CHAPTER 2

Creating and Destroying Objects

THIS chapter concerns creating and destroying objects: when and how to create them, when and how to avoid creating them, how to ensure they are destroyed in a timely manner, and how to manage any cleanup actions that must precede their destruction.

Item 1: Consider static factory methods instead of constructors

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique that should be a part of every programmer's toolkit. A class can provide a public *static factory method*, which is simply a static method that returns an instance of the class. Here's a simple example from `Boolean` (the *boxed primitive* class for `boolean`). This method translates a `boolean` primitive value into a `Boolean` object reference:

```
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Note that a static factory method is not the same as the *Factory Method* pattern from *Design Patterns* [Gamma95]. The static factory method described in this item has no direct equivalent in *Design Patterns*.

A class can provide its clients with static factory methods instead of, or in addition to, public constructors. Providing a static factory method instead of a public constructor has both advantages and disadvantages.

One advantage of static factory methods is that, unlike constructors, they have names. If the parameters to a constructor do not, in and of themselves, describe the object being returned, a static factory with a well-chosen name is easier to use and the resulting client code easier to read. For example, the constructor `BigInteger(int, int, Random)`, which returns a `BigInteger` that is probably prime, would have been better expressed as a static factory method named `BigInteger.probablePrime`. (This method was added in Java 4.)

A class can have only a single constructor with a given signature. Programmers have been known to get around this restriction by providing two constructors whose parameter lists differ only in the order of their parameter types. This is a really bad idea. The user of such an API will never be able to remember which constructor is which and will end up calling the wrong one by mistake. People reading code that uses these constructors will not know what the code does without referring to the class documentation.

Because they have names, static factory methods don't share the restriction discussed in the previous paragraph. In cases where a class seems to require multiple constructors with the same signature, replace the constructors with static factory methods and carefully chosen names to highlight their differences.

A second advantage of static factory methods is that, unlike constructors, they are not required to create a new object each time they're invoked. This allows immutable classes (Item 17) to use preconstructed instances, or to cache instances as they're constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects. The `Boolean.valueOf(boolean)` method illustrates this technique: it *never* creates an object. This technique is similar to the *Flyweight* pattern [Gamma95]. It can greatly improve performance if equivalent objects are requested often, especially if they are expensive to create.

The ability of static factory methods to return the same object from repeated invocations allows classes to maintain strict control over what instances exist at any time. Classes that do this are said to be *instance-controlled.* There are several reasons to write instance-controlled classes. Instance control allows a class to guarantee that it is a singleton (Item 3) or noninstantiable (Item 4). Also, it allows an immutable value class (Item 17) to make the guarantee that no two equal instances exist: `a.equals(b)` if and only if `a == b`. This is the basis of the *Flyweight* pattern [Gamma95]. Enum types (Item 34) provide this guarantee.

A third advantage of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type. This gives you great flexibility in choosing the class of the returned object.

One application of this flexibility is that an API can return objects without making their classes public. Hiding implementation classes in this fashion leads to a very compact API. This technique lends itself to *interface-based frameworks* (Item 20), where interfaces provide natural return types for static factory methods.

Prior to Java 8, interfaces couldn't have static methods. By convention, static factory methods for an interface named `Type` were put in a *noninstantiable companion class* (Item 4) named `Types`. For example, the Java Collections Framework has forty-five utility implementations of its interfaces, providing unmodifiable collections, synchronized collections, and the like. Nearly all of these implementations are exported via static factory methods in one noninstantiable class (`java.util.Collections`). The classes of the returned objects are all nonpublic.

The Collections Framework API is much smaller than it would have been had it exported forty-five separate public classes, one for each convenience implementation. It is not just the *bulk* of the API that is reduced but the *conceptual weight:* the number and difficulty of the concepts that programmers must master in order to use the API. The programmer knows that the returned object has precisely the API specified by its interface, so there is no need to read additional class documentation for the implementation class. Furthermore, using such a static factory method requires the client to refer to the returned object by interface rather than implementation class, which is generally good practice (Item 64).

As of Java 8, the restriction that interfaces cannot contain static methods was eliminated, so there is typically little reason to provide a noninstantiable companion class for an interface. Many public static members that would have been at home in such a class should instead be put in the interface itself. Note, however, that it may still be necessary to put the bulk of the implementation code behind these static methods in a separate package-private class. This is because Java 8 requires all static members of an interface to be public. Java 9 allows private static methods, but static fields and static member classes are still required to be public.

A fourth advantage of static factories is that the class of the returned object can vary from call to call as a function of the input parameters. Any subtype of the declared return type is permissible. The class of the returned object can also vary from release to release.

The `EnumSet` class (Item 36) has no public constructors, only static factories. In the OpenJDK implementation, they return an instance of one of two subclasses, depending on the size of the underlying enum type: if it has sixty-four or fewer elements, as most enum types do, the static factories return a `RegularEnumSet` instance, which is backed by a single `long`; if the enum type has sixty-five or more elements, the factories return a `JumboEnumSet` instance, backed by a `long` array.

The existence of these two implementation classes is invisible to clients. If `RegularEnumSet` ceased to offer performance advantages for small enum types, it could be eliminated from a future release with no ill effects. Similarly, a future release could add a third or fourth implementation of `EnumSet` if it proved beneficial for performance. Clients neither know nor care about the class of the object they get back from the factory; they care only that it is some subclass of `EnumSet`.

A fifth advantage of static factories is that the class of the returned object need not exist when the class containing the method is written. Such flexible static factory methods form the basis of *service provider frameworks*, like the Java Database Connectivity API (JDBC). A service provider framework is a system in which providers implement a service, and the system makes the implementations available to clients, decoupling the clients from the implementations.

There are three essential components in a service provider framework: a *service interface*, which represents an implementation; a *provider registration API*, which providers use to register implementations; and a *service access API*, which clients use to obtain instances of the service. The service access API may allow clients to specify criteria for choosing an implementation. In the absence of such criteria, the API returns an instance of a default implementation, or allows the client to cycle through all available implementations. The service access API is the flexible static factory that forms the basis of the service provider framework.

An optional fourth component of a service provider framework is a *service provider interface*, which describes a factory object that produce instances of the service interface. In the absence of a service provider interface, implementations must be instantiated reflectively (Item 65). In the case of JDBC, `Connection` plays the part of the service interface, `DriverManager.registerDriver` is the provider registration API, `DriverManager.getConnection` is the service access API, and `Driver` is the service provider interface.

There are many variants of the service provider framework pattern. For example, the service access API can return a richer service interface to clients than the one furnished by providers. This is the *Bridge* pattern [Gamma95]. Dependency injection frameworks (Item 5) can be viewed as powerful service providers. Since Java 6, the platform includes a general-purpose service provider framework, `java.util.ServiceLoader`, so you needn't, and generally shouldn't, write your own (Item 59). JDBC doesn't use `ServiceLoader`, as the former predates the latter.

The main limitation of providing only static factory methods is that classes without public or protected constructors cannot be subclassed. For example, it is impossible to subclass any of the convenience implementation classes in the Collections Framework. Arguably this can be a blessing in disguise because it encourages programmers to use composition instead of inheritance (Item 18), and is required for immutable types (Item 17).

A second shortcoming of static factory methods is that they are hard for programmers to find. They do not stand out in API documentation in the way that constructors do, so it can be difficult to figure out how to instantiate a class that provides static factory methods instead of constructors. The Javadoc tool may someday draw attention to static factory methods. In the meantime, you can reduce this problem by drawing attention to static factories in class or interface documentation and by adhering to common naming conventions. Here are some common names for static factory methods. This list is far from exhaustive:

- **`from`**—A *type-conversion method* that takes a single parameter and returns a corresponding instance of this type, for example:

