DZone

# 5 Hidden Secrets in Java

**by Justin Albano MVB · Feb. 21, 18 · Java Zone**



As programming languages grow, it is inevitable that hidden features begin to appear and constructs that were never intended by the founders begin to creep into common usage. Some of these features rear their head as idioms and become accepted parlance in the language, while others become anti-patterns and are relegated to the dark corners of the language community. In this article, we will take a look at five Java secrets that are often overlooked by the large population of Java developers (some for good reason). With each description, we will look at the use cases and rationale that brought each feature into the existence and look at some examples when it may be appropriate to use these features.

The reader should note that not all these features are not truly hidden in the language, but are often unused in daily programming. While some may be very useful at appropriate times, others are almost always a poor idea and are shown in this article to peek the interest of the reader (and possibly give him or her a good laugh). The reader should use his or her judgment when deciding when to use the features described in this article: Just because it can be done does not mean it should.

## 1. Annotation Implementation

Since Java Development Kit (JDK) 5, annotations have an integral part of many Java applications and frameworks. In a vast majority of cases, annotations are applied to language constructs, such as classes, fields, methods, etc., but there is another case in which annotations can be applied: As implementable interfaces. For example, suppose we have the following annotation definition:

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  public @interface Test {
4      String name();
5  }
```

Normally, we would apply this annotation to a method, as in the following:

```
1  public class MyTestFixure {
2
3      @Test
```

```
    public void givenFooWhenBarThenBaz() {
        // ...
    }
}
```

We can then process this annotation, as described in Creating Annotations in Java. If we also wanted to create an interface that allows for tests to be created as objects, we would have to create a new interface, naming it something other than `Test`:

```
public interface TestInstance {
    public String getName();
}
```

Then we could instantiate a `TestInstance` object:

```
public class FooTestInstance implements TestInstance {

    @Override
    public String getName() {
        return "Foo";
    }
}

TestInstance myTest = new FooTestInstance();
```

While our annotation and interface are nearly identical, with very noticeable duplication, there does not appear to be a way to merge these two constructs. Fortunately, looks are deceiving and there is a technique for merging these two constructs: Implement the annotation:

```
public class FooTest implements Test {

    @Override
    public String name() {
        return "Foo";
    }

    @Override
    public Class<? extends Annotation> annotationType() {
        return Test.class;
    }
}
```

Note that we must implement the `annotationType` method and return the type of the annotation as well, since this is implicitly part of the `Annotation` interface. Although in nearly every case, implementing an annotation is not a sound design decision (the Java compiler will show a warning when implementing an interface), it can be useful in a select few circumstances, such as within annotation-driven frameworks.

# 2. Instance Initialization

In Java, as with most object-oriented programming languages, objects are exclusively instantiated using a constructor (with some critical exceptions, such as Java object deserialization). Even when we create static factory methods to create objects, we are simply wrapping a call to the constructor of an object to instantiate it. For example:

```
public class Foo {

   private final String name;

   private Foo(String name) {
       this.name = name;
   }

   public static Foo withName(String name) {
       return new Foo(name);
   }
}

Foo foo = Foo.withName("Bar");
```

Therefore, when we wish to initialize an object, we consolidate the initialization logic into the constructor of the object. For example, we set the `name` field of the `Foo` class within its parameterized constructor. While it may appear to be a sound assumption that *all* of the initialization logic is found in the constructor or set of constructors for a class, this is not the case in Java. Instead, we can also use instance initialization to execute code when an object is created:

```
public class Foo {

    {
        System.out.println("Foo:instance 1");
    }

    public Foo() {
        System.out.println("Foo:constructor");
    }
```

```
}
```

Instance initializers are specified by adding initialization logic within a set of braces within the definition of a class. When the object is instantiated, its instance initializers are called first, followed by its constructors. Note that more than one instance initializer may be specified, in which case, each is called in the order it appears within the class definition. Apart from instance initializers, we can also create static initializers, which are executed when the class is loaded into memory. To create a static initializer, we simply prefix an initializer with the keyword `static`:

```
public class Foo {

    {
        System.out.println("Foo:instance 1");
    }

    static {
        System.out.println("Foo:static 1");
    }

    public Foo() {
        System.out.println("Foo:constructor");
    }
}
```

When all three initialization techniques (constructors, instance initializers, and static initializers) are present in a class, static initializers are always executed first (when the class is loaded into memory) in the order they are declared, followed by instance initializers in the order they are declared, and lastly by constructors. When a superclass is introduced, the order of execution changes slightly:

1. Static initializers of superclass, in order of their declaration
2. Static initializers of subclass, in order of their declaration
3. Instance initializers of superclass, in order of their declaration
4. Constructor of superclass
5. Instance initializers of subclass, in order of their declaration
6. Constructor of subclass

For example, we can create the following application:

```
public abstract class Bar {

    private String name;
```

```
4
5       static {
6           System.out.println("Bar:static 1");
7       }
8
9       {
10          System.out.println("Bar:instance 1");
11      }
12
13      static {
14          System.out.println("Bar:static 2");
15      }
16
17      public Bar() {
18          System.out.println("Bar:constructor");
19      }
20
21      {
22          System.out.println("Bar:instance 2");
23      }
24
25      public Bar(String name) {
26          this.name = name;
27          System.out.println("Bar:name-constructor");
28      }
29  }
30
31  public class Foo extends Bar {
32
33      static {
34          System.out.println("Foo:static 1");
35      }
36
37      {
38          System.out.println("Foo:instance 1");
39      }
40
41      static {
42          System.out.println("Foo:static 2");
43      }
44
45      public Foo() {
46          System.out.println("Foo:constructor");
47      }
```

```
48
49      public Foo(String name) {
50          super(name);
51          System.out.println("Foo:name-constructor");
52      }
53
54      {
55          System.out.println("Foo:instance 2");
56      }
57
58      public static void main(String... args) {
59          new Foo();
60          System.out.println();
61          new Foo("Baz");
62      }
63  }
```

If we execute this code, we receive the following output:

```
1   Bar:static 1
2   Bar:static 2
3   Foo:static 1
4   Foo:static 2
5   Bar:instance 1
6   Bar:instance 2
7   Bar:constructor
8   Foo:instance 1
9   Foo:instance 2
10  Foo:constructor
11
12  Bar:instance 1
13  Bar:instance 2
14  Bar:name-constructor
15  Foo:instance 1
16  Foo:instance 2
17  Foo:name-constructor
```

Note that the static initializers were only executed *once*, even though two `Foo` objects were created. While instance and static initializers can be useful, initialization logic should be placed in constructors and methods (or static methods) should be used when complex logic is required to initialize the state of an object.

# 3. Double-Brace Initialization

Many programming languages include some syntactic mechanism to quickly and concisely create a list or map (or dictionary) without using verbose boilerplate code. For example, C++ includes brace initialization which allows developers to quickly create a list of enumerated values, or even initialize entire objects if the constructor for the object supports this functionality. Unfortunately, prior to JDK 9, no such feature was included (we will touch on this inclusion shortly). In order to naively create a list of objects, we would do the following:

```
List<Integer> myInts = new ArrayList<>();
myInts.add(1);
myInts.add(2);
myInts.add(3);
```

While this accomplishes our goal of creating a new list initialized with three values, it is overly verbose, requiring the developer to repeat the name of the list variable for each addition. In order to shorten this code, we can use double-brace initialization to add the same three elements:

```
List<Integer> myInts = new ArrayList<>() {{
    add(1);
    add(2);
    add(3);
}};
```

Double-brace initialization–which earns its name from the set of two open and closed curly braces–is actually a composite of multiple syntactic elements. First, we create an anonymous inner class that extends the `ArrayList` class. Since `ArrayList` has no abstract methods, we can create an empty body for the anonymous implementation:

```
List<Integer> myInts = new ArrayList<>() {};
```

Using this code, we essentially create an anonymous subclass of `ArrayList` that is exactly the same as the original `ArrayList` . One of the major differences is that our inner class has an implicit reference to the containing class (in the form of a captured `this` variable) since we are creating a non-static inner class. This allows us to write some interesting–if not convoluted–logic, such as adding the captured `this` variable to the anonymous, double-brace initialized inner class:

```
public class Foo {

    public List<Foo> getListWithMeIncluded() {
        return new ArrayList<Foo>() {{
            add(Foo.this);
```

```
        }};
    }

    public static void main(String... args) {
        Foo foo = new Foo();
        List<Foo> fooList = foo.getListWithMeIncluded();
        System.out.println(foo.equals(fooList.get(0)));
    }
}
```

If this inner class were statically defined, we would not have access to `Foo.this`. For example, the following code, which statically creates the named `FooArrayList` inner class, does not have access to the `Foo.this` reference and is therefore *not compilable*:

