flatMap()` applies the given function to the elements of this stream, where the function conceptually produces an arbitrary number of values for each input value consumed. Because this is the streams library, the function returns a mapper function returns a `Stream` to represent an arbitrary number of values. These values are drained from the stream and passed flatly (without distinguishing between different groups of output values) to the output stream. It can be seen as a combination of a `map()` and a `flat()` call.
- `Stream.limit()` truncates the stream to the given size.
- `Stream.sorted()` sorts the elements according to the provided `Comparator`, or according to natural ordering of none is specified.
- `Stream.concat()` returns a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
- etc.

Terminal operations can be any of

- `Stream.forEach()`, which executes the given `Consumer` on each element in the stream. Among the least powerful of the terminal operations the least stream-friendly, explicitly iterative, and not amenable to parallelization. It should only be used to report the result of a stream computation (e.g. `forEach(System.out::println)`), not to perform the computation.
- `IntStream.sum()`, etc.
- `Stream.collect()` performs a mutable reduction operation on the elements using a `Collector`, which encapsulates a reduction strategy. *Reduction* means combining the elements of a stream into a single object. There are three collectors that return true `Collection`s: `toList()`, `toSet()`, and `toCollection(collectionFactory)`. The simplest map collector is `toMap(keyMapper, valueMapper)`, which takes a function mapping elements to keys and a function mapping elements to values. If multiple stream elements map to the same key, the pipeline will terminate with an `IllegalStateException`. Therefore, more complicated forms of `toMap()` allow you to deal with such collisions. One way is to provide `toMap()` with an additional `BinaryOperator<V>` merge function. This three-argument form of `toMap()` is also useful to make a map from a key to a chosen element associated with that key or to produce a

collector that imposes a last-write-wins policy when there are collisions. The third and final version of `toMap()` takes a fourth argument, which is a map factory for use when you want to specify a particular map implementation. In addition to `toMap()` method, the Collectors API provides `groupingBy()`, which returns `Collector`s to produce maps that group elements into categories based on a classifier function, where the categories serve as the keys. The simplest version of the `groupingBy()` method takes only a classifier and returns a map whose values are lists of all the elements in each category. If you want `groupingBy()` to return a collector that produces a map with values other than lists, you can specify a *downstream collector* in addition to a classifier, which produces the value per category, e.g. `toSet()` produces sets as values rather than lists, `toCollection(collectionFactory)` lets you create the collections, and `counting()` associates each category with the number of elements in the category (rather than a collection containing the elements). The third version of `groupingBy()` lets you specify a map factory in addition to a downstream collector[16], which gives you control over what implementation of the containing map as well as the contained collections.
`Collectors.minBy` and `maxBy` take a comparator and return the minimum or maximum element in the stream, i.e. they don't involve `Collection`s. They are minor generalizations of the `Stream.min()` and `max` and are the collector analogues of the binary operators returned by the like-named methods in `BinaryOperator`.
`joining()` operates only on streams of `CharSequence` instances such as strings and returns a collector that simply concatenates the elements in its parameterless form. Its one argument form takes a delimiter for inserting between adjacent elements. The three argument form takes a prefix and suffix in addition to the delimiter.

- etc.

A stream should be operated on only once, so that a source can't feed two or more pipelines and can't be traversed more than once.
Many terminal operations on streams return `Optional`s. A `Stream<Optional<T>>` is also not uncommon. If you want to get a `Stream<T>` containing all the elements in the nonempty optionals, you can do:

```
streamOfOptionals.filter(Optional::isPresent).map(Optional::get)
```

or more concisely, with the `Optional.stream()` adapter that turns an `Optional` into a `Stream` containing one element if it is present in the optional, or none if it is empty:

```
streamOfOptionals.flatMap(Optional::stream)
```

Streams make it very easy to uniformly transform, filter, combine, accumulate, and search sequences of elements. With streams, it is hard to access corresponding elements from multiple stages of a pipeline simultaneously. Streams can make programs shorter and clearer, but when used inappropriately, they can make programs difficult to read and maintain [Item 45]. You should refrain from using streams to process `char` values. If you're not sure whether a task is better served by streams or iteration, try both and see which works better. In the absence of explicit types, careful naming of lambda arguments to stream operations is essential to the readability of stream pipelines. Using helper methods is even more important for readability in stream pipelines than in iterative code because pipelines lack explicit type information and named temporary variables. Because the concept of streams is based on the paradigm of functional programming, any function objects that you pass into stream operations should be a *pure function*, i.e. one where the result depends only on the input and is free of side-effects (changing external state etc.) [Item 46]. If you're using streams but not the FP paradigm, then you just have iterative code masquerading as streams code, and you're better of using iterative code instead. Java programmers know how to use for-each loops, and they therefore often misuse the `forEach()` terminal operation, because it is similar, but as stated above, it should only be used to report the result of a stream computation.

Implementing the `Iterable<T>` interface allows the object to be the target of a for-each loop. The interface contains one abstract method, `iterator()`, which returns an `Iterator<T>`. The `Stream` interface contains an `iterator()` method, and `Stream`'s specification for this method is compatible with `Iterable`'s, but `Stream` frustratingly does not extend `Iterable`, preventing `Stream`s to be used in for-each loops. You could implement a `Stream`-to-`Iterable` adapter, but that's annoying. If you're writing a method in a public API that returns a sequence of objects, you ideally should want to return a `Stream` for people that write stream pipelines and an `Iterable` for people who write for-each statements.[17] Since the `Collection` interface is a subtype of `Iterable` and has a `stream()` method, it provides for both iteration and stream access, making `Collection` or an appropriate subtype generally the best return type for a public, sequence-returning method [Item 47]. Arrays also provide for easy iteration and stream access with `Arrays.asList()` and `Stream.of()` methods. If the sequence you're returning is small enough to fit easily in memory, you're probably best off returning one of the standard collection

[16]This method violates the standard telescoping argument list pattern; the `mapFactory` parameter precedes, rather than follows, the `downStream` parameter.
[17]Unless you have a good reason to believe that most users only use one of the two mechanisms

implementations, but do not store a large sequence in memory just to return it as a collection. If the sequence you're returning is large but can be represented concisely, consider implementing a special-purpose collection. A downside is that `Collection` has an `int`-returning `size()` method, which limits the length of the returned sequence to `Integer.MAX_VALUE`. If it isn't feasible to return a `Collection`, return a `Stream` or `Iterable`, whichever seems more natural.

By default, stream pipelines run sequentially. Making a pipeline execute in parallel is as simple as invoking the parallel method on any stream in the pipeline, e.g. calling `Collection.parallelStream()` instead of just `stream()`, or by invoking `BaseStream.parallel()` on a `Stream`. However, it is seldom appropriate to do so [Item 48].
Concerning performance, parallelizing a pipeline is unlikely to increase its performance if the source is `Stream.iterate()`, or the intermediate operation `limit()` is used, and can even make it worse. Performance gains from parallelism are best on streams over `ArrayList`s, `HashMap`s, `HashSet`s, `ConcurrentHashMap`s, arrays, `int` ranges, and `long` ranges, because they can all be split into subranges to be worked by parallel threads[18], and because they provide good-to-excellent *locality-of-reference*; sequential element references are stored together in memory. Without it, threads spend much of their time idly waiting for data to be transferred from memory into the processor's cache. Primitive arrays offer the best locality-of-reference, because the data itself is stored contiguously in memory. Object arrays may have less locality-of-reference, because the objects referred to by those references may not be close together. If a terminal operation is inherently sequential and does a lot of work, this greatly hampers parallelizability, whereas terminal operations that are reductions (except mutable reductions performed by `Stream.collect()` or are short-circuiting (e.g. `anyMatch()`, `allMatch()`, `noneMatch()`) are very suitable. If you write your own `Stream`, `Iterable`, or `Collection` implementation and you want decent parallel performance, you must override `spliterator()` and test the parallel performance of the resulting streams extensively. In general, you won't get a good speedup from parallelization unless the pipeline is doing enough real work to offset the costs associated with parallelism. Under the right circumstances, it is possible to achieve near-linear speedup in the number of processor cores.
Concerning correctness, parallelizing a pipeline can lead to incorrect results and unpredictable behavior (*safety failures*). Even though the Stream specification places stringent requirements on mappers, filters, and other programmer-supplied function objects[19], nothing prevents you from violating these requirements.

