Short Circuit

Java

Effective Java
by Joshua Bloch
Third Edition
Up until Chapter 7

Summary by Emiel Bos

1 Differences with C++

Unlike in C++, in Java there are no global functions or variables. All code belongs to a class, and all values are objects (except for a couple of primitive types). There is no operator overloading, there are no unsigned integer types, and things like pointers and references are abstracted away. There is automatic garbage collection. Unlike C++, classes can only inherit from one class. Bitwise right shift operator has a signed and unsigned version version. The signed version is the regular `>>` operator and which fills the new leftmost bit with the sign (the old 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 `~`. Java's syntax is mostly derived from C/C++. 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.

2 Programs

Even though it is not enforced by the compiler, each `.java` file should contain at most one top-level 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 `.class` file (still 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) {

}
```

The last argument of a method may alternatively be declared as a variable arity, or varargs, parameter, by adding `...` after the type, in which case the method becomes a variable arity, or 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. The main method can therefore also have `String... args` as parameter. Varargs are simply implemented as arrays under the hood.

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.

3 Ecosystem

The JVM + the Java Class Library (JCL; Java's standard library and analogue to C++'s standard library) is packed into the Java Runtime Environment (JRE). The JRE + development tools (javac, javadoc, profilers, etc.) are packed into the

Java Development Kit (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, and can be imported with the `import` keyword (after any `package` declarations): `package`/`import <packagename>[.<subpackagename>....<subpackagename>].<classname>`. A class's full/official/technical name includes its package name. You can use a class without `import`, but then you must refer to it by its full name. The directory path of the file `classname.java` must correspond to the (full) package 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.

Third party libraries are typically packaged as Java Archive (JAR) files. 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`. 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.

4 Exception handling

try-catch-finally blocks looks like this:

```
try {
   methodThrowingExceptions(); // Statements that may throw exceptions
} catch (IOException | IllegalArgumentException ex) {
   reportException(ex); // Both IOException and IllegalArgumentException will be caught and handled
       here. 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
}
```

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. Resources must implement
    \lstinline{AutoClosable}.
   xEnc.writeObject(object);
} catch (IOException ex) {
   Logger.getLogger(Serializer.class.getName()).log(Level.SEVERE, null, ex);
}
```

You can throw exception 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 `Error` (represents serious problems/abnormal conditions that a reasonable

application should not try to catch) and `Exception` (which any reasonable application might want to catch).

Checked exceptions are checked at compile time, and code calling methods throwing such exceptions must either handle the exception or it must specify the exception using the `throws` keyword. 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. *Unchecked exceptions* are not checked at compile time and are not required to be caught or declared in a throws clause, e.g. `IndexOutOfBoundsException`, `ArithmeticException`, etc. In C++, where all exceptions are unchecked, where it is up to the programmers to specify or catch the exceptions. In Java, exceptions under `Error` and `RuntimeException` (a direct subclass of `Exception`) classes are unchecked exceptions, everything else under `Throwable` is checked.

5 Primitives

Java has the following primitives that can be cast/converted: `byte`, `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 is safe for ASCII characters. Each primitive has a wrapper/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.

6 Reference types

Reference types include class types, interface types, and array types. Primitives can't be `null`, reference types (e.g. boxed primitives) can. Java 10 introduced `var` as a type specifier for inferring types automatically, so it's like C++'s `auto`.

6.1 Classes

`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:

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

Make each class or member as inaccessible as possible [Item 15]. `public` en 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 a adequate job of describing the abstraction provided by the class), however, because client code is confined to the same package. `abstract` specifies that a class only serves as a base class and cannot be instantiated (i.e. it may have `abstract` methods), `final` specifies that it cannot be extended from and cannot have any subclasses. Methods also have keywords/modifiers: `abstract` for methods with no implementation, `final` for methods that can't be overridden (methods are by default virtual, unlike C++), `static` fields aren't tied to an instance, `native` for methods implemented through JNI 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 concrete method (methods in an interface are by default abstract). It 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 and are usually overloaded instead. Fields also 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.

Primitive values are stored directly in variables. Instantiations of reference types (objects and arrays) are always stored as references in variables. 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.

Classes inherit with the `extends` keyword, and can reference its direct superclass with the `super` keyword. Classes can only inherit from one class. If the superclass does not have a constructor without parameters the subclass must specify in its constructors 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 overridden simply invokes the corresponding method in the component, called *forwarding*. Often, all the forwarding methods are contained in a 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.
On the other hand, you should either design and document for inheritance or prohibit it (using Item 17) [Item 19]. For each public or protected method, the documentation must indicate which overridable (i.e. non`final` plus `public` or `protected`) 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.

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 member fields (which you can inline; very useful), and then executes its own code. So 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 also 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 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, and which lastly returns a new instance of the actual object itself in a `build()` method [Item 2]. Constructors can be private, which is useful for singleton classes (only one instance allowed) classes. Besides a private constructor, you can export a `public static` member (either a `public static` field or a `private static field` and a `public static Singleton getInstance()` method). However, best practice is to implement singletons as single-element enums [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, but you should avoid using it since they're unpredictable, slow and unnecessary [Item 8]! Cleaners are slightly better, but still avoid. 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.

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 classse that happens to be declared inside another class and accessed as `OuterClass.InnerClass`. Does not qualify as an inner class (all the following types are inner classes).

  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 cannot be modified afterwards.
- Anonymous classes; 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 nonstatic context, but even if then they cannot have any static members other than constant variables. You should prefer lambdas [Item 42].
- 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].

Implicit casting for objects happens when the source type extends or implements the target type (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.

Immutable classes are those whose instances cannot be modified. The boxed primitive types are also 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 could have nonfinal fields in which you cache expensive computations the first time they are needed, called *lazy initialization*), ensure that the class can't be extended (subclasses can compromise immutability) (by either making the class `final` or making all of its constructors (package) `private` and adding `public static` factories, which is the most flexible because it allows the use of multiple package-private implementation classes), make all fields `final` and `private`, and 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 an they require no synchronization, they can be shared and 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 converts that data to a string. `StringBuilder` can be used to, well, efficiently built sequences of characters, which can finally be converted to an immutable `String`.

6.2 Interfaces

Interfaces are like abstract classes which contain no fields and usually define a number of 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 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], such as *constant interfaces*, which contain only constants. Interfaces can inherit from other interfaces. Interfaces may have `default` methods (with a default implementation), `private` (called from 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]. The convention is to write any `public static final` field in uppercase. Ensure that `public static final` fields are either primitive values or references to immutable objects (so they can't be arrays, for example) [Item 17].

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*, is 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 a an interface, where the 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").

6.3 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 that exports each enumeration constant as a `public static final` field (written in uppercase letters). All advice for classes naturally hold for enums. It has no constructor and is 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, 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 [Item 34], because enums provide compile-time type safety, are more readable and more powerful. 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 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;

   Planet(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*.
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);
}
```

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.

6.4 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, and `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 container 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 write an unrelated class containing an instance of the first class and provide a "view" method that returns the contained instance (called composition). A good `equals()` method 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 must always be `Object`, or else you overload instead of override. Always override `hashCode()` when you override `equals()`, or your class will violate the gen-

eral 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. `compareTo()` is similar to `equals()`, but should returns a negative integer, zero, or a positive integer as 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. Implementing it indicates that instances if the implementing class have a natural ordering, which allows them to be sorted like `Arrays.sort(arr)`/`Collections.sort(col)` or used in a `TreeSet`/`TreeMap`. Implementing `Comparable` is recommended [Item 14]. `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-extensability 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. A class that implements `Comparable` does not need a comparator anymore. A class that implements `Comparator<T>` represents a comparison function, which can be passed to e.g. `Arrays.sort(arr)`/`Collections.sort(col)`. The `Comparator<T>` interface has a `compare(T o1, T o2)` function, a `reversed()` function 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 (i.e. not `static`) and is instead called on `Comparator`, returning a comparator that first applies the original 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 `compareTo()`. '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. (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 on a superclass. The `Cloneable` interface and `Object.clone()` are an obvious case of object-oriented design gone wrong.) 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)` are true, 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. To achieve this independence, 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 not and should not, 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 `clone()`d as well (and they

may not even be enough for a deep copy). `clone()` functions as a 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`. A notable exception to this rule is arrays, which are best copied with `clone()`.

6.5 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. `java.util.Arrays` is a JCL class containing various 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 elements still refer to the same things. Use `int[] listB = Arrays.copyOf(listA, listA.length);` to make a deep copy. Arrays can be created in any of the following ways:

```
int[] numbers = new int[5]; // Initializes all elements to the default value (`boolean`:
    \lstinline{false}, \lstinline{int`: \lstinline{0}, \lstinline{double`: \lstinline{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
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];
```

Arrays are *covariant* – if `SubClass` is a subtype of `SuperClass`, then `Sub[]` is a subtype of `Super[]` – and *reified*[1] – they know and enforce their element type at runtime. These are disadvantages, as it can lead to runtime exceptions as `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; type information is erased at runtime). 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[]`).

6.6 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 (i.e. without angle brackets). 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[2], because the compiler can't catch type errors, and you instead risk `ClassCastException`s at runtime [Item 26]. 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`, you can't pass it to a parameter

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

[2] Exceptions are the use raw types in class literals (`List.class`, `String[].class`, and `int.class` are all legal, but `List<String>.class` and `List<?>.class`) 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
   ...
}
```

of type `List<Object>`). When using generic `Collection`s, the compiler inserts invisible casts 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 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.

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), (and else it is *unbounded*). 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, with most often happens in connection with the `Comparable<T>` interface, e.g. `String implements Comparable<String>`, or:

```
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<?>`. It is legal, though rarely useful, to create arrays of unbounded wildcard types.
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 parameterizarions of the generic type `Generic`. For this reason, use bounded wildcard types on generic/parameterized input parameters (of a function) that represent producers (use `extends`) or consumers (use `super`) for maximum API flexibility [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 relations). 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<?>> scheduledFutures` 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 `null`s 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 [Item 27], because the you know that your code is typesafe (i.e. no `ClassCastException`s). 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.

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)).

Even though generics and varargs parameters were both introduced Java 5, they do not play along well. Varargs parameters are actually arrays[3], 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, with the two exceptions 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.

`Collection`s are limited to a fixed number of type parameters per container. 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.[4] 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*.

6.7 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 its elements, and is type parameterized by the element type. 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 constructors 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. 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 `null`s. There isn't really a nice way to declare and initialize an `ArrayList` with a set of preexisting values.
- `Deque<E>`: double-ended queue interface, which you can use as a (LIFO) stack with `push()`/`addFirst()` and `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). Doesn't allow `null`s. Implemented by `ArrayDeque` and `LinkedList`. `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`.
- `LinkedList<E>`: double-linked list implementation of the `List` and `Deque` interfaces. Very similar to `ArrayDeque`, but this also supports adding at and getting arbitrary indices. Permits `null`.

[3] *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.

[4] You might think that you couldn't put anything in it because of the unbounded wildcard type, but the wildcard 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.

- `TreeSet<E>`/`TreeMap<K,V>`: binary search tree (BST) based sets and maps, i.e. the items are unique and sorted.
- `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()`.

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.

7 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[5], 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.

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

`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 parse and retrieve modifiers for all three types of member. `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.

8 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/elements, 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 elements, 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.
  - `@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.

9 Multi-threading

Java has built-in tools for multi-threading. You can use the `synchronize` keywords on blocks of code:

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

or on member functions:

Notes: - Converting a char to int: int i = c - '0'; If you want the index of a letter in the alphabet: int i = c - 'a'; - 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.