```
public class Foo {

    public List<Foo> getListWithMeIncluded() {
        return new FooArrayList();
    }

    private static class FooArrayList extends ArrayList<Foo> {{
        add(Foo.this);
    }}
}
```

Resuming the construction of our double-brace initialized `ArrayList`, once we have created the non-static inner class, we then use instance initialization, as we saw above, to execute the addition of the three initial elements when the anonymous inner class is instantiated. Since anonymous inner classes are immediately instantiated and only one object of the anonymous inner class ever exist, we have essentially created a non-static inner singleton object that adds the three initial elements when it is created. This can be made more obvious if we separate the pair of braces, where one brace clearly constitutes the definition of the anonymous inner class and the other brace denotes the start of the instance initialization logic:

```
List<Integer> myInts = new ArrayList<>() {
    {
        add(1);
        add(2);
        add(3);
    }
};
```

While this trick can be useful, JDK 9 (JEP 269) has supplanted the utility of this trick with a set of static

While this trick can be useful, JDK 9 (JEP 269) has supplanted the utility of this trick with a set of static factory methods for `List` (as well as many of the other collection types). For example, we could have created the `List` above using these static factory methods, as illustrated in the following listing:

```
List<Integer> myInts = List.of(1, 2, 3);
```

This static factory technique is desirable for two main reasons: (1) No anonymous inner class is created and (2) the reduction in boilerplate code (noise) required to create the `List`. The caveat to creating a `List` in this manner is that the resulting `List` is immutable, and therefore cannot be modified once it has been created. In order to create a mutable `List` with the desired initial elements, we are stuck with either using the naive technique or double-brace initialization.

Note that the naive initialization, double-brace initialization, and the JDK 9 static factory methods are not just available for `List`. They are also available for `Set` and `Map` objects, as illustrated in the following snippet:

```
// Naive initialization
Map<String, Integer> myMap = new HashMap<>();
myMap.put("Foo", 10);
myMap.put("Bar", 15);

// Double-brace initialization
Map<String, Integer> myMap = new HashMap<>() {{
    put("Foo", 10);
    put("Bar", 15);
}};

// Static factory initialization
Map<String, Integer> myMap = Map.of("Foo", 10, "Bar", 15);
```

It is important to consider the nature of double-brace initialization before deciding to use it. While it does improve the readability of code, it carries with it some implicit side-effects.

# 4. Executable Comments

Comments are an essential part of almost every program and the main benefit of comments is that they are not executed. This is made even more evident when we comment out a line of code within our program: We want to retain the code in our application but we do not want it to be executed. For example, the following program results in `5` being printed to standard output:

```
public static void main(String args[]) {
    int value = 5;
    // value = 8;
    System.out.println(value);
```

```
}
```

While it is a fundamental assumption that comments are never executed, it is not completely true. For example, what does the following snippet print to standard output?

```
public static void main(String args[]) {
    int value = 5;
    // \u000dvalue = 8;
    System.out.println(value);
}
```

A good guess would be `5` again, but if we run the above code, we see `8` printed to standard output. The reason behind this seeming bug is the Unicode character `\u000d` ; this character is actually a Unicode carriage return, and Java source code is consumed by the compiler as Unicode formatted text files. Adding this carriage return pushes the assignment `value = 8` to the line directly following the comment, ensuring that it is executed. This means that the above snippet is effectively equal to the following:

```
public static void main(String args[]) {
    int value = 5;
    //
value = 8;
    System.out.println(value);
}
```

Although this appears to be a bug in Java, it is actually a conscious inclusion in the language. The original goal of Java was to create a platform independent language (hence the creation of the Java Virtual Machine, or JVM) and interoperability of the source code is a key aspect of this goal. By allowing Java source code to contain Unicode characters, we can include non-Latin characters in a universal manner. This ensures that code written in one region of the world (that may include non-Latin characters, such as in comments) can be executed in any other. For more information, see Section 3.3 of the Java Language Specification, or JLS.

We can take this to the extreme and even write an entire application in Unicode. For example, what does the following program do (source code obtained from Java: Executing code in comments?!)?