10 Serialization

Object serialization is Java's framework for encoding objects (states) as byte streams (*serializing*) and reconstructing objects from their encodings (*deserializing*), used for sending from one VM to another or storing on disk for later deserialization. A Java object is serializable if its class or any of its superclasses implements either the `java.io.Serializable` marker interface or its subinterface, `java.io.Externalizable`. Only information that identifies a class is recorded, but not a class's definition ("class file"), and it is the responsibility of the system to locate and load the necessary class files. Static fields belong to a class and are also not serialized, and fields with the `transient` keyword are ignored. Every instance field that can be declared `transient` (i.e. that is not part of the logical state of the object) should be.

`java.io.ObjectInputStream`/`ObjectOutputStream` are high level classes that both extend and are constructed with a `java.io.InputStream`/`OutputStream` and that have `readObject()`/`writeObject(Object o)` to read/write primitive types and objects graphs (basically dependency graphs). `readObject()` is essentially a magic constructor that can be made to instantiate objects of any `Serializable`-implementing type on the class path, and can execute code from any of these types, and all this code therefore contributes to the large `attack surface`[20] of Java serialization, which is a fundamental problem [Item 85].
Attackers and security researchers study the serializable types in the Java and third-party libraries, looking for methods invoked during deserialization that perform potentially dangerous activities called *gadgets*. Multiple gadgets can form a *gadget chain*, which are sometimes sufficiently powerful to allow executing arbitrary native code on the underlying hardware, given only the opportunity to submit a carefully crafted byte stream for deserialization.
Even without using any gadgets, you can easily mount a denial-of-service attack with *deserialization bombs*, short streams that require a long time to deserialize.

You implement custom serialization by defining `readObject(ObjectInputStream s)` and `writeObject(ObjectOutputStream s)` on your class. The serialization specification requires you to first invoke `defaultReadObject()`/`defaultWriteObject()` on the `ObjectInput/OutputStream` even if you define a custom serialization, which makes it possible to add non-`transient` instance fields in a later release while preserving backward and forward compatibility. If an instance is serialized in a later version and deserialized in an earlier version, the added fields will be ignored. You must impose

[18]To do this, the streams library uses the *spliterator*, returned by `spliterator()` on `Stream` and `Iterable`.

[19]For example, the accumulator and combiner functions passed to Stream's reduce operation must be associative, non-interfering, and stateless.

[20]The attack surface includes classes in the Java platform libraries, in third-party libraries such as Apache Commons Collections, and in the application itself

any synchronization on object serialization that you would impose on any other method that reads the entire state of the object.

Even though it's easy to serialize a class by adding `implements Serializable`, you should do so with great caution [Item 86]. Its byte-stream encoding (or *serialized form*) becomes part of its exported API. The Java platform specifies a default serialized form, which is a reasonably efficient encoding of the *physical* representation of the object graph rooted at the object, i.e. it describes the data contained in the object and in every object that is reachable from this object, together with the topology by which they're interlinked. A class can override this default serialization and define its own way of serializing objects of that class. You should seriously consider this, and only go for the default if that encoding is reasonable from the standpoint of flexibility, performance, and correctness [Item 87]. Since the ideal serialized form of an object contains only its *logical* data, the default serialized form can be appropriate if an object's physical representation is identical to its logical content (but even then, you often must provide a `readObject()` method to ensure invariants and security). If this is not the case, the default serialization has four disadvantages:

- The serialized form will forever be tied to the class's original internal representation (including private and package-private instance fields), and since you're generally required to support the serialized form forever, you lose flexibility to change a class's implementation.[21]
- It can consume excessive space. For example, a serialized form can unnecessarily represent each entry in a linked list and all the links, even though these are implementation details now worthy of inclusion.
- It can consume excessive time. The serialization logic has no knowledge of the topology of the object graph, so it must go through an expensive graph traversal.
- It can cause stack overflows during the recursive traversal of the object graph, even for moderately sized object graphs.

Also, deserialization is a "hidden constructor" with all of the same issues as other constructors. Relying on the default deserialization mechanism can easily leave objects open to invariant corruption and illegal access; the default `readObject()` is essentially a public constructor that takes as parameters the values for each nontransient field in the object and stored the values in the fields with no validation whatsoever. That is, you should provide a `readObject()` for your class that (first calls `defaultReadObject()` and then) checks the validity of the deserialized object, throwing `InvalidObjectException` if invalid. What's more, during deserialization (and before validity checking), immutable classes must defensively copy any field containing a private reference to a mutable object, because an attacker can obtain such references by appending those additional rogue "previous object references" to the byte stream. Like a constructor, a `readObject()` method must not invoke an overridable method, either directly or indirectly. If an entire object graph must be validated after it is deserialized, use the `ObjectInputValidation` interface.
Lastly, implementing `Serializable` increases the testing burding, as with every revision it needs to be checked that it is possible to serialize an instance in the new release and deserialize it in old releases, and vice versa.
Because every subclass of a `Serializable`-implementing class is serializable as well, you should rarely do so if you design a class for inheritance. If you it anyways, it is critical to prevent subclasses from overriding the `finalize()` method by overriding it and declaring it `final`, or else the class will be susceptible to finalizer attackss. And if the class has invariants that would be violated if its instance fields were initialized to their default values (e.g. `false` for `boolean`, null for reference types), you must add this `readObjectNoData()` method, which was added in Java 4 to cover a corner case involving the addition of a serializable superclass to an existing serializable class:

```
private void readObjectNoData() throws InvalidObjectException {
   throw new InvalidObjectException("Stream data required");
}
```

The best way to avoid serialization exploits is to never deserialize anything, and else, only deserialize trusted data. Avoid RMI traffic from untrusted sources in particular. If you can't avoid serialization of untrusted data, use the object deserialization filtering facility `java.io.ObjectInputFilter`, which lets you specify a filter that is applied to data streams before they're deserialized. It operates at the class granularity, letting you accept or reject certain classes. Accepting classes by default and rejecting a list of potentially dangerous ones is known as *blacklisting*, while rejecting classes by default and accepting a list of those presumed safe is known as *whitelisting*; you should prefer the latter. A tool called Serial Whitelist Application Trainer (SWAT) can be used to automatically prepare a whitelist for your application. The filtering facility will also protect you against excessive memory usage, and excessively deep object graphs, but it will not protect you against serialization bombs like the one shown above.

[21] It is still possible (using `ObjectOutputStream.putFields()` and `ObjectInputStream.readFields()`), but it can be difficult and is ugly.

Every serializable class has a unique identification number associated with it, called a *stream unique identifier*, but more commonly known as a *serial version UID*. If you do not specify this number by declaring a static final long field named `serialVersionUID`, the system automatically generates it at runtime by applying a cryptographic hash function (SHA-1) to the structure of the class. If you change the name of the class, the interfaces it implements, or any of its members, the generated serial version UID changes and compatibility with previous serializations will be broken, resulting in an `InvalidClassException` at runtime. For this reason (and the small performance benefit of not hashing the object), regardless of what serialized form you choose (even the default), declare an explicit `serialVersionUID` in every serializable class you write. It is not required that serial version UIDs be unique. If you ever want to make a new version of a class that is incompatible with existing versions, you need only change the serial version UID value.

If you enforce singletonness by having a private constructor and a static fields that holds the instance, you compromise this singletonness if you add `implements Serializable`. Any `readObject()` method, whether explicit or default, returns a newly created instance, distinct from the static instance.
The `readResolve()` feature allows you to replace/substitute/resolve another instance for the one created by `readObject()`, i.e., if the deserialized object's class defines `readResolve()` method with the proper declaration, it is invoked on the newly created object after it is deserialized, and the object reference returned by this method is then returned to the caller instead of the deserialized object. In other words, the deserialized object is simply ignored and no reference to it is retained, so it immediately becomes eligible for garbage collection. For this reason, if you depend on `readResolve()` for instance control, all instance fields with object reference types must be declared transient. Otherwise, it is possible for a determined attacker to secure a reference to the deserialized object before its `readResolve()` method is run. As mentioned before in Item 3, you're better off making such a class a single-element enum type, in which case it's both serializable and instance-controlled. However, for a serializable instance-controlled class whose instances are not known at compile time, you need to resort to the `readResolve()` way.