  ```
  Date d = Date.from(instant);
  ```

- **`of`**—An *aggregation method* that takes multiple parameters and returns an instance of this type that incorporates them, for example:

  ```
  Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
  ```

- **`valueOf`**—A more verbose alternative to `from` and `of`, for example:

  ```
  BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
  ```

- **`instance`** or **`getInstance`**—Returns an instance that is described by its parameters (if any) but cannot be said to have the same value, for example:

  ```
  StackWalker luke = StackWalker.getInstance(options);
  ```

- **`create`** or **`newInstance`**—Like `instance` or `getInstance`, except that the method guarantees that each call returns a new instance, for example:

  ```
  Object newArray = Array.newInstance(classObject, arrayLen);
  ```

- **`get`*****Type***—Like `getInstance`, but used if the factory method is in a different class. *Type* is the type of object returned by the factory method, for example:

  ```
  FileStore fs = Files.getFileStore(path);
  ```

- **`new`*****Type***—Like `newInstance`, but used if the factory method is in a different class. *Type* is the type of object returned by the factory method, for example:

  ```
  BufferedReader br = Files.newBufferedReader(path);
  ```

- ***type***—A concise alternative to `get`*Type* and `new`*Type*, for example:

  ```
  List<Complaint> litany = Collections.list(legacyLitany);
  ```

In summary, static factory methods and public constructors both have their uses, and it pays to understand their relative merits. Often static factories are preferable, so avoid the reflex to provide public constructors without first considering static factories.

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

Static factories and constructors share a limitation: they do not scale well to large numbers of optional parameters. Consider the case of a class representing the Nutrition Facts label that appears on packaged foods. These labels have a few required fields—serving size, servings per container, and calories per serving—and more than twenty optional fields—total fat, saturated fat, trans fat, cholesterol, sodium, and so on. Most products have nonzero values for only a few of these optional fields.

What sort of constructors or static factories should you write for such a class? Traditionally, programmers have used the *telescoping constructor* pattern, in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters. Here's how it looks in practice. For brevity's sake, only four optional fields are shown:

```
// Telescoping constructor pattern - does not scale well!
public class NutritionFacts {
    private final int servingSize;  // (mL)            required
    private final int servings;     // (per container) required
    private final int calories;     // (per serving)   optional
    private final int fat;          // (g/serving)     optional
    private final int sodium;       // (mg/serving)    optional
    private final int carbohydrate; // (g/serving)     optional

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
    public NutritionFacts(int servingSize, int servings,
           int calories, int fat, int sodium, int carbohydrate) {
        this.servingSize  = servingSize;
        this.servings     = servings;
        this.calories     = calories;
        this.fat          = fat;
        this.sodium       = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

When you want to create an instance, you use the constructor with the shortest parameter list containing all the parameters you want to set:

```
NutritionFacts cocaCola =
    new NutritionFacts(240, 8, 100, 0, 35, 27);
```

Typically this constructor invocation will require many parameters that you don't want to set, but you're forced to pass a value for them anyway. In this case, we passed a value of `0` for `fat`. With "only" six parameters this may not seem so bad, but it quickly gets out of hand as the number of parameters increases.

In short, **the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.** The reader is left wondering what all those values mean and must carefully count parameters to find out. Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won't complain, but the program will misbehave at runtime (Item 51).

A second alternative when you're faced with many optional parameters in a constructor is the *JavaBeans* pattern, in which you call a parameterless constructor to create the object and then call setter methods to set each required parameter and each optional parameter of interest:

```
// JavaBeans Pattern - allows inconsistency, mandates mutability
public class NutritionFacts {
    // Parameters initialized to default values (if any)
    private int servingSize  = -1; // Required; no default value
    private int servings     = -1; // Required; no default value
    private int calories     = 0;
    private int fat          = 0;
    private int sodium       = 0;
    private int carbohydrate = 0;

    public NutritionFacts() { }
    // Setters
    public void setServingSize(int val)  { servingSize = val; }
    public void setServings(int val)     { servings = val; }
    public void setCalories(int val)     { calories = val; }
    public void setFat(int val)          { fat = val; }
    public void setSodium(int val)       { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

This pattern has none of the disadvantages of the telescoping constructor pattern. It is easy, if a bit wordy, to create instances, and easy to read the resulting code:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, **a JavaBean may be in an inconsistent state partway through its construction.** The class does not have the option of enforcing consistency merely by checking the validity of the constructor parameters. Attempting to use an object when it's in an inconsistent state may cause failures that are far removed from the code containing the bug and hence difficult to debug. A related disadvantage is that **the JavaBeans pattern precludes the possibility of making a class immutable** (Item 17) and requires added effort on the part of the programmer to ensure thread safety.

It is possible to reduce these disadvantages by manually "freezing" the object when its construction is complete and not allowing it to be used until frozen, but this variant is unwieldy and rarely used in practice. Moreover, it can cause errors at runtime because the compiler cannot ensure that the programmer calls the freeze method on an object before using it.

Luckily, there is a third alternative that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern. It is a form of the *Builder* pattern [Gamma95]. Instead of making the desired object directly, the client calls a constructor (or static factory) with all of the required parameters and gets a *builder object*. Then the client calls setter-like methods on the builder object to set each optional parameter of interest. Finally, the client calls a parameterless `build` method to generate the object, which is typically immutable. The builder is typically a static member class (Item 24) of the class it builds. Here's how it looks in practice:

```
// Builder Pattern
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;

        // Optional parameters - initialized to default values
        private int calories      = 0;
        private int fat           = 0;
        private int sodium        = 0;
        private int carbohydrate  = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings    = servings;
        }

        public Builder calories(int val)
            { calories = val;      return this; }
        public Builder fat(int val)
            { fat = val;           return this; }
        public Builder sodium(int val)
            { sodium = val;        return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = val;  return this; }

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

The NutritionFacts class is immutable, and all parameter default values are in one place. The builder's setter methods return the builder itself so that invocations can be chained, resulting in a *fluent API*. Here's how the client code looks:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
        .calories(100).sodium(35).carbohydrate(27).build();
```

This client code is easy to write and, more importantly, easy to read. **The Builder pattern simulates named optional parameters** as found in Python and Scala.

Validity checks were omitted for brevity. To detect invalid parameters as soon as possible, check parameter validity in the builder's constructor and methods. Check invariants involving multiple parameters in the constructor invoked by the build method. To ensure these invariants against attack, do the checks on object fields after copying parameters from the builder (Item 50). If a check fails, throw an IllegalArgumentException (Item 72) whose detail message indicates which parameters are invalid (Item 75).

The Builder pattern is well suited to class hierarchies. Use a parallel hierarchy of builders, each nested in the corresponding class. Abstract classes have abstract builders; concrete classes have concrete builders. For example, consider an abstract class at the root of a hierarchy representing various kinds of pizza:

```
// Builder pattern for class hierarchies
public abstract class Pizza {
   public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
   final Set<Topping> toppings;

   abstract static class Builder<T extends Builder<T>> {
      EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
      public T addTopping(Topping topping) {
         toppings.add(Objects.requireNonNull(topping));
         return self();
      }

      abstract Pizza build();

      // Subclasses must override this method to return "this"
      protected abstract T self();
   }
   Pizza(Builder<?> builder) {
      toppings = builder.toppings.clone(); // See Item 50
   }
}
```

Note that Pizza.Builder is a *generic type* with a *recursive type parameter* (Item 30). This, along with the abstract self method, allows method chaining to work properly in subclasses, without the need for casts. This workaround for the fact that Java lacks a self type is known as the *simulated self-type* idiom.

Here are two concrete subclasses of `Pizza`, one of which represents a standard New-York-style pizza, the other a calzone. The former has a required size parameter, while the latter lets you specify whether sauce should be inside or out:

```
public class NyPizza extends Pizza {
    public enum Size { SMALL, MEDIUM, LARGE }
    private final Size size;

    public static class Builder extends Pizza.Builder<Builder> {
        private final Size size;

        public Builder(Size size) {
            this.size = Objects.requireNonNull(size);
        }

        @Override public NyPizza build() {
            return new NyPizza(this);
        }

        @Override protected Builder self() { return this; }
    }

    private NyPizza(Builder builder) {
        super(builder);
        size = builder.size;
    }
}

public class Calzone extends Pizza {
    private final boolean sauceInside;

    public static class Builder extends Pizza.Builder<Builder> {
        private boolean sauceInside = false; // Default

        public Builder sauceInside() {
            sauceInside = true;
            return this;
        }

        @Override public Calzone build() {
            return new Calzone(this);
        }

        @Override protected Builder self() { return this; }
    }

    private Calzone(Builder builder) {
        super(builder);
        sauceInside = builder.sauceInside;
    }
}
```

Note that the `build` method in each subclass's builder is declared to return the correct subclass: the `build` method of `NyPizza.Builder` returns `NyPizza`, while the one in `Calzone.Builder` returns `Calzone`. This technique, wherein a subclass method is declared to return a subtype of the return type declared in the superclass, is known as *covariant return typing*. It allows clients to use these builders without the need for casting.

The client code for these "hierarchical builders" is essentially identical to the code for the simple `NutritionFacts` builder. The example client code shown next assumes static imports on enum constants for brevity:

```
NyPizza pizza = new NyPizza.Builder(SMALL)
        .addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
        .addTopping(HAM).sauceInside().build();
```

A minor advantage of builders over constructors is that builders can have multiple varargs parameters because each parameter is specified in its own method. Alternatively, builders can aggregate the parameters passed into multiple calls to a method into a single field, as demonstrated in the `addTopping` method earlier.

The Builder pattern is quite flexible. A single builder can be used repeatedly to build multiple objects. The parameters of the builder can be tweaked between invocations of the `build` method to vary the objects that are created. A builder can fill in some fields automatically upon object creation, such as a serial number that increases each time an object is created.

The Builder pattern has disadvantages as well. In order to create an object, you must first create its builder. While the cost of creating this builder is unlikely to be noticeable in practice, it could be a problem in performance-critical situations. Also, the Builder pattern is more verbose than the telescoping constructor pattern, so it should be used only if there are enough parameters to make it worthwhile, say four or more. But keep in mind that you may want to add more parameters in the future. But if you start out with constructors or static factories and switch to a builder when the class evolves to the point where the number of parameters gets out of hand, the obsolete constructors or static factories will stick out like a sore thumb. Therefore, it's often better to start with a builder in the first place.

In summary, **the Builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters**, especially if many of the parameters are optional or of identical type. Client code is much easier to read and write with builders than with telescoping constructors, and builders are much safer than JavaBeans.

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

A *singleton* is simply a class that is instantiated exactly once [Gamma95]. Singletons typically represent either a stateless object such as a function (Item 24) or a system component that is intrinsically unique. **Making a class a singleton can make it difficult to test its clients** because it's impossible to substitute a mock implementation for a singleton unless it implements an interface that serves as its type.

There are two common ways to implement singletons. Both are based on keeping the constructor private and exporting a public static member to provide access to the sole instance. In one approach, the member is a final field:

```
// Singleton with public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }

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

The private constructor is called only once, to initialize the public static final field `Elvis.INSTANCE`. The lack of a public or protected constructor *guarantees* a "monoelvistic" universe: exactly one `Elvis` instance will exist once the `Elvis` class is initialized—no more, no less. Nothing that a client does can change this, with one caveat: a privileged client can invoke the private constructor reflectively (Item 65) with the aid of the `AccessibleObject.setAccessible` method. If you need to defend against this attack, modify the constructor to make it throw an exception if it's asked to create a second instance.