```
\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020\u0020
\u0063\u006c\u0061\u0073\u0073\u0020\u0055\u0067\u006c\u0079
\u007b\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0020\u0020
\u0020\u0020\u0020\u0020\u0073\u0074\u0061\u0074\u0069\u0063
\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u0069\u006e\u0028
\u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005d\u0020\u0020
\u0020\u0020\u0020\u0020\u0061\u0072\u0067\u0073\u0029\u007b
```

```
\u0053\u0079\u0073\u0074\u0065\u006d\u002e\u006f\u0075\u0074
\u002e\u0070\u0072\u0069\u006e\u0074\u006c\u006e\u0028\u0020
\u0022\u0048\u0065\u006c\u006c\u006f\u0020\u0077\u0022\u002b
\u0022\u006f\u0072\u006c\u0064\u0022\u0029\u003b\u007d\u007d
```

If the above is placed in a file named Ugly.java and executed, it prints `Hello world` to standard output. If we convert these escaped Unicode characters into American Standard Code for Information Interchange (ASCII) characters, we obtain the following program:

```
public
class Ugly
{public
    static
void main(
String[]
    args){
System.out
.println(
"Hello w"+
"orld");}}
```

Although it is important to know that Unicode characters can be included in Java source code, it is highly suggested that they are avoided unless required (for example, to include non-Latin characters in comments). If they are required, be sure not to include characters, such as carriage return, that change the expected behavior of the source code.

# 5. Enum Interface Implementation

One of the limitations of enumerations (enums) compared to classes in Java is that enums cannot extend another class or enum. For example, it is *not possible* to execute the following:

```
public class Speaker {

    public void speak() {
        System.out.println("Hi");
    }
}

public enum Person extends Speaker {

    JOE("Joseph"),
    JIM("James");
```

```
12
13      private final String name;
14
15      private Person(String name) {
16          this.name = name;
17      }
18  }
19
20  Person.JOE.speak();
```

We can, however, have our enum implement an interface and provide an implementation for its abstract methods as follows:

```
1   public interface Speaker {
2       public void speak();
3   }
4
5   public enum Person implements Speaker {
6
7       JOE("Joseph"),
8       JIM("James");
9
10      private final String name;
11
12      private Person(String name) {
13          this.name = name;
14      }
15
16      @Override
17      public void speak() {
18          System.out.println("Hi");
19      }
20  }
21
22  Person.JOE.speak();
```

We can now also use an instance of `Person` anywhere a `Speaker` object is required. Whatsmore, we can also provide an implementation of the abstract methods of an interface on a per-constant basis (called constant-specific methods):

```
1   public interface Speaker {
2       public void speak();
3   }
```

```
public enum Person implements Speaker {

    JOE("Joseph") {
        public void speak() { System.out.println("Hi, my name is Joseph"); }
    },
    JIM("James"){
        public void speak() { System.out.println("Hey, what's up?"); }
    };

    private final String name;

    private Person(String name) {
        this.name = name;
    }

    @Override
    public void speak() {
        System.out.println("Hi");
    }
}

Person.JOE.speak();
```

Unlike some of the other secrets in this article, this technique should be encouraged where appropriate. For example, if an enum constant, such as `JOE` or `JIM`, can be used in place of an interface type, such as `Speaker`, the enum that defines the constant should implement the interface type. For more information, see Item 38 (pp. 176-9) of Effective Java, 3rd Edition.

# Conclusion

In this article, we looked at five hidden secrets in Java, namely: (1) Annotations can be extended, (2) instance initialization can be used to configure an object upon instantiation, (3) double-brace initialization can be used to execute instructions when creating an anonymous inner class, (4) comments can sometimes be executed, and (5) enums can implement interfaces. While some of these features have their appropriate uses, some of them should be avoided (i.e. creating executable comments). When deciding to use these secrets, be sure to obey the following rule: Just because something can be done, does not mean that it should.

---

## Like This Article? Read More From DZone

**Java Holiday Calendar 2016 (Day 1): Use the @FunctionalInterface Annotation**

**Extensible Enum With Interface**

**The Highly Useful Java ChronoUnit Enum**


**Free DZone Refcard**
**Getting Started With Kotlin**

Topics: JAVA , INITIALIZATION , EXECUTABLE COMMENTS , ENUM INTERFACE , ANNOTATIONS , TUTORIAL

Opinions expressed by DZone contributors are their own.

## Java Partner Resources



# Akka: Data Access Actors

**by Alexey Zvolinskiy MVB · Feb 24, 18 · Java Zone**



An actor model gives us an outstanding solution for the building of high scale and high load systems. Those of you who work with Akka know that in the actor model, everything should be represented as an actor. In some sense, this axiom simplifies software development. Is this circumstance as good as it seems? In this article, I'm going to demonstrate some approaches that may be applied to actors that need to interact with a database. Just imagine, what is the sense of a high scale and high load system if it does not communicate with a database?

# Input

Let's assume that we want to implement an actor for accessing a person table:

```
//Postgres syntax

CREATE TABLE person (
  id serial primary key,
  full_name text not null,
  phone text,
  created_at timestamp not null
);
```

What operations do we want to have for the person table? Probably a standard CRUD set of operations: create, select by id, update full_name and phone, and delete by id.

When we know what we want, we can start doing some steps to implement the solution. Our goal is to answer the question: **What is the right way to design a data access actor**?

# Approach #1

Everything starts from a case class:

```
case class Person(id: Int, fullName: String, phone: Option[String], createdAt: LocalD
```

Let's assume that we have some database provider trait. In our case, it may be something like `PostgresDB` . It represents a driver to the database. That means that we can create some abstraction for data access:

```
trait PersonRepository {

  implicit val ec: ExecutionContext
  val db: PostgresDB

  def createPerson(fullName: String, phone: Option[String]): Future[Int]
  def getPerson(id: Int): Future[Option[Person]]
  def updatePerson(fullName: String, phone: Option[String]): Future[Boolean]
  def deletePerson(id: Int): Future[Boolean]
}
```

Of course, now we have to create some realization of this trait:

```
class PersonRepositoryImpl(db: PostgresDB)(implicit val ec: ExecutionContext) extends
```

```
class PersonRepositoryImpl(db: PostgreSQL)(implicit val ec: ExecutionContext) extends

    //implementation of the methods

}
```

Wait! This article is about actors, isn't it? So why don't we see any code related to actors? Let's correct this somehow. The most rational idea is to define messages for the actor first:

```
object PersonDataActor {

  case class CreatePerson(fullName: String, phone: Option[String])
  case class GetPerson(id: Int)
  case class UpdatePerson(fullName: String, phone: Option[String])
  case class DeletePerson(id: Int)
  //Then we can add response messages here as a reaction on the messages declared abo

}
```

With this set of messages, we can create `PersonDataActor`:

```
class PersonDataActor(personRepo: PersonRepository) extends Actor {

  implicit val system = context.system
  implicit val ec: ExecutionContext = system.dispatcher

  override def receive: Receive = {
    case cp: CreatePerson => //corresponding function call from personRepo
    case gp: GetPerson =>    //corresponding function call from personRepo
    case up: UpdatePerson => //corresponding function call from personRepo
    case dp: DeletePerson => //corresponding function call from personRepo
  }

}
```

Well. That's it.

Is this approach good to be used in a production? Well, at least it works. The more significant advantage is that we can mock `personRepo` for testing purposes. Hence the `PersonDataActor` is testable.

Unfortunately, when you need more than 1 repository for some reason, the actor's constructor becomes "fat".

```
class PersonDataActor(personRepo: PersonRepository,
                      mobProviderRepo: MobileProviderRepository,
                      phoneBlackListRepo: PhoneBlackListRepository) extends Actor {
  //...

}
```

That's how the things going in the approach #1.

# Approach #2

I hope that you have read the previous section, because I'm going to refer to it. So why don't we pass just one parameter to the actor's constructor? I mean `PostgresDB`. If we do so, this makes the actor construction more elegant, because all the repositories can be initialized inside of the actor:

```
class PersonDataActor(postgresDB: PostgresDB) extends Actor {
  val personRepo = new PersonRepositoryImpl(postgresDB)
  val mobProviderRepo = new MobileProviderRepositoryImpl(postgresDB)
  val phoneBlackListRepo = new PhoneBlackListRepositoryImpl(postgresDB)

  //...
}
```

Is this approach better than the first one? Actually no, because "elegance" of `PersonDataActor` constructor gives you less than it takes back. This code is hard to test: You are not able to mock the repositories as you need to according to test scenarios. So you will need to create an in-memory DB for each test suite run.

# Summary

I tried to highlight the problems that may occur with data access actors when you design your actor system. This article is definitely just the tip of the iceberg, though. Maybe I missed something when I tried to enforce separation of concerns in the context of actors. Anyway, I'll be really glad to read about your experience in this area.

How would you implement this data access actor?

---