There is way to implement `Serializable` at a reduced risk of bugs and security problems with a technique called *serialization proxy pattern* [Item 90]. A *serialization proxy* is a private nested class that concisely represents the logical state of an instance of the enclosing, meant-to-be-serialized class. The default serialized form of the serialization proxy is the perfect serialized form of the enclosing class. Both the enclosing class and its serialization proxy must implement `Serializable`. The sole constructor of the proxy takes as sole argument an instance of the enclosing class and merely copies its data (no consistency checking or defensive copying needed). The following `writeReplace()` method in the enclosing class will translate an instance of the enclosing class to its serialization proxy prior to serialization:

```
private Object writeReplace() {
   return new SerializationProxy(this);
}
```

The serialization system will never generate a serialized instance of the enclosing class, but an attacker might still fabricate one. To defend against this, simply add a `readObject()` to the enclosing class which always throws a `InvalidObjectException`.
Finally, provide a `readResolve()` method on the `SerializationProxy` class that returns a logically equivalent instance of the enclosing class, which essentially makes the serialization system translate the proxy back upon deserialization.
This pattern largely eliminates the extralinguistic character of serialization. The deserialized instance is created with the class's public API, i.e. using the same constructors, static factories, and methods as any other instance. You only – as usual – have to ensure that the class's static factories or constructors establish the invariants and that its instance methods maintain them, but you don't need to worry about deserialization maintains them. Besides, the pattern negates the bogus byte-stream attack and the internal field theft attack, allows the fields to be final (required for true immutability), and doesn't involve a great deal of thought. Lastly, it allows the deserialized instance to have a different class from the originally serialized instance. `EnumSet` has no public constructors, only static factories. From the client's perspective, those return `EnumSet` instances, but in the current OpenJDK implementation, they return one of two subclasses: a `RegularEnumSet` if the underlying enum type has 64 or fewer elements, and a `JumboEnumSet` otherwise. Because `EnumSet` uses the serialization proxy pattern, if you serialize a `RegularEnumSet` whose enum type has sixty elements, then add five more elements to the enum type, and then deserialize the enum set, it will be a `JumboEnumSet` instance once it is deserialized. However, the pattern is not compatible with classes that are extendable by their users, nor with some classes whose object graphs contain circularities (if you attempt to invoke a method on such an object from within its serialization proxy's `readResolve()` method, you'll get a `ClassCastException` because you don't have the object yet, only its serialization proxy). Finally, it is more expensive to serialize and deserialize then defensive copying.

There is no reason to use Java serialization in any new system you write. There are other mechanisms referred to as *cross-platform structured-data representations* (or sometimes, serialization systems) that translate between objects and byte sequences that avoid many of the dangers of Java serialization, while being simpler and offering cross-platform support, high performance, a large ecosystem of tools, and a broad community of expertise. They don't support arbitrary object graphs, but only support simple, structured/nested attribute-value pairs with only a few primitive and array data types, but these turn out to be sufficient for building extremely distributed systems. The two leading cross-platform structured data representations are

- JSON is text-based and human-readable (and very efficient at that) and was designed for browser-server communication (originally for JavaScript). JSON is exclusively a data representation.
- Protocol Buffers, aka protobuf, are binary and substantially more efficient and were designed by Google for storing and interchanging structured data among its servers. Protobuf offers *schemas* (types) to document and enforce appropriate usage. pbtxt is an alternative text representation for protobuf.

11 Javadoc

The Java programming environment offers the Javadoc utility for generating API documentation automatically from source code with specially formatted documentation comments, or *doc comments*. You should write doc comments for all exposed API elements (i.e. every exported class, interface, constructor, method, and field declaration) [Item 56]. Public classes should not use default constructors, because there is no way to provide doc comments for them. For maintainable code, you should also write (less thorough) doc comments for most unexported stuff. The doc comment for a method should describe succinctly the contract between the method and its client, saying what it does (not how it does it). An example:

```
/**
* Returns the element at the specified position in this list.
*
* <p>This method is <i>not</i> guaranteed to run in constant
* time. In some implementations it may run in time proportional
* to the element position.
*
* @param index index of element to return; must be
* non-negative and less than the size of this list
* @return the element at the specified position in this list
* @throws IndexOutOfBoundsException if the index is out of range
* ({@code index < 0 || index >= this.size()})
*/
E get(int index);
```

The doc comment should enumerate all of the method's preconditions (things that have to be true before calling) – which are described implicitly by the `@throws` tag that specifies unchecked exceptions (one for each precondition violation), and can also be specified per relevant parameter with that parameter's `@param` tag – its postconditions (things that will be true after the invocation), and any side effects (an observable change in the state of the system that is not obviously required in order to achieve the postcondition). The doc comment should have a `@param` tag with a noun phrase for every parameter, a `@return` tag with a noun phrase unless the method has `void` as return type or the text would be identical to the method description, and a `@throws` tag with "if"-clause for every individual (checked or unchecked) exception with precise conditions under which each one is thrown [Item 74]. (You shouldn't declare unchecked exceptions with the `throws` keyword, however. Documenting an exception without declaring it gives a cue that an exception is unchecked.) If an exception is thrown by many methods in a class for the same reason, you can document it in the class's documentation comment. Be sure to document all type parameters for generic types and methods (e.g. `@param <K> blabla...`), all constants of an enum type, and all members of an annotation type. Javadoc translates doc comments into HTML, and arbitrary HTML elements in doc comments end up in the resulting HTML document. The `@code` tag causes the code fragment to be rendered in code font and suppresses processing of HTML markup and nested Javadoc tags in the code fragment. The `@literal` tag does the same except render it in code font, so you can use to generate documentation that contains HTML metacharacter, e.g. `{@literal r < 1}`. For multiline code, use `<pre>{@code here}</pre>`. By convention, the word "this" refers to the object on which a method is invoked when it is used in the doc comment for an instance method. Doc comments should be readable both in the source code and in the generated documentation. The first sentence (or preferably and according to convention, a verb or noun phrase) of each doc comment becomes the `summary` description of the element, which ends at the first period that is followed by a space, tab, or line terminator, or at the first block tag (you can use `@literal` to escape such sequences. No two members or constructors in a class or interface should have the same summary description, so pay particular attention to descriptions of overloadings.
When you design a class for inheritance, you must document its *self-use patterns* (so programmers know the semantics of overriding its methods) with the `@implSpec` tag, which describe the contract between a method and its subclass, allowing subclasses to rely on implementation behavior if they inherit the method or call it via `super`.

Javadoc also generates a client-side index, which takes the form of a search box in the upper-right corner of the page. API elements are indexed automatically, but you can add additional terms with the `@index` tag, e.g.:

```
* This method complies with the {@index IEEE 754} standard.
```

Package-level doc comments should be placed in a `package-info.java` file (with a package declaration and its possible annotations) and module-level comments should be placed in a `module-info.java` file. You should document the thread-safety level of classes or static methods, and the serialized form of serializable classes. The latter is done by using the `@serial` tag to document private fields if those private fields define a public API through the serialized form of the class, and the `@serialData` tag to document what data private serialization methods write (`writeObject()`). document private fields with the `@serial` tag if those private fields define a public API through the serialized form of the class. Both tags tell Javadoc to place that documentation on a special page that documents serialized forms.

If an API element does not have a doc comment, Javadoc searches for the most specific applicable doc comment to "inherit", giving preference to interfaces over superclasses. You can also inherit parts of doc comments from supertypes using the `@inheritDoc` tag, allowing reusing of doc comments from interfaces that elements implement, rather than copying these comments.
Javadoc automatically checks for adherence to many of the recommendations in this item.

How a class behaves when its methods are used concurrently and its thread safety are an important part of its contract with its clients and its documentation [Item 82]. Javadoc does not include the synchronized modifier in its output, and with good reason: it is an implementation detail, not a part of its API. A class must clearly document what level of thread safety it supports:

- *Immutable*; instances of this class appear constant, and no external synchronization is necessary. Examples: include `String`, `Long`, `BigInteger`.
- *Unconditionally thread-safe*; instances of this class are mutable, but the class has sufficient internal synchronization that its instances can be used concurrently without any external synchronization. Examples: `AtomicLong` and `ConcurrentHashMap`
- *Conditionally thread-safe*; like unconditionally thread-safe, except some methods require external synchronization for safe concurrent use. Documenting these requires care; you must indicate which invocation sequences require external synchronization, and which lock(s) (typically, the lock on the instance itself) must be acquired to execute these sequences. Examples: 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 with external synchronization. Examples: the general-purpose collection implementations like `ArrayList` and `HashMap`.
- *Thread-hostile*; this class is unsafe for concurrent use even if every method invocation is surrounded by external synchronization. Thread hostility usually results from modifying static data without synchronization. Such classes or methods typically result from the failure to consider concurrency and are typically fixed or deprecated.

