Classes and objects in Java

Learn how to make classes, fields, methods, constructors, and objects work together in your Java applications

Classes, fields, methods, constructors, and objects are the building blocks of object-based Java applications. This tutorial teaches you how to declare classes, describe attributes via fields, describe behaviors via methods, initialize objects via constructors, and instantiate objects from classes and access their members. Along the way, you'll also learn about setters and getters, method overloading, setting access levels for fields, constructors, and methods, and more. *Note that code examples in this tutorial compile and run under Java 12.*

Advanced techniques: Fields and methods in Java

TABLE OF CONTENTS

Class declaration

A *class* is a template for manufacturing objects. You declare a class by specifying the class keyword followed by a non-reserved identifier that names it. A pair of matching open and close brace characters ({ and }) follow and delimit the class's body. This syntax appears below:

```
class identifier
{
  // class body
}
```

By convention, the first letter of a class's name is uppercased and subsequent characters are lowercased (for example, Employee). If a name consists of multiple words, the first letter of each word is uppercased (such as SavingsAccount). This naming convention is called CamelCasing.

The following example declares a class named Book:

```
class Book
{
  // class body
}
```

A class's body is populated with fields, methods, and constructors. Combining these language features into classes is known as *encapsulation*. This capability lets us program at a higher level of abstraction (classes and objects) rather than focusing separately on data structures and functionality.

What is an object-based application?

An *object-based application* is an application whose design is based on declaring classes, creating objects from them, and designing interactions between objects.

Utility classes

A class can be designed to have nothing to do with object manufacturing. Instead, it exists as a placeholder for class fields and/or class methods. Such a class is known as a *utility class*. An example of a utility class is the Java standard class library's Math class.

Multi-class applications and main()

A Java application is implemented by one or more classes. Small applications can be accommodated by a single class, but larger applications often require multiple classes. In that case one of the classes is designated as the *main class* and contains the main() entry-point method. For example, Listing 1 presents an application built using three classes: A, B, and C; C is the main class.

Listing 1. A Java application with multiple classes

```
class A
{
}

class B
{
}

class C
{
  public static void main(String[] args)
  {
    System.out.println("Application C entry point");
  }
}
```

You could declare these three classes in a single source file, such as D.java. You would then compile this source file as follows:

```
javac D.java
```

The compiler generates three class files: A.class, B.class, and C.class. Run this application via the following command:

```
java C
```

You should observe the following output:

```
Application C entry point
```

Alternatively, you could declare each class in its own source file. By convention, the source file's name matches the class name. You would declare A in A.java, for instance. You could then compile these source files separately:

```
javac A.java
javac B.java
javac C.java
```

To save time, you could compile all three source files at once by replacing the file name with an asterisk (but keep the .java file extension):

```
javac *.java
```

Either way, you would run the application via the following command:

```
java C
```

Public classes

Java lets you declare a class with public access via the public keyword. When you declare a class public, you must store it in a file with the same name. For example, you would store public class C {} in C.java. You may declare only one public class in a source file.

When designing multi-class applications, you will designate one of these classes as the main class and locate the main() method in it. However, there is nothing to prevent you from declaring main() methods in the other classes, perhaps for testing purposes. This technique is shown in Listing 2.

Listing 2. Declaring more than one main() method

```
class A
{
  public static void main(String[] args)
  {
    System.out.println("Testing class A");
  }
}
```

```
class B
{
  public static void main(String[] args)
  {
    System.out.println("Testing class B");
  }
}


class C
{
  public static void main(String[] args)
  {
    System.out.println("Application C entry point");
  }
}
```

After compiling the source code, you would execute the following commands to test the helper classes A and B, and to run the application class C:

```
java A
java B
java C
```

You would then observe the following lines of output, one line per java command:

```
Testing class A
Testing class B
Application C entry point
```

Be careful with main()

Placing a main() method in each class can be confusing, especially if you forget to document the main class. Also, you might forget to remove these methods before putting the application into production, in which case their presence would add bulk to the application. Furthermore, someone might run one of the supporting classes, which could disrupt the application's environment.

Fields: Describing attributes

A class models a real-world entity in terms of state (attributes). For example, a vehicle has a color and a checking account has a balance. A class can also include non-entity state. Regardless, state is stored in variables that are known as *fields*. A field declaration has the following syntax:

```
[static] type identifier [ = expression ] ;
```

A field declaration optionally begins with keyword static (for a non-entity attribute) and continues with a *type* that's followed by a non-reserved *identifier* that names the field. The field can be explicitly initialized by specifying = followed by an *expression* with a compatible type. A semicolon terminates the declaration.

The following example declares a pair of fields in Book:

```
class Book
{
  String title;
  int pubYear; // publication year
}
```

The title and pubYear field declarations are identical to the variable declarations. These fields are known as *instance fields* because each object contains its own copy of them.

The title and pubYear fields store values for a specific book. However, you might want to store state that is independent of any particular book. For example, you might want to record the total number of Book objects created. Here's how you would do it:

```
class Book
{
  // ...


  static int count;
}
```

This example declares a count integer field that stores the number of Book objects created. The declaration begins with the static keyword to indicate that there is only one copy of this field in memory. Each Book object can access this copy, and no object has its own copy. For this reason, count is known as a *class field*.

Initialization

The previous fields were not assigned values. When you don't explicitly initialize a field, it's implicitly initialized with all of its bits set to zero. You interpret this default value as false (for boolean), '\u0000' (for char), 0 (for int), 0L (for long), 0.0F (for float), 0.0 (for double), or null (for a reference type).

However, it is also possible to explicitly initialize a field when the field is declared. For example, you could specify static int count = 0; (which isn't necessary because count defaults to 0), String logfile = "log.txt";, static int ID = 1;, or even double sinPIDiv2 = Math.sin(Math.PI / 2);.