In the second approach to implementing singletons, the public member is a static factory method:

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public static Elvis getInstance() { return INSTANCE; }

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

All calls to `Elvis.getInstance` return the same object reference, and no other `Elvis` instance will ever be created (with the same caveat mentioned earlier).

The main advantage of the public field approach is that the API makes it clear that the class is a singleton: the public static field is final, so it will always contain the same object reference. The second advantage is that it's simpler.

One advantage of the static factory approach is that it gives you the flexibility to change your mind about whether the class is a singleton without changing its API. The factory method returns the sole instance, but it could be modified to return, say, a separate instance for each thread that invokes it. A second advantage is that you can write a *generic singleton factory* if your application requires it (Item 30). A final advantage of using a static factory is that a *method reference* can be used as a supplier, for example `Elvis::instance` is a `Supplier<Elvis>`. Unless one of these advantages is relevant, the public field approach is preferable.

To make a singleton class that uses either of these approaches *serializable* (Chapter 12), it is not sufficient merely to add `implements Serializable` to its declaration. To maintain the singleton guarantee, declare all instance fields `transient` and provide a `readResolve` method (Item 89). Otherwise, each time a serialized instance is deserialized, a new instance will be created, leading, in the case of our example, to spurious `Elvis` sightings. To prevent this from happening, add this `readResolve` method to the `Elvis` class:

```
// readResolve method to preserve singleton property
private Object readResolve() {
     // Return the one true Elvis and let the garbage collector
     // take care of the Elvis impersonator.
    return INSTANCE;
}
```

A third way to implement a singleton is to declare a single-element enum:

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;

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

This approach is similar to the public field approach, but it is more concise, provides the serialization machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialization or reflection attacks. This approach may feel a bit unnatural, but **a single-element enum type is often the best way to implement a singleton**. Note that you can't use this approach if your singleton must extend a superclass other than `Enum` (though you *can* declare an enum to implement interfaces).

Item 4: Enforce noninstantiability with a private constructor

Occasionally you'll want to write a class that is just a grouping of static methods and static fields. Such classes have acquired a bad reputation because some people abuse them to avoid thinking in terms of objects, but they do have valid uses. They can be used to group related methods on primitive values or arrays, in the manner of `java.lang.Math` or `java.util.Arrays`. They can also be used to group static methods, including factories (Item 1), for objects that implement some interface, in the manner of `java.util.Collections`. (As of Java 8, you can also put such methods *in* the interface, assuming it's yours to modify.) Lastly, such classes can be used to group methods on a final class, since you can't put them in a subclass.

Such *utility classes* were not designed to be instantiated: an instance would be nonsensical. In the absence of explicit constructors, however, the compiler provides a public, parameterless *default constructor*. To a user, this constructor is indistinguishable from any other. It is not uncommon to see unintentionally instantiable classes in published APIs.

Attempting to enforce noninstantiability by making a class abstract does not work. The class can be subclassed and the subclass instantiated. Furthermore, it misleads the user into thinking the class was designed for inheritance (Item 19). There is, however, a simple idiom to ensure noninstantiability. A default constructor is generated only if a class contains no explicit constructors, so **a class can be made noninstantiable by including a private constructor**:

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    }
    ...  // Remainder omitted
}
```

Because the explicit constructor is private, it is inaccessible outside the class. The `AssertionError` isn't strictly required, but it provides insurance in case the constructor is accidentally invoked from within the class. It guarantees the class will never be instantiated under any circumstances. This idiom is mildly counterintuitive because the constructor is provided expressly so that it cannot be invoked. It is therefore wise to include a comment, as shown earlier.

As a side effect, this idiom also prevents the class from being subclassed. All constructors must invoke a superclass constructor, explicitly or implicitly, and a subclass would have no accessible superclass constructor to invoke.

Item 5: Prefer dependency injection to hardwiring resources

Many classes depend on one or more underlying resources. For example, a spell checker depends on a dictionary. It is not uncommon to see such classes implemented as static utility classes (Item 4):

```
// Inappropriate use of static utility - inflexible & untestable!
public class SpellChecker {
    private static final Lexicon dictionary = ...;

    private SpellChecker() {} // Noninstantiable

    public static boolean isValid(String word) { ... }
    public static List<String> suggestions(String typo) { ... }
}
```

Similarly, it's not uncommon to see them implemented as singletons (Item 3):

```
// Inappropriate use of singleton - inflexible & untestable!
public class SpellChecker {
    private final Lexicon dictionary = ...;

    private SpellChecker(...) {}
    public static INSTANCE = new SpellChecker(...);

    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

Neither of these approaches is satisfactory, because they assume that there is only one dictionary worth using. In practice, each language has its own dictionary, and special dictionaries are used for special vocabularies. Also, it may be desirable to use a special dictionary for testing. It is wishful thinking to assume that a single dictionary will suffice for all time.

You could try to have `SpellChecker` support multiple dictionaries by making the `dictionary` field nonfinal and adding a method to change the dictionary in an existing spell checker, but this would be awkward, error-prone, and unworkable in a concurrent setting. **Static utility classes and singletons are inappropriate for classes whose behavior is parameterized by an underlying resource.**

What is required is the ability to support multiple instances of the class (in our example, `SpellChecker`), each of which uses the resource desired by the client (in our example, the dictionary). A simple pattern that satisfies this requirement is to **pass the resource into the constructor when creating a new instance**. This is one form of *dependency injection*: the dictionary is a *dependency* of the spell checker and is *injected* into the spell checker when it is created.

```
// Dependency injection provides flexibility and testability
public class SpellChecker {
    private final Lexicon dictionary;

    public SpellChecker(Lexicon dictionary) {
        this.dictionary = Objects.requireNonNull(dictionary);
    }

    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

The dependency injection pattern is so simple that many programmers use it for years without knowing it has a name. While our spell checker example had only a single resource (the dictionary), dependency injection works with an arbitrary number of resources and arbitrary dependency graphs. It preserves immutability (Item 17), so multiple clients can share dependent objects (assuming the clients desire the same underlying resources). Dependency injection is equally applicable to constructors, static factories (Item 1), and builders (Item 2).

A useful variant of the pattern is to pass a resource *factory* to the constructor. A factory is an object that can be called repeatedly to create instances of a type. Such factories embody the *Factory Method* pattern [Gamma95]. The `Supplier<T>` interface, introduced in Java 8, is perfect for representing factories. Methods that take a `Supplier<T>` on input should typically constrain the factory's type parameter using a *bounded wildcard type* (Item 31) to allow the client to pass in a factory that creates any subtype of a specified type. For example, here is a method that makes a mosaic using a client-provided factory to produce each tile:

```
Mosaic create(Supplier<? extends Tile> tileFactory) { ... }
```

Although dependency injection greatly improves flexibility and testability, it can clutter up large projects, which typically contain thousands of dependencies. This clutter can be all but eliminated by using a *dependency injection framework*, such as Dagger [Dagger], Guice [Guice], or Spring [Spring]. The use of these frameworks is beyond the scope of this book, but note that APIs designed for manual dependency injection are trivially adapted for use by these frameworks.

In summary, do not use a singleton or static utility class to implement a class that depends on one or more underlying resources whose behavior affects that of the class, and do not have the class create these resources directly. Instead, pass the resources, or factories to create them, into the constructor (or static factory or builder). This practice, known as dependency injection, will greatly enhance the flexibility, reusability, and testability of a class.

Item 6: Avoid creating unnecessary objects

It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed. Reuse can be both faster and more stylish. An object can always be reused if it is immutable (Item 17).

As an extreme example of what not to do, consider this statement:

```
String s = new String("bikini");  // DON'T DO THIS!
```

The statement creates a new `String` instance each time it is executed, and none of those object creations is necessary. The argument to the `String` constructor (`"bikini"`) is itself a `String` instance, functionally identical to all of the objects created by the constructor. If this usage occurs in a loop or in a frequently invoked method, millions of `String` instances can be created needlessly.

The improved version is simply the following:

```
String s = "bikini";
```

This version uses a single `String` instance, rather than creating a new one each time it is executed. Furthermore, it is guaranteed that the object will be reused by any other code running in the same virtual machine that happens to contain the same string literal [JLS, 3.10.5].

You can often avoid creating unnecessary objects by using *static factory methods* (Item 1) in preference to constructors on immutable classes that provide both. For example, the factory method `Boolean.valueOf(String)` is preferable to the constructor `Boolean(String)`, which was deprecated in Java 9. The constructor *must* create a new object each time it's called, while the factory method is never required to do so and won't in practice. In addition to reusing immutable objects, you can also reuse mutable objects if you know they won't be modified.

Some object creations are much more expensive than others. If you're going to need such an "expensive object" repeatedly, it may be advisable to cache it for reuse. Unfortunately, it's not always obvious when you're creating such an object. Suppose you want to write a method to determine whether a string is a valid Roman numeral. Here's the easiest way to do this using a regular expression:

```
// Performance can be greatly improved!
static boolean isRomanNumeral(String s) {
    return s.matches("^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

The problem with this implementation is that it relies on the `String.matches` method. **While `String.matches` is the easiest way to check if a string matches a regular expression, it's not suitable for repeated use in performance-critical situations.** The problem is that it internally creates a `Pattern` instance for the regular expression and uses it only once, after which it becomes eligible for garbage collection. Creating a `Pattern` instance is expensive because it requires compiling the regular expression into a finite state machine.

To improve the performance, explicitly compile the regular expression into a `Pattern` instance (which is immutable) as part of class initialization, cache it, and reuse the same instance for every invocation of the `isRomanNumeral` method:

```
// Reusing expensive object for improved performance
public class RomanNumerals {
    private static final Pattern ROMAN = Pattern.compile(
            "^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

    static boolean isRomanNumeral(String s) {
        return ROMAN.matcher(s).matches();
    }
}
```

The improved version of `isRomanNumeral` provides significant performance gains if invoked frequently. On my machine, the original version takes 1.1 µs on an 8-character input string, while the improved version takes 0.17 µs, which is 6.5 times faster. Not only is the performance improved, but arguably, so is clarity. Making a static final field for the otherwise invisible `Pattern` instance allows us to give it a name, which is far more readable than the regular expression itself.

If the class containing the improved version of the `isRomanNumeral` method is initialized but the method is never invoked, the field `ROMAN` will be initialized needlessly. It would be possible to eliminate the initialization by *lazily initializing* the field (Item 83) the first time the `isRomanNumeral` method is invoked, but this is *not* recommended. As is often the case with lazy initialization, it would complicate the implementation with no measurable performance improvement (Item 67).

When an object is immutable, it is obvious it can be reused safely, but there are other situations where it is far less obvious, even counterintuitive. Consider the case of *adapters* [Gamma95], also known as *views*. An adapter is an object that delegates to a backing object, providing an alternative interface. Because an adapter has no state beyond that of its backing object, there's no need to create more than one instance of a given adapter to a given object.

For example, the `keySet` method of the `Map` interface returns a `Set` view of the `Map` object, consisting of all the keys in the map. Naively, it would seem that every call to `keySet` would have to create a new `Set` instance, but every call to `keySet` on a given `Map` object may return the same `Set` instance. Although the returned `Set` instance is typically mutable, all of the returned objects are functionally identical: when one of the returned objects changes, so do all the others, because they're all backed by the same `Map` instance. While it is largely harmless to create multiple instances of the `keySet` view object, it is unnecessary and has no benefits.

Another way to create unnecessary objects is *autoboxing*, which allows the programmer to mix primitive and boxed primitive types, boxing and unboxing automatically as needed. **Autoboxing blurs but does not erase the distinction between primitive and boxed primitive types.** There are subtle semantic distinctions and not-so-subtle performance differences (Item 61). Consider the following method, which calculates the sum of all the positive `int` values. To do this, the program has to use `long` arithmetic because an `int` is not big enough to hold the sum of all the positive `int` values:

```
// Hideously slow! Can you spot the object creation?
private static long sum() {
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;

    return sum;
}
```

This program gets the right answer, but it is *much* slower than it should be, due to a one-character typographical error. The variable `sum` is declared as a `Long` instead of a `long`, which means that the program constructs about 2^{31} unnecessary `Long` instances (roughly one for each time the `long i` is added to the `Long sum`). Changing the declaration of `sum` from `Long` to `long` reduces the runtime from 6.3 seconds to 0.59 seconds on my machine. The lesson is clear: **prefer primitives to boxed primitives, and watch out for unintentional autoboxing.**

This item should not be misconstrued to imply that object creation is expensive and should be avoided. On the contrary, the creation and reclamation of small objects whose constructors do little explicit work is cheap, especially on modern JVM implementations. Creating additional objects to enhance the clarity, simplicity, or power of a program is generally a good thing.