Unless it is obvious from the return type, static factories must document the thread safety of the returned object.

12 Optimization

Strive to write good programs rather than fast ones [Item 67]. Good programs embody the principle of *information hiding*; where possible, they localize design decisions within individual components so those can be changed without affecting the rest. However, you should keep performance in mind during the design process to avoid decisions that limit performance, especially regarding APIs, wire-level protocols, persistent data formats, and other "permanent" components that with the outside world. It is a very bad idea to warp an API to achieve good performance, because performance issues may go away in a future release. It is generally the case that good API design is consistent with good performance. Once you've carefully designed and cleanly implemented your program and performance is not good enough, then it's time to optimize. Measure performance before and after each attempted optimization and use profiling tools to help you decide where to focus your optimization efforts. The Java Microbenchmark Harness (JMH) is a *microbenchmarking framework* for building, running, and analysing nano/micro/milli/macro benchmarks that provides unparalleled visibility into the detailed performance of Java code. The need to measure the effects of attempted optimization is even greater in Java than more traditional languages like C/C++, because Java has a weaker/less defined *performance model*; the “abstraction gap” between what the programmer writes and what the CPU executes is greater and the relative cost of the various primitive operations is less well defined. It also varies from implementation to implementation, release to release, and processor to processor.

Lazy initialization is the act of delaying the initialization of a field until its value is needed. It's mainly used for optimization, but also to break harmful circularities in class and instance initialization. As with most optimization, use normal initialization unless you need lazy initialization [Item 83]. If two or more threads share a lazily initialized field, you need synchronization.
If you use lazy initialization to break an initialization circularity, use a synchronized accessor:

```
private FieldType field;

private synchronized FieldType getField() {
   if (field == null)
      field = computeFieldValue();
   return field;
}
```

If you need to use lazy initialization for performance on a static field, use the *lazy initialization holder class idiom*, which exploits the guarantee that a class will not be initialized until it is used:

```
private static class FieldHolder {
   static final FieldType field = computeFieldValue();
}

private static FieldType getField() { return FieldHolder.field; } // Causes initialization of the
    FieldHolder class
```

If you need to use lazy initialization for performance on an instance field, use the *double-check idiom*:

```
private volatile FieldType field; // Volatility is critical, because there is no locking once the
    field is initialized

private FieldType getField() {
   FieldType result = field; // This local variable ensures that field is read only once if it's
       already initialized, for performance (but it's not necessary)
   if (result == null) { // First check (no locking)
      synchronized(this) {
         if (field == null) // Second check (with locking)
            field = result = computeFieldValue();
      }
   }
   return result;
}
```

If the instance field can tolerate repeated initialization, you can dispense with the second (synchronized) check, in which case it becomes the *single-check idiom*. If you don't care whether every thread recalculates the value of a field and the type of the field is a primitive other than `long` or `double`, then you may remove the `volatile` modifier in the single-check idiom, in which case it becomes the *racy single-check idiom*.

For interval timing, always use `System.nanoTime` rather than `System.currentTimeMillis`, because it is both more accurate and more precise and is unaffected by adjustments to the system's real-time clock.

13 Build tools

13.1 Apache Maven

Maven is a build automation and dependency management tool used primarily for Java projects/artifacts (but also supports C#, Ruby, Scala, etc.). Maven uses the Convention over Configuration philosophy, i.e. it provides default values for the project's configuration and only exceptions need to be specified, and the more you stick to conventions, the smaller your POM file will stay.

13.1.1 POM

Maven projects are configured using an XML-based *project object model* (POM) in a `pom.xml` file, which specifies how a project is built and its dependencies. A minimal POM need only contain a `<project> root`, `<modelVersion>`, `<groupId>`,

`<artifactId>`, and `<version>`. A complete POM reference. A maximalist POM example:

```
<project> <!-- Always the top-level element -->
   <modelVersion>4.0.0</modelVersion> <!-- Indicates what version of the object model this POM is
       using. Changes very infrequently, but it is mandatory. Is always 4.0.0 for Maven 2.x POMs -->

   <!-- The parent of our POM, from which configuration and properties are inherited -->
   <parent>
      <groupId>com.mycompany.app</groupId>
      <artifactId>my-parent-app</artifactId>
      <version>1</version>
      <relativePath>../parent/pom.xml</relativePath> <!-- You only need to specify this if the
          parent POM is not in the parent directory -->
   </parent>

   <!-- Project coordinates which uniquely identify this project/artifact -->
   <groupId>com.mycompany.app</groupId> <!-- Fully qualified domain name of the organization. You
       can omit this if you want to inherit -->
   <artifactId>my-app</artifactId> <!-- Unique base name of the artifact being generated by this
       project (typically a JAR) -->
   <version>1.0</version> <!-- Version of the artifact generated by the project. You can omit this
       if you want to inherit -->
   <packaging>jar</packaging> <!-- Default is JAR, so this is superfluous. Different options bind
       different goals to phases -->

   <!-- Display name and URL of the project. Optional. -->
   <name>my-app</name>
   <url>http://www.example.com</url>

   <dependencies>
      <dependency>
         <groupId>org.junit.jupiter</groupId>
         <artifactId>junit-jupiter-engine</artifactId>
         <version>5.9.1</version>
         <scope>test</scope>
      </dependency>
   </dependencies>

   <!-- Handles things like declaring your project's directory structure and managing plugins -->
   <build>
      <pluginManagement>
         <plugins>
            <!-- This plugin is used by default so this just changes the configuration -->
            <plugin>
               <groupId>org.apache.maven.plugins</groupId>
               <artifactId>maven-compiler-plugin</artifactId>
               <version>3.8.1</version>
               <!-- Applies the given parameters to every goal of this plugin -->
               <configuration>
                  <source>1.5</source>
                  <target>1.5</target>
               </configuration>
            </plugin>

            <plugin>
               <groupId>org.codehaus.modello</groupId>
               <artifactId>modello-maven-plugin</artifactId>
               <version>1.8.1</version>
               <!-- You can run the same goal multiple times (i.e. multiple executions) with
                   different configuration -->
               <executions>
                  <execution>
                     <phase>generate-sources</phase> <!-- Just for demonstration, because
                         modello:java only makes sense in this phase and that phase is the default
                         -->
                     <configuration>
                        <models>
                           <model>src/main/mdo/maven.mdo</model>
                        </models>
                        <version>4.0.0</version>
                     </configuration>
                     <goals>
                        <goal>java</goal> <!-- modello:java generates Java source code -->
                     </goals>
```

```
                </execution>
             </executions>
          </plugin>
       </plugins>
    </pluginManagement>

    <!-- References to external files with values for filtering. Alternatively, you can define
        values in <properties> -->
    <filters>
       <filter>src/main/filters/myfilters.properties</filter>
    </filters>

    <resources>
       <resource>
          <directory>src/main/resources</directory>
          <filtering>true</filtering> <!-- All resources are filtered, meaning variables
              (delimited by ${...} or @...@) are replaced with values. False by default, in which
              case the variables remain. -->
       </resource>
    </resources>
 </build>

 <!-- Filter values (basically variables). Alternatively, you can define these properties
     externally and give the file(s) in <filters> -->
 <properties>
    <my.filter.value>hello</my.filter.value>
 </properties>

 <!-- For deploying JARs to an external repository. Authentication information goes in
     settings.xml -->
 <distributionManagement>
    <repository>
       <id>mycompany-repository</id>
       <name>MyCompany Repository</name>
       <url>scp://repository.mycompany.com/repository/maven2</url>
    </repository>
 </distributionManagement>
</project>
```

A conventional Maven project directory structure has the `pom.xml` in the root directory (accessible as `${project.basedir}`), and `${project.basedir}/src/main/java`, `src/main/resources`, `src/test/java`, and `src/test/resources` directories.