Although you can initialize an instance field through direct assignment, it's more common to perform this initialization in a constructor, which I'll demonstrate later. In contrast, a class field (especially a class constant) is typically initialized through direct assignment of an expression to the field.

Lifetime and scope

An instance field is born when its object is created and dies when the object is garbage collected. A class field is born when the class is loaded and dies when the class is unloaded or when the application ends. This property is known as *lifetime*.

Instance and class fields are accessible from their declarations to the end of their declaring classes. Furthermore, they are accessible to external code in an object context only (for instance fields) or object and class contexts (for class fields) when given suitable access levels. This property is known as *scope*.

Methods: Describing behaviors

In addition to modeling the state of a real-world entity, a class also models its behaviors. For example, a vehicle supports movement and a checking account supports deposits and withdrawals. A class can also include non-entity behaviors. Regardless, Java programmers use *methods* to describe behaviors. A method declaration has the following syntax:

```
[static] returnType identifier ( [parameterList] )
{
  // method body
}
```

A method declaration optionally begins with keyword static (for a non-entity behavior) and continues with a *returnType* that's followed by a non-reserved *identifier* that names the method. The name is then followed by a round bracket-delimited optional *parameterList*. A brace-delimited body containing code to execute when the method is called follows.

The *return type* identifies the type of values that are returned from the method via the return statement, which I'll discuss later. For example, if a method returns strings, its return type would be set to String. When a method doesn't return a value, its return type is set to void.

The *parameter list* is a comma-separated list of parameter declarations: each declaration consists of a type followed by a non-reserved identifier that names the parameter. A *parameter* is a variable that receives an *argument* (an expression value whose type is compatible with its corresponding parameter) when a method or constructor is called.

A parameter is local to its method or constructor. It comes into existence when the method or constructor is called and disappears when the method or constructor returns to its caller. In other words, its lifetime is the method execution. A parameter can be accessed by any code within the method. Its scope is the entire method.

The following example declares four methods in the Book class:

```
class Book
{
```

```
  // ...

  String getTitle()
  {
    return title;
  }

  int getPubYear()
  {
    return pubYear;
  }

  void setTitle(String _title)
  {
    title = _title;
  }

  void setPubYear(int _pubYear)
  {
    pubYear = _pubYear;
  }
}
```

The getTitle() and getPubYear() methods return the values of their respective fields. They use the return statement to return these values to the caller. Note that the type of this statement's expression must be compatible with the method's return type.

The setTitle() and setPubYear() methods let you set the values of the title and pubYear fields. Their return types are set to keyword void to indicate that they don't return any values to their callers. All four methods are known as *instance methods* because they affect only the objects on which they are called.

Setters and getters

The "set" prefix identifies setTitle() and setPubYear() as *setter methods*, meaning that they set values. Similarly, the "get" prefix identifies getTitle() and getPubYear() as *getter methods*, which means that they get values.

If you're wondering about the need for setter/getter methods in lieu of directly accessing title and pubYear, you'll find some good reasons in the "What is the point of getters and setters?" topic at StackOverflow.com.

The getTitle(), getPubYear(), setTitle(), and setPubYear() methods affect a single object's copies of the title and pubYear fields. However, you might want to declare a method that's independent of any particular book. For example, you might want to introduce a method that outputs the number of Book objects, as follows:

```
class Book
```

```
{
  // ...


  static void showCount()
  {
    System.out.println("count = " + count);
  }
}
```

This example declares a showCount() method that will output the value of the count field. The declaration begins with the static keyword to indicate that this method belongs to the class and cannot access individual object state; no objects need to be created. For this reason, showCount() is known as a *class method*.

Local variables

Within a method or constructor, you can declare additional variables as part of its implementation. These variables are known as *local variables* because they are local to the method/constructor. They exist only while the method or constructor is executing and cannot be accessed from outside the method/constructor. Consider the following example:

```
static void average(double[] values)
{
  double sum = 0.0;
  for (int i = 0; i < values.length; i++)
    sum += values[i];
  System.out.println("Average: " + (sum / values.length));
}
```

This example presents a method named average for calculating and outputting the average of an array of doubles. In addition to parameter values, it declares local variables sum and i to help in the calculation. sum's lifetime and scope range from its point of declaration to the end of the method. i's lifetime and scope are confined to the for loop.

A local variable's lifetime and scope are limited to the block in which it's been declared, as well as to sub-blocks. This is why i is inaccessible outside of the for loop, whose header is followed by a single-statement block. The following example should clarify this fact:

```
{
  int i;
  {
    i = 1; // okay: i still exists
  }
}
i = 2; // error: i no longer exists
```

Method overloading

Java lets you declare methods with the same name but with different parameter lists in the same class. This feature is known as *method overloading*. When the compiler encounters a method-call expression, it compares the called method's comma-separated list of arguments with each overloaded method's parameter list as it looks for the correct method to call.

Two same-named methods are overloaded when their parameter lists differ in number or order of parameters. Alternatively, two same-named methods are overloaded when at least one parameter differs in type. For example, consider the following four draw() methods, which draw a shape or string at the current or specified draw position:

```
void draw(Shape shape)
{
  // drawing code
}

void draw(Shape shape, double x, double y)
{
  // drawing code
}

void draw(String string)
{
  // drawing code
}

void draw(String string, double x, double y)
{
  // drawing code
}
```

When the compiler encounters draw("abc");, it will select the third method because it offers a matching parameter list. However, what will the compiler do when it encounters draw(null, 10, 20);? It will report a "reference to draw is ambiguous" error message because there are two methods from which to choose.

You cannot overload a method by changing only the return type. For example, you couldn't specify int add(int x, int y) and double add(int x, int y) because the compiler doesn't have enough information to distinguish between