Conversely, avoiding object creation by maintaining your own *object pool* is a bad idea unless the objects in the pool are extremely heavyweight. The classic example of an object that *does* justify an object pool is a database connection. The cost of establishing the connection is sufficiently high that it makes sense to reuse these objects. Generally speaking, however, maintaining your own object pools clutters your code, increases memory footprint, and harms performance. Modern JVM implementations have highly optimized garbage collectors that easily outperform such object pools on lightweight objects.

The counterpoint to this item is Item 50 on *defensive copying*. The present item says, “Don’t create a new object when you should reuse an existing one,” while Item 50 says, “Don’t reuse an existing object when you should create a new one.” Note that the penalty for reusing an object when defensive copying is called for is far greater than the penalty for needlessly creating a duplicate object. Failing to make defensive copies where required can lead to insidious bugs and security holes; creating objects unnecessarily merely affects style and performance.

Item 7: Eliminate obsolete object references

If you switched from a language with manual memory management, such as C or C++, to a garbage-collected language such as Java, your job as a programmer was made much easier by the fact that your objects are automatically reclaimed when you're through with them. It seems almost like magic when you first experience it. It can easily lead to the impression that you don't have to think about memory management, but this isn't quite true.

Consider the following simple stack implementation:

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

There's nothing obviously wrong with this program (but see Item 29 for a generic version). You could test it exhaustively, and it would pass every test with flying colors, but there's a problem lurking. Loosely speaking, the program has a "memory leak," which can silently manifest itself as reduced performance due to increased garbage collector activity or increased memory footprint. In extreme cases, such memory leaks can cause disk paging and even program failure with an `OutOfMemoryError`, but such failures are relatively rare.

So where is the memory leak? If a stack grows and then shrinks, the objects that were popped off the stack will not be garbage collected, even if the program using the stack has no more references to them. This is because the stack maintains *obsolete references* to these objects. An obsolete reference is simply a reference that will never be dereferenced again. In this case, any references outside of the "active portion" of the element array are obsolete. The active portion consists of the elements whose index is less than `size`.

Memory leaks in garbage-collected languages (more properly known as *unintentional object retentions*) are insidious. If an object reference is unintentionally retained, not only is that object excluded from garbage collection, but so too are any objects referenced by that object, and so on. Even if only a few object references are unintentionally retained, many, many objects may be prevented from being garbage collected, with potentially large effects on performance.

The fix for this sort of problem is simple: null out references once they become obsolete. In the case of our `Stack` class, the reference to an item becomes obsolete as soon as it's popped off the stack. The corrected version of the `pop` method looks like this:

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

An added benefit of nulling out obsolete references is that if they are subsequently dereferenced by mistake, the program will immediately fail with a `NullPointerException`, rather than quietly doing the wrong thing. It is always beneficial to detect programming errors as quickly as possible.

When programmers are first stung by this problem, they may overcompensate by nulling out every object reference as soon as the program is finished using it. This is neither necessary nor desirable; it clutters up the program unnecessarily. **Nulling out object references should be the exception rather than the norm.** The best way to eliminate an obsolete reference is to let the variable that contained the reference fall out of scope. This occurs naturally if you define each variable in the narrowest possible scope (Item 57).

So when should you null out a reference? What aspect of the `Stack` class makes it susceptible to memory leaks? Simply put, it *manages its own memory.* The *storage pool* consists of the elements of the `elements` array (the object reference cells, not the objects themselves). The elements in the active portion of the array (as defined earlier) are *allocated*, and those in the remainder of the array are *free*. The garbage collector has no way of knowing this; to the garbage collector, all of the object references in the `elements` array are equally valid. Only the programmer knows that the inactive portion of the array is unimportant. The programmer effectively communicates this fact to the garbage collector by manually nulling out array elements as soon as they become part of the inactive portion.

Generally speaking, **whenever a class manages its own memory, the programmer should be alert for memory leaks**. Whenever an element is freed, any object references contained in the element should be nulled out.

Another common source of memory leaks is caches. Once you put an object reference into a cache, it's easy to forget that it's there and leave it in the cache long after it becomes irrelevant. There are several solutions to this problem. If you're lucky enough to implement a cache for which an entry is relevant exactly so long as there are references to its key outside of the cache, represent the cache as a `WeakHashMap`; entries will be removed automatically after they become obsolete. Remember that `WeakHashMap` is useful only if the desired lifetime of cache entries is determined by external references to the key, not the value.

More commonly, the useful lifetime of a cache entry is less well defined, with entries becoming less valuable over time. Under these circumstances, the cache should occasionally be cleansed of entries that have fallen into disuse. This can be done by a background thread (perhaps a `ScheduledThreadPoolExecutor`) or as a side effect of adding new entries to the cache. The `LinkedHashMap` class facilitates the latter approach with its `removeEldestEntry` method. For more sophisticated caches, you may need to use `java.lang.ref` directly.

A third common source of memory leaks is listeners and other callbacks. If you implement an API where clients register callbacks but don't deregister them explicitly, they will accumulate unless you take some action. One way to ensure that callbacks are garbage collected promptly is to store only *weak references* to them, for instance, by storing them only as keys in a `WeakHashMap`.

Because memory leaks typically do not manifest themselves as obvious failures, they may remain present in a system for years. They are typically discovered only as a result of careful code inspection or with the aid of a debugging tool known as a *heap profiler.* Therefore, it is very desirable to learn to anticipate problems like this before they occur and prevent them from happening.

Item 8: Avoid finalizers and cleaners

Finalizers are unpredictable, often dangerous, and generally unnecessary. Their use can cause erratic behavior, poor performance, and portability problems. Finalizers have a few valid uses, which we'll cover later in this item, but as a rule, you should avoid them. As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries. The Java 9 replacement for finalizers is *cleaners*. **Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.**

C++ programmers are cautioned not to think of finalizers or cleaners as Java's analogue of C++ destructors. In C++, destructors are the normal way to reclaim the resources associated with an object, a necessary counterpart to constructors. In Java, the garbage collector reclaims the storage associated with an object when it becomes unreachable, requiring no special effort on the part of the programmer. C++ destructors are also used to reclaim other nonmemory resources. In Java, a `try`-with-resources or `try-finally` block is used for this purpose (Item 9).