Larger projects should be divided into sub-projects, each with its own POM. You can write a root/parent POM through which you can compile all the children (called *modules* in the documentation) with a single command. POMs can also *inherit* configuration from other POMs; things that are inherited are: dependencies, developers and contributors, plugin lists (including reports), plugin executions with matching ids, plugin configuration, and resources. All POMs inherit from the Super POM by default (which contains all the default configuration) unless explicitly set. If you have several Maven projects with similar configurations, you can make a parent project, put the common configuration in it, and have the other projects inherit that.

Instead of having modules inherit from a parent (i.e. the modules specify their `<parent>`), you can also *aggregate* the children in a parent POM (i.e. the parent specifies its `<modules>`). If a Maven command is invoked against the parent project, it will then be executed to the children as well. If you have a group of projects that are built or processed together, you can create a parent project and have that parent project declare those projects as its modules, and then you'd only have to build the parent. To do this, the parent POM's packaging should be `pom` and the directories of it's childrens' POMs must be specified:

```
<modules>
   <module>my-module</module> <!-- Relative path. In this case, the my-module subdirectory is in
       this POM's directory. By convention, the module's artifactId is the directory name -->
</modules>
```

Inheritance is for copying configuration, while aggregation is for batch building. You can have both inheritance and aggregation, i.e. have your modules specify a parent project while having that parent project specify those children projects as its modules.

The version number can have `-SNAPSHOT` as suffix, which means the code can be unstable or changed. A snapshop version is the development version before and older than the final release version. When releasing, a version of x.y-SNAPSHOT releases as x.y and the development version increments to x.(y+1)-SNAPSHOT.

13.1.2 Filtering

In a resource file, you can have variables with `${<property name>}` syntax that can be replaced at build time (called *filtering*) with a value defined in:

- your `pom.xml`, which are referenced with the `.`-delimited XML elements that define the value, e.g. `${project.version}`. Any field of the POM that is a single value element can be referenced as a variable.[22]. Besides those, there are `${project.project.basedir}`, `${project.baseUri}` and `${maven.build.timestamp}`. You can define custom properties and their values in the `<properties>` element. Properties defined in a parent are inherited in its children (and there is no way to avoid that).
- the user's `settings.xml`, which can be referenced using property names beginning with `settings`, e.g. `${settings.localReposit`
- a property defined in an external properties file, for which you only need to add a reference to this external file in your `pom.xml`.
- a system property, e.g. those built into Java (`java.version` or `ser.home`) or properties defined on the command line using the standard Java `-D` parameter.

If you have both text and binary (e.g. images) resources, it is recommended to have one folder `src/main/resources` (default) for the binary resources and another `src/main/resources-filtered` for the textual resources, and only specify the latter in `<filters>`.

Variables are processed after inheritance, so if a parent project uses a variable that is also in the child, then the child definition will be used.

13.1.3 Dependencies

For each dependency, you'll need to define at least `groupId`, `artifactId`, `version`, and `scope`, except when you want to inherit a dependency from a parent POM, in which case you only need to define `groupId` and `artifactId`. `<dependency>`'s are declared inside the `<dependencies>` element. Child modules inherit `<dependency>`s. You can wrap `<dependencies>` inside a `<dependencyManagement>` element, which can be used to consolidate dependencies and their versions without adding dependencies which are inherited by all children, because `<dependency>`s under `<dependencyManagement>` aren't automatically inherited. They're only included in a child module if they are also specified in that child module's POM's `<dependencies>` section or the parent's `<dependencies>` section (the one not wrapped in a `<dependencyManagement>` element).

`scope` is any of:

- `compile`; the default scope. Dependencies are available in all classpaths of a project and are propagated to dependent projects.
- `provided`; much like compile, but indicates you expect the JDK or a (web) container to provide the dependency at runtime, i.e. it is added to the classpath for compilation and test, but not runtime. Not transitive.
- `runtime`; indicates that the dependency is not required for compilation, but is required for execution, i.e. it is added to the runtime and test classpaths, but not the compile classpath.
- `test`; indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive. Typically this scope is used for test libraries such as JUnit and Mockito. It is also used for non-test libraries such as Apache Commons IO if those libraries are used in unit tests (`src/test/java`) but not in the model code (`src/main/java`).
- `system`; similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always
- `import`; only supported on a dependency of type `pom` in the `<dependencyManagement>` section. It indicates the dependency is to be replaced with the effective list of dependencies in the specified POM's `<dependencyManagement>` section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

Maven will first check the *local repository* aka cache (default is `${user.home}/.m2/repository`), and if it isn't available

[22]You can use `pom` as an alias for the project (root) element, but this is deprecated and discouraged.

there, Maven will download the dependency from a remote repository (default is the Maven 2 Central Repository at `https://repo.maven.apache.org/maven2/`, but you can also use e.g. a company-private repository instead of or in addition to the default). Maven also retrieves all the transitive dependencies and store them in the user's local repository.

For deploying JARs to an external repository, you have to configure the `<repository>` `<url>` in the `pom.xml`'s `<distributionManagement`
and the authentication information for connecting to the repository in `settings.xml`:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    https://maven.apache.org/xsd/settings-1.0.0.xsd">
   <servers>
      <server>
         <id>mycompany-repository</id>
         <username>jvanzyl</username>
         <privateKey>/path/to/identity</privateKey> <!-- Default value is ~/.ssh/id_dsa -->
         <passphrase>my_key_passphrase</passphrase>
      </server>
   </servers>
</settings>
```

The `mvn dependency:tree` command prints a tree representation of your project dependencies.

13.1.4 Plugins

Maven is essentially a plugin execution framework, i.e. most of Maven's functionality is in plugins. The user provides only configuration for the project, while the configurable plugins do the actual work. In general, users should not have to write plugins themselves.[23] Each *plugin* is a collection of *goals* (or tasks) with a general common purpose and that can be executed using the command `mvn [plugin-name]:[goal-name] <parameters...>`, e.g. `mvn compiler:compile`, `mvn surefire:test`, `mvn jar:jar`. There are plugins for building, testing, source control management, running a web server, etc. Some basic plugins are included in every project by default, with sensible default settings.

`<plugin>`'s are declared inside the `<plugins>` element. Similar to dependencies, you can use `<pluginManagement>` to consolidate plugins and their configurations without adding plugins which are inherited by all children. `<plugin>`s inside a `<pluginManagement>` element will only be included in a child module if they are also specified in that child module's POM's `<plugins>` section (or that of the parent), and `<plugin>`s not in `<pluginManagement>` are always transferred to child module(s).

When specifying a plugin, you can use the `<executions>` element to run the same goal multiple times with different configurations, and each of which can be IDed so that during inheritance or the application of profiles you can control whether goal configuration is merged or turned into an additional execution. When multiple executions are given that match a particular phase, they are executed in the order specified in the POM, with default or inherited executions running first.

13.1.5 Lifecycles

Since it's cumbersome to run goal commands separately for each build, Maven introduces *lifecycles*, which are lists of named phases that can be used to give order to goal execution. Each *phase* consists of one or more goals. Conversely, each goal can be bound to zero or more build phases (and any goal not bound to any build phase can always be directly invoked). All goals associated with each of the phases up to and including some specified phase can be executed with `mvn [phase]`. Each type of packaging (specified in the `<packaging>` element in the POM; JAR by default) has a default binding of goals to phases, although those bindings are nearly identical. [24] A list of all phases:

1. `clean`; clean up artifacts created by prior builds by deleting the `target` directory. This is undesirable and is not part of most standard lifecycles.
2. `validate`; validate whether the project is correct and all necessary information is available.
3. `initialize`; initialize build state, e.g. set properties or create directories.
4. `generate-sources`; generate any source code for inclusion in compilation.

[23]In Ant and make, all this has to be specified manually with imperative procedures.

[24]Phases and goals may be executed in sequence interchangeable in one command by listing them after another. Phase executions still execute everything up to and including that phase.

5. `process-sources`; process the source code, for example to filter any values.
6. `generate-resources`; generate resources for inclusion in the package.
7. `process-resources`; copy and process (filter) the resources into the destination directory, ready for packaging. JAR packaging associates the `resources:resources` goal by default.
8. `compile`; compile the source code of the project. Compiled classes are conventionally placed in `${project.basedir}/target/cla` JAR packaging associates the `compiler:compile` goal by default.
9. `process-classes`; post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
10. `generate-test-sources`; generate any test source code for inclusion in compilation.
11. `process-test-sources`; process the test source code, for example to filter any values. JAR packaging associates the `resources:testResources` goal by default.
12. `generate-test-resources`; create resources for testing.
13. `process-test-resources`; copy and process the resources into the test destination directory.
14. `process-test-classes`; post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes.
15. `test-compile`; only compiles the test sources, and doesn't actually execute the tests. JAR packaging associates the `compiler:testCompile` goal by default.
16. `test`; test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed. JAR packaging associates the `surefire:test` goal by default. The surefire plugin (which executes the test) looks for tests contained in files with a particular naming convention, which by default includes `**/*Test.java`, `**/Test*.java`, and `**/*TestCase.java`, and excludes `**/Abstract*Test.java` and `**/Abstract*TestCase.java`.
17. `prepare-package`; perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package.
18. `package`; package the compiled code in its distributable format (usually JAR and usually as `${project.basedir}/target/<artifa` The package will contain generated `pom.xml` and `pom.properties` files (and a `MANIFEST.MF` file if you don't create one yourself) with metadata. JAR packaging associates the `jar:jar` goal by default.
19. `pre-integration-test`; perform actions required before integration tests are executed, e.g. setting up the required environment.
20. `integration-test`; process and deploy the package if necessary into an environment where integration tests can be run.
21. `post-integration-test`; perform actions required after integration tests have been executed, e.g. cleaning up the environment.
22. `verify`; run any checks on the results of integration tests to verify the package is valid and meets quality criteria. If there are no integration tests, the effect is the same as `package`.
23. `install`; install the package into the local repository, for use as a dependency in other projects locally.
24. `deploy`; in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects. JAR packaging associates the `deploy:deploy` goal by default.
25. `site`; generate site documentation for this project. Not part of most standard lifecycles.

The phases named with hyphenated-words are not usually directly called from the command line, because they sequence the build, producing intermediate results that are not useful outside the build.[25]

There are three built-in build lifecycles: default, clean and site. The default lifecycle enable users new to a project to

[25]Code coverage tools such as Jacoco and execution container plugins such as Tomcat, Cargo, and Docker bind goals to the `pre-integration-test` phase to prepare the integration test container environment. These plugins also bind goals to the `post-integration-test` phase to collect coverage statistics or decommission the integration test container. Failsafe and code coverage plugins bind goals to `integration-test` and `verify` phases. The net result is test and coverage reports are available after the `verify` phase. If `integration-test` were to be called from the command line, no reports are generated. Worse is that the integration test container environment is left in a hanging state; the Tomcat webserver or Docker instance is left running, and Maven may not even terminate by itself.

build, test and install quickly with `mvn install`, which builds the project and places its binaries in the local repository.

13.1.6 Profiles

Even though Maven strives for portability as much as possible, sometimes different build environments require different build configurations. Maven supports build *profiles*, which modify the POM at build time and are meant to be used in complementary sets to give equivalent-but-different parameters for a set of target environments. Used properly, profiles can be used while still preserving project portability. For example:

```
<profile>
   <id>no-tests</id>
   <properties>
      <maven.test.skip>true</maven.test.skip>
   </properties>
</profile>
```

Profiles can be defined per project in the POM itself, per user in the user settings (`%USER_HOME%/.m2/settings.xml`), and globally in the global settings (`${maven.home}/conf/settings.xml`). Profiles in the `settings.xml` takes higher priority than profiles in the POM. Profiles can be triggered

- explicitly:
  - using the `-P` command line flag followed by a comma-delimited list of profile IDs to use:

    ```
    mvn groupId:artifactId:goal -P profileOne, profileTwo, ?profileThree
    ```

    These are activated in addition to any profiles which are activated in the other explicit ways. Maven will refuse to activate or deactivate a profile that cannot be resolved. To prevent this, prefix the profile identifier with an `?`, marking it as optional.
  - using the `<activeProfiles>` section in `settings.xml`, which takes a list of `<activeProfile>` elements with a profile-id:

    ```
    <activeProfiles>
       <activeProfile>profileOne</activeProfile>
    </activeProfiles>
    ```

  - by setting the `<activeByDefault>` element to true in the `<profile>` configuration in a POM:

    ```
    <profiles>
       <profile>
       <id>profileOne</id>
          <activation>
             <activeByDefault>true</activeByDefault>
          </activation>
       </profile>
    </profiles>
    ```

    All profiles that are `activeByDefault` in the POM are automatically deactivated when a profile in the POM is activated on the command line or through its activation config.
- implicitly, based on the detected state of the build environment, of which the triggers are specified in the `<activation>` element of the `<profile>` configuration in `<profiles>` in the POM.
  - JDK version:

    ```
    <activation>
       <jdk>1.4</jdk> <!-- Triggers when the version starts with "1.4" -->
       <!-- ...or... -->
       <jdk>[1.3,1.6)</jdk> <!-- Honours versions 1.3, 1.4 and 1.5 -->
    </activation>
    ```

  - OS:

    ```
    <activation>
       <os>
          <name>Windows XP</name>
    ```

```
        <family>Windows</family>
        <arch>x86</arch>
        <version>5.1.2600</version>
    </os>
</activation>
```

These are matched against the Java System properties `os.name`, `os.arch`, `os.version` and the family being derived from those. Each value can be prefixed with `!` to negate the matching. The value for version may be prefixed with `regex:` for regular pattern matching. All given OS conditions must match for the trigger.

  – Properties:

```
<activation>
    <property>
        <name>debug</name> <!-- Triggers if "debug" is defined (with any value) -->
    </property>
    <!-- ...or... -->
    <property>
        <name>!debug</name> <!-- Triggers if "debug" is not defined -->
    </property>
    <!-- ...or... -->
    <property>
        <name>debug</name>
        <value>!true</value> <!-- Triggers if "debug" is not defined or is defined with a
            value which is not "true" -->
    </property>
</activation>
```

- Files

```
<activation>
    <file>
        <exists>path/to/file</exists>
    </file>
    <!-- ...or... -->
    <file>
        <missing>path/to/file</missing>
    </file>
</activation>
```

Different implicit activation types can be combined in one profile. The profile is then only active if all conditions are met. You can't use the same type more than once in one profile.

Profiles specified externally (i.e in `settings.xml` or `profiles.xml`) can only modify the `<repositories>` and `<pluginRepositories>` sections, plus an extra `<properties>` section, because external runtime modifications are not portable, i.e. they will not be distributed when the POM is deployed to the repository system. Profiles specified in POMs can only modify that project and its sub-modules, but they can modify a lot more of those: `<repositories>`,`<pluginRepositories>`, `<dependencies>`, `<plugins>`, `<properties>` (not actually available in the main POM, but used behind the scenes), `<modules>`, `<reports>`, `<reporting>`, `<dependencyManagement>`, `<distributionManagement>`, and a subset of the `<build>` element consisting of: `<defaultGoal>`, `<resources>`, `<testResources>`, `<directory>`, `<finalName>`, `<filters>`, `<pluginManagement>`, `<plugins>`. All profile elements in a POM from active profiles overwrite the global elements with the same name of the POM or extend those in case of collections. In case multiple profiles are active in the same POM or external file, the ones which are defined later take precedence over the ones defined earlier (independent of their profile id and activation order).

There are two main problem areas to keep in mind when using profiles:

- External properties (usually used in plugin configurations), i.e. defined outside the `pom.xml` but not in a corresponding profile inside it.
- The incomplete specification of a natural set of profiles. This happens when you specify a necessary property (e.g. the path of an appserver) only in some profiles, but not all profiles, so that it breaks when using any of the other profiles.

You can use the Maven Help Plugin to tell what profiles are in effect during a build, with `mvn help:active-profiles`, which will give a bulleted list of the id of the profile with an activation property of `env=dev` together with the source where it was declared.

13.1.7 Archetypes

Archetypes in Maven help authors create Maven project templates for users, and provides users with the means to generate parameterized versions of those project templates (i.e. directory structure and `pom.xml`) quickly and easily. Maven provides several Archetype artifacts.

13.2 Gradle

Alternative technologies like Gradle and sbt as build tools do not rely on XML, but keep the key concepts Maven introduced. Most popular Java IDEs support development with Maven through extensions.

14 Testing frameworks

14.1 JUnit

JUnit supports Apache Ant, Apache Maven and Gradle build tools. JUnit 5 is a project and consists of three modules:

- JUnit Platform (`junit-jupiter-engine` module) in a runtime dependency that provides the `TestEngine` implementation required for running tests.
- JUnit Jupiter (`junit-jupiter-api` module) is a compile-time dependency that contains all of the API and annotations required to write test cases (but not necessarily to run them).
- JUnit Vintage (`junit-vintage-engine` module) provides a `TestEngine` for running JUnit 3 and JUnit 4 based tests. If you have a mixed JUnit 4 or JUnit 3 test case along with the latest JUnit 5 test cases, then the application needs backward compatibility in the form of the JUnit Vintage engine.

The `junit5-samples` repository hosts a collection of sample projects with appropriate build scripts (e.g., `build.gradle`, `pom.xml`, etc.).

Almost all core annotations are located in the `org.junit.jupiter.api` package in the `junit-jupiter-api` module. A *container* is any node in the test tree that contains other containers or tests as its children. An example of a container is a *test class*, which is any top-level class, static member class, or `@Nested` class that contains at least one test method. A *test method* is any instance method that is directly annotated or meta-annotated with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`. With the exception of `@Test`, these create a container in the test tree that groups tests or, potentially (for `@TestFactory`), other containers. Test classes must not be abstract and must have a single constructor. A *test* is any node in the test tree that verifies expected behavior when executed (e.g. a `@Test`-annotated method). Each test is executed in a new instance of its encompassing class. This means that non static class variables will be unique per test. Test and lifecycle methods may be declared locally or inherited from superclasses or interfaces. They must not be abstract or private[26] and must not return a value (except `@TestFactory`).

An example of a `Test.class`:

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import org.junit.jupiter.api.*;

class StandardTests {

   @BeforeAll
   static void initAll() {
   }

   @BeforeEach
   void init() {
   }

   @Test
   void succeedingTest() {
   }

   @Test
   void failingTest() {
      fail("a failing test");
```

[26]It is recommended to omit the public modifier for test classes, test methods, and lifecycle methods unless there is a technical reason for doing so, e.g. when a test class is extended by a test class in another package or to simplify testing on the module path when using the Java Module System.

```
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

A common convention (but not a requirement) is to always use the `Test` suffix for test classes.

A `@BeforeEach`/`@AfterEach`-annotated method is executed before/after each test method in the current class. A `@BeforeAll`/`@AfterAll`-annotated method is executed before/after all test methods in the current class. Test methods or test classes may be disabled:

- with the `@Disabled` annotation (and an optional though recommended `String` reason).
- annotation-based conditions in the `org.junit.jupiter.api.condition` package that allow developers to enable or disable containers and tests declaratively. When multiple `ExecutionCondition` extensions are registered, a container or test is disabled as soon as one of the conditions returns disabled. Examples are `@EnabledOnOs(value=MAC, architectures="aarch64")`, `@DisabledOnOs({ LINUX, WINDOWS })`, `@EnabledOnJre(JAVA_8)`, `@DisabledForJreRange(min = JAVA_9)`, `@EnabledInNativeImage`, `@EnabledIfSystemProperty(named="os.arch", matches=".*64.*")`, `@EnabledIfEnviro` `matches="staging-server")`, `@EnabledIf("customCondition")` (where `customCondition` is a boolean-returning, parameterless function.
- via a custom `ExecutionCondition`.

Test classes and methods can be tagged via the `@Tag` annotation, e.g. `@Tag("fast")`, which can be used to filter and search.

14.1.1 Display names

You can customize how test names are displayed in test reports and by test runners and IDEs, and those names may contain spaces, special characters, and even emoji. There are four ways, in order of precedence:

1. Value of the @DisplayName annotation, if present, e.g.

   ```
   @Test
   @DisplayName("Display name")
   void displayNameTest() {
   }
   ```

2. By calling the `DisplayNameGenerator` specified in the `@DisplayNameGeneration` annotation, if present. You can create your own generator by implementing the `DisplayNameGenerator` interface, but it comes with four built-in, nested implementations/classes:

   - `Standard`; matches the standard display name generation behavior in place since JUnit Jupiter 5.0 was released.
   - `Simple`; removes trailing parentheses for methods with no parameters.
   - `ReplaceUnderscores`; replaces underscores with spaces.
   - `IndicativeSentences`; generates complete sentences by concatenating the names of the test and the enclosing classes.

3. By calling the default `DisplayNameGenerator` configured via the `junit.jupiter.displayname.generator.default` configuration parameter, if present, e.g., in `src/test/resources/junit-platform.properties`: `junit.jupiter.displayname.g` `= org.junit.jupiter.api.DisplayNameGenerator$ReplaceUnderscores`.

4. By calling `org.junit.jupiter.api.DisplayNameGenerator.Standard`.

14.1.2 Asserts

All JUnit Jupiter assertions are static methods of the form `assert*()` in the `org.junit.jupiter.api.Assertions` class, and are used to define the postconditions for a test case. When an assertion fails, the test case is considered to have failed, and the test execution will stop at that point. Assertions are used to check that the actual result of a test matches the expected result. Most assertions will throw an `AssertionFailedError` if the assertion fails.

For supplying error messages, every type of assertion and overloading of parameter types also has an overloading that takes a `String message` as last parameter and one that takes a `Supplier<String> messageSupplier` as last parameter. `assertEquals()`, `assertTrue()` and `assertNotNull()` are self-explanatory. `assertAll(Executable... executables)` reports all failures together. `assertThrows()` assumes that the specified `Executable` thows the specified `Exception`, e.g. `assertThrows(ArithmeticException.class, ()-> divide(1, 0));` The various `assertTimeoutPreemptively()` methods execute the provided executable or supplier in a different thread than the calling code, which can lead to undesirable side effects if the code that is executed within the executable or supplier relies on `java.lang.ThreadLocal` storage.

JUnit supports the use of third-party assertion libraries such as AssertJ, Hamcrest, Truth, etc., e.g. when more power and additional functionality such as matchers are needed.

14.1.3 Assumptions

All JUnit Jupiter assumptions are static methods of the form `assume*()` in the `org.junit.jupiter.api.Assumptions` class, and are used to define the preconditions for a test case. If the assumption is not met, the test case is skipped/ignored rather than failed. Assumptions are useful when the test case depends on specific conditions, and you want to prevent the test from running if those conditions are not met. Like asserts, every type of assumption and overloading of parameter types also has an overloading that takes a `String message` as last parameter and one that takes a `Supplier<String> messageSupplier` as last parameter. There's `assumeTrue(boolean assumption)`, `assumeFalse()`, `abort()`, and `assumingThat(boolean assumption, Executable executable)`, which executes the `Executable` only if the boolean is true. There are also overloads which replace every `boolean` with a `BooleanSupplier`.

14.1.4 Execution order

By default, test classes and methods will be ordered using an algorithm that is deterministic but intentionally nonobvious. To control the order in which test methods are executed, annotate your test class or test interface with `@TestMethodOrder` and specify the desired `MethodOrderer` implementation. You can implement your own custom `MethodOrderer` or use one of the following built-in ones:

- `MethodOrderer.DisplayName`; sorts test methods alphanumerically based on their display names.
- `MethodOrderer.MethodName`; sorts test methods alphanumerically based on their names and formal parameter lists.
- `MethodOrderer.OrderAnnotation`; sorts test methods numerically based on values specified via the `@Order` annotation; i.e. you number your test methods, starting with `@Order(1)`.
- `MethodOrderer.Random`; orders test methods pseudo-randomly and supports configuration of a custom seed.

You can set the default with the `junit.jupiter.testmethod.order.default` configuration parameter.
All this also goes for test classes; with all mentions of "method" replaced with "class".

14.2 Mockito

Mockito allows the easy creation of test double objects (mock objects) for unit testing, and works with JUnit. A *mock object* is an object that imitates and simulates the behavior of a production object in limited ways, useful when a real object is impractical or impossible to incorporate into a unit test. Mock objects have the same interface as the real objects they mimic. The simplest form returns pre-arranged responses (and is commonly called a *stub*[27], as in a method stub) and the most complex form imitates a production object's complete logic. The mock object knows in advance what is

[27] The definitions of mock, fake, and stub are not consistent across the literature.

supposed to happen during the test (e.g. which of its methods calls will be invoked, etc.) and how it's supposed to react (e.g. what return value to provide). Mockito mocks interfaces by using reflection. Once created, a mock will remember all interactions. Then you can selectively verify whatever interactions you are interested in. Don't mock types you don't own, don't mock value objects, and don't mock everything. For example, you really don't need to mock data classes or classes with a lot of generated code (through annotations, for example). After creating a mock, you can potentially define stubs on the mock, then execute the test – during which interactions with/method calls on the mock are recorded, and lastly these interactions/method calls can potentially be verified.

The `@ExtendWith` annotation in JUnit 5 is used to register extensions.[28] Mockito provides a `MockitoExtension` class in the `mockito-junit-jupiter` dependency that can be used with this annotation to enable Mockito annotations and simplify the creation of mocks.

14.2.1 Creation

```
import org.junit.jupiter.api.*;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
class TestClass {
   List mockedList = mock(List.class); // Mocking an interface
   // or, with Mockito 4.10+:
   List mockedList = mock();
   // or
   @Mock List mockedList;
}
```

Because JUnit executes each `@Test`-annotated test method in a test class in a new instance of that class, every test gets fresh mocks with no prior interactions (given that they're non-static `@Mock` or `@Spy`-annotated instance fields). You can reset a mock within a test with `reset(mock)`, but using this method could be an indication of poor testing and testing too much in one test.

If you annotate a field with `@InjectMocks`, Mockito will create (an object of the type of) it by automatically injecting all the relevant mocks (created by `@Mock` and `@Spy`) into it. In other words, `@Mock` and `@InjectMocks` are complimentary:

```
class Game {
   private Player player;

   public Game(Player player) {
      this.player = player;
   }

   public String attack() {
      return "Player attack with: " + player.getWeapon();
   }
}

class Player {
   private String weapon;

   public Player(String weapon) {
      this.weapon = weapon;
   }

   String getWeapon() {
      return weapon;
   }
}

@RunWith(MockitoJUnitRunner.class)
class GameTest {
   @Mock Player player;
   @InjectMocks Game game;

   @Test
   public void attackWithSwordTest() throws Exception {
      Mockito.when(player.getWeapon()).thenReturn("Sword"); // Mock a Player
      assertEquals("Player attack with: Sword", game.attack()); // Create a Game and put the Player
```

[28] In JUnit 4 and earlier, the equavalent is `@RunWith`.

```
            mock into it
    }
}
```

Generally, `@InjectMocks` is used on the field with as type the one that corresponds to the current test class, like above.

14.2.2 Stubbing

Stubbing is the specification of behavior of a mocked object when a particular method is invoked.

```
LinkedList mockedList = mock(LinkedList.class); // Mocking a (concrete) class

when(mockedList.get(0)).thenReturn("first"); // Specify that get(0) should return "first"
when(mockedList.get(1)).thenThrow(new RuntimeException());

System.out.println(mockedList.get(0)); // Prints "first"
System.out.println(mockedList.get(1)); // Throws RuntimeException
System.out.println(mockedList.get(999)); // Prints null

verify(mockedList).get(0); // Although it is possible to verify a stubbed invocation, usually it's
     just redundant
```

By default, unstubbed value-returning methods will return either `null`, a primitive/primitive wrapper value (e.g. `0` for `int` and `Integer`), or an empty collection. Stubbing can be overridden, so you can have a default setup, but have test methods override it (but this is a potential code smell that points out too much stubbing).

For both stubbing and verification, arguments in method calls to mock objects can be argument matchers: `anyInt()`, `anyBoolean()`, `anyString()`, `isNull()`, etc. If you are using argument matchers in a call, all arguments in that call have to be provided by matchers, but you can use `eq("another argument")`, because `eq()` is also a matcher.

You can also chain stubbing specifications for consecutive calls to the same method invocation:

```
when(mock.someMethod("some arg"))
.thenThrow(new RuntimeException())
.thenReturn("foo");

mock.someMethod("some arg"); // First call throws runtime exception:
System.out.println(mock.someMethod("some arg")); // Second call prints "foo"
System.out.println(mock.someMethod("some arg")); // Any consecutive call prints "foo" as well (last
     stubbing wins)
```

or shorter: `when(mock.someMethod("some arg")).thenReturn("one", "two", "three");`

`doReturn()`, `doThrow()`, `doAnswer()`, `doNothing()`, `doCallRealMethod()` should be used (rather than the `then...()` family) when you stub void methods (because the Java compiler does not like void methods inside brackets), stub methods on spy objects (sometimes it's impossible or impractical to use `when()` for stubbing spies, e.g. when the real method would throw an exception), or stub the same method more than once (to change the behaviour of a mock in the middle of a test) `doThrow(new RuntimeException()).when(mockedList).clear();`. Beware that `when(Object)` is always recommended for stubbing, because it is argument type-safe and more readable (especially when stubbing consecutive calls).

14.2.3 Verification

After executing a test scenario, you can verify that methods were called with specific arguments, how many times they were called, and whether they weren't called. This example mocks and tests the `List` class (in practice, you'd use a real `List`, but this is just for demonstration):

```
// Execution. Interactions with the mock are recorded
mockedList.add("one");
mockedList.clear();

// Verifications. Check methods were called with given arguments.
verify(mockedList).add("one");
verify(mockedList).clear();
```

You can perform verifications on the number of method invocations:

```
mockedList.add("twice");
mockedList.add("twice");
verify(mockedList, times(2)).add("twice");
```

`times(1)` is the default in `verify()` and can be omitted. `never()` is an alias for `times(0)`. There's also `atMostOnce()`, `atLeastOnce()`, `atLeast(int x)`, `atMost(int x)`.

By default, Mockito doesn't consider the order of the calls. The `InOrder` allows verification in order:

```
InOrder inOrder = inOrder(someMock); // Create inOrder object passing mocks that need to be
    verified in order

inOrder.verify(someMock).add("first"); // Make sure that someMock was first called with argument
    "first"
inOrder.verify(someMock).add("second");


InOrder inOrder = inOrder(firstMock, secondMock); // Create inOrder object passing mocks that need
    to be verified in order

inOrder.verify(firstMock).add("first"); // Make sure that firstMock was called before secondMock
inOrder.verify(secondMock).add("second");
inOrder.verify(secondMock).add("third");
```

`verifyNoMoreInteractions()` (used after all the `verify()` method calls) makes sure everything is verified. If any method verification is still left, it will fail and provide a proper message. Use only when relevant. You can ignore stubs when verifying with `verifyNoMoreInteractions(ignoreStubs(mock, mockTwo));` or `InOrder inOrder = inOrder(ignoreStubs(mock, mockTwo));`

Where `thenReturn()` returns some fixed value, `thenAnswer()` can be used to perform some operation or the value needs to be computed at run time. `thenAnswer()` takes an object of a class that implements the functional interface `org.mockito.stubbing.Answer`, which has method `answer()`, which will be invoked by `thenAnswer()`:

```
when(mock.someMethod(anyString())).thenAnswer(
   new Answer() {
      public Object answer(InvocationOnMock invocation) {
         Object[] args = invocation.getArguments();
         Object mock = invocation.getMock();
         return "called with arguments: " + Arrays.toString(args);
    }
});

System.out.println(mock.someMethod("foo")); // Prints "called with arguments: [foo]"
```

14.2.4 Spies

A *spy* is a wrapped real object instance, on which its real methods are called (with all the side effects that entails), while still tracking every interaction. Spies should be used carefully and occasionally, for example when dealing with legacy code. You can still stub its method, which overrides the real method with the stub. Mockito does not delegate calls to the passed real instance; it actually creates a copy of it, so you won't see any effects on the real instance when calling (unstubbed) methods on the spy. Watch out for final methods. Mockito doesn't mock final methods so the bottom line is: when you spy on real objects + you try to stub a final method = trouble. Also you won't be able to verify those method as well.

```
List list = new LinkedList();
List spy = spy(list);
// or
@Spy List spy
// or
SomeAbstract spy = spy(SomeAbstract.class); // Uses constructor to create instance, useful for
    abstract classes
when(spy.size()).thenReturn(100); // Stub one method, the rest remains real
```

A spy is a *partial mock*. You can also make a mock into a partial mock by specifying select methods to call the real counterpart. As with spies, However, avoid partial mocks for new, test-driven and well-designed code.

```
Foo mock = mock(Foo.class);
when(mock.someMethod()).thenCallRealMethod();
```