One shortcoming of finalizers and cleaners is that there is no guarantee they'll be executed promptly [JLS, 12.6]. It can take arbitrarily long between the time that an object becomes unreachable and the time its finalizer or cleaner runs. This means that you should **never do anything time-critical in a finalizer or cleaner.** For example, it is a grave error to depend on a finalizer or cleaner to close files because open file descriptors are a limited resource. If many files are left open as a result of the system's tardiness in running finalizers or cleaners, a program may fail because it can no longer open files.

The promptness with which finalizers and cleaners are executed is primarily a function of the garbage collection algorithm, which varies widely across implementations. The behavior of a program that depends on the promptness of finalizer or cleaner execution may likewise vary. It is entirely possible that such a program will run perfectly on the JVM on which you test it and then fail miserably on the one favored by your most important customer.

Tardy finalization is not just a theoretical problem. Providing a finalizer for a class can arbitrarily delay reclamation of its instances. A colleague debugged a long-running GUI application that was mysteriously dying with an `OutOfMemoryError`. Analysis revealed that at the time of its death, the application had thousands of graphics objects on its finalizer queue just waiting to be finalized and reclaimed. Unfortunately, the finalizer thread was running at a lower priority than another application thread, so objects weren't getting finalized at the rate they became eligible for finalization. The language specification makes no guarantees as to which thread will execute finalizers, so there is no portable way to prevent this sort of problem other than to refrain from using finalizers. Cleaners are a bit better than finalizers in this regard because class authors have control over their own cleaner threads, but cleaners still run in the background, under the control of the garbage collector, so there can be no guarantee of prompt cleaning.

Not only does the specification provide no guarantee that finalizers or cleaners will run promptly; it provides no guarantee that they'll run at all. It is entirely possible, even likely, that a program terminates without running them on some objects that are no longer reachable. As a consequence, you should **never depend on a finalizer or cleaner to update persistent state.** For example, depending on a finalizer or cleaner to release a persistent lock on a shared resource such as a database is a good way to bring your entire distributed system to a grinding halt.

Don't be seduced by the methods `System.gc` and `System.runFinalization`. They may increase the odds of finalizers or cleaners getting executed, but they don't guarantee it. Two methods once claimed to make this guarantee: `System.runFinalizersOnExit` and its evil twin, `Runtime.runFinalizersOnExit`. These methods are fatally flawed and have been deprecated for decades [ThreadStop].

Another problem with finalizers is that an uncaught exception thrown during finalization is ignored, and finalization of that object terminates [JLS, 12.6]. Uncaught exceptions can leave other objects in a corrupt state. If another thread attempts to use such a corrupted object, arbitrary nondeterministic behavior may result. Normally, an uncaught exception will terminate the thread and print a stack trace, but not if it occurs in a finalizer—it won't even print a warning. Cleaners do not have this problem because a library using a cleaner has control over its thread.

There is a *severe* performance penalty for using finalizers and cleaners. On my machine, the time to create a simple `AutoCloseable` object, to close it using `try`-with-resources, and to have the garbage collector reclaim it is about 12 ns. Using a finalizer instead increases the time to 550 ns. In other words, it is about 50 times slower to create and destroy objects with finalizers. This is primarily because finalizers inhibit efficient garbage collection. Cleaners are comparable in speed to finalizers if you use them to clean all instances of the class (about 500 ns per instance on my machine), but cleaners are much faster if you use them only as a safety net, as discussed below. Under these circumstances, creating, cleaning, and destroying an object takes about 66 ns on my machine, which means you pay a factor of five (not fifty) for the insurance of a safety net *if* you don't use it.

Finalizers have a serious security problem: they open your class up to *finalizer attacks*. The idea behind a finalizer attack is simple: If an exception is thrown from a constructor or its serialization equivalents—the `readObject` and `readResolve` methods (Chapter 12)—the finalizer of a malicious subclass can run on the partially constructed object that should have "died on the vine." This finalizer can record a reference to the object in a static field, preventing it from being garbage collected. Once the malformed object has been recorded, it is a simple matter to invoke arbitrary methods on this object that should never have been allowed to exist in the first place. **Throwing an exception from a constructor should be sufficient to prevent an object from coming into existence; in the presence of finalizers, it is not.** Such attacks can have dire consequences. Final classes are immune to finalizer attacks because no one can write a malicious subclass of a final class. **To protect nonfinal classes from finalizer attacks, write a final `finalize` method that does nothing.**

So what should you do instead of writing a finalizer or cleaner for a class whose objects encapsulate resources that require termination, such as files or threads? Just **have your class implement `AutoCloseable`,** and require its clients to invoke the `close` method on each instance when it is no longer needed, typically using `try`-with-resources to ensure termination even in the face of exceptions (Item 9). One detail worth mentioning is that the instance must keep track of whether it has been closed: the `close` method must record in a field that the object is no longer valid, and other methods must check this field and throw an `IllegalStateException` if they are called after the object has been closed.

So what, if anything, are cleaners and finalizers good for? They have perhaps two legitimate uses. One is to act as a safety net in case the owner of a resource neglects to call its `close` method. While there's no guarantee that the cleaner or finalizer will run promptly (or at all), it is better to free the resource late than never if the client fails to do so. If you're considering writing such a safety-net finalizer, think long and hard about whether the protection is worth the cost. Some Java library classes, such as `FileInputStream`, `FileOutputStream`, `ThreadPoolExecutor`, and `java.sql.Connection`, have finalizers that serve as safety nets.

A second legitimate use of cleaners concerns objects with *native peers*. A native peer is a native (non-Java) object to which a normal object delegates via native methods. Because a native peer is not a normal object, the garbage collector doesn't know about it and can't reclaim it when its Java peer is reclaimed. A cleaner or finalizer may be an appropriate vehicle for this task, assuming the performance is acceptable and the native peer holds no critical resources. If the performance is unacceptable or the native peer holds resources that must be reclaimed promptly, the class should have a `close` method, as described earlier.

Cleaners are a bit tricky to use. Below is a simple `Room` class demonstrating the facility. Let's assume that rooms must be cleaned before they are reclaimed. The `Room` class implements `AutoCloseable`; the fact that its automatic cleaning safety net uses a cleaner is merely an implementation detail. Unlike finalizers, cleaners do not pollute a class's public API:

```
// An autocloseable class using a cleaner as a safety net
public class Room implements AutoCloseable {
    private static final Cleaner cleaner = Cleaner.create();

    // Resource that requires cleaning. Must not refer to Room!
    private static class State implements Runnable {
        int numJunkPiles; // Number of junk piles in this room

        State(int numJunkPiles) {
            this.numJunkPiles = numJunkPiles;
        }

        // Invoked by close method or cleaner
        @Override public void run() {
            System.out.println("Cleaning room");
            numJunkPiles = 0;
        }
    }

    // The state of this room, shared with our cleanable
    private final State state;

    // Our cleanable. Cleans the room when it's eligible for gc
    private final Cleaner.Cleanable cleanable;

    public Room(int numJunkPiles) {
        state = new State(numJunkPiles);
        cleanable = cleaner.register(this, state);
    }

    @Override public void close() {
        cleanable.clean();
    }
}
```

The static nested `State` class holds the resources that are required by the cleaner to clean the room. In this case, it is simply the `numJunkPiles` field, which represents the amount of mess in the room. More realistically, it might be a final `long` that contains a pointer to a native peer. `State` implements `Runnable`, and its `run` method is called at most once, by the `Cleanable` that we get when we register our `State` instance with our cleaner in the `Room` constructor. The call to the `run` method will be triggered by one of two things: Usually it is triggered by a call to

Room's close method calling Cleanable's clean method. If the client fails to call the close method by the time a Room instance is eligible for garbage collection, the cleaner will (hopefully) call State's run method.

It is critical that a State instance does not refer to its Room instance. If it did, it would create a circularity that would prevent the Room instance from becoming eligible for garbage collection (and from being automatically cleaned). Therefore, State must be a *static* nested class because nonstatic nested classes contain references to their enclosing instances (Item 24). It is similarly inadvisable to use a lambda because they can easily capture references to enclosing objects.

As we said earlier, Room's cleaner is used only as a safety net. If clients surround all Room instantiations in try-with-resource blocks, automatic cleaning will never be required. This well-behaved client demonstrates that behavior:

```
public class Adult {
    public static void main(String[] args) {
        try (Room myRoom = new Room(7)) {
            System.out.println("Goodbye");
        }
    }
}
```

As you'd expect, running the Adult program prints Goodbye, followed by Cleaning room. But what about this ill-behaved program, which never cleans its room?

```
public class Teenager {
    public static void main(String[] args) {
        new Room(99);
        System.out.println("Peace out");
    }
}
```

You might expect it to print Peace out, followed by Cleaning room, but on my machine, it never prints Cleaning room; it just exits. This is the unpredictability we spoke of earlier. The Cleaner spec says, "The behavior of cleaners during System.exit is implementation specific. No guarantees are made relating to whether cleaning actions are invoked or not." While the spec does not say it, the same holds true for normal program exit. On my machine, adding the line System.gc() to Teenager's main method is enough to make it print Cleaning room prior to exit, but there's no guarantee that you'll see the same behavior on your machine.

In summary, don't use cleaners, or in releases prior to Java 9, finalizers, except as a safety net or to terminate noncritical native resources. Even then, beware the indeterminacy and performance consequences.

Item 9: Prefer try-with-resources to try-finally

The Java libraries include many resources that must be closed manually by invoking a close method. Examples include InputStream, OutputStream, and java.sql.Connection. Closing resources is often overlooked by clients, with predictably dire performance consequences. While many of these resources use finalizers as a safety net, finalizers don't work very well (Item 8).

Historically, a try-finally statement was the best way to guarantee that a resource would be closed properly, even in the face of an exception or return:

```
// try-finally - No longer the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

This may not look bad, but it gets worse when you add a second resource:

```
// try-finally is ugly when used with more than one resource!
static void copy(String src, String dst) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dst);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
    } finally {
        in.close();
    }
}
```

It may be hard to believe, but even good programmers got this wrong most of the time. For starters, I got it wrong on page 88 of *Java Puzzlers* [Bloch05], and no one noticed for years. In fact, two-thirds of the uses of the close method in the Java libraries were wrong in 2007.

Even the correct code for closing resources with `try-finally` statements, as illustrated in the previous two code examples, has a subtle deficiency. The code in both the `try` block and the `finally` block is capable of throwing exceptions. For example, in the `firstLineOfFile` method, the call to `readLine` could throw an exception due to a failure in the underlying physical device, and the call to `close` could then fail for the same reason. Under these circumstances, the second exception completely obliterates the first one. There is no record of the first exception in the exception stack trace, which can greatly complicate debugging in real systems—usually it's the first exception that you want to see in order to diagnose the problem. While it is possible to write code to suppress the second exception in favor of the first, virtually no one did because it's just too verbose.

All of these problems were solved in one fell swoop when Java 7 introduced the `try`-with-resources statement [JLS, 14.20.3]. To be usable with this construct, a resource must implement the `AutoCloseable` interface, which consists of a single `void`-returning `close` method. Many classes and interfaces in the Java libraries and in third-party libraries now implement or extend `AutoCloseable`. If you write a class that represents a resource that must be closed, your class should implement `AutoCloseable` too.

Here's how our first example looks using `try`-with-resources:

```
// try-with-resources - the the best way to close resources!
static String firstLineOfFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    }
}
```

And here's how our second example looks using `try`-with-resources:

```
// try-with-resources on multiple resources - short and sweet
static void copy(String src, String dst) throws IOException {
    try (InputStream   in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

Not only are the `try`-with-resources versions shorter and more readable than the originals, but they provide far better diagnostics. Consider the `firstLineOfFile` method. If exceptions are thrown by both the `readLine` call and the (invisible) `close`, the latter exception is *suppressed* in favor of the former. In fact, multiple exceptions may be suppressed in order to preserve the exception that you actually want to see. These suppressed exceptions are not merely discarded; they are printed in the stack trace with a notation saying that they were suppressed. You can also access them programmatically with the `getSuppressed` method, which was added to `Throwable` in Java 7.

You can put catch clauses on `try`-with-resources statements, just as you can on regular `try-finally` statements. This allows you to handle exceptions without sullying your code with another layer of nesting. As a slightly contrived example, here's a version our `firstLineOfFile` method that does not throw exceptions, but takes a default value to return if it can't open the file or read from it:

```
// try-with-resources with a catch clause
static String firstLineOfFile(String path, String defaultVal) {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return defaultVal;
    }
}
```

The lesson is clear: Always use `try`-with-resources in preference to `try-finally` when working with resources that must be closed. The resulting code is shorter and clearer, and the exceptions that it generates are more useful. The `try`-with-resources statement makes it easy to write correct code using resources that must be closed, which was practically impossible using `try-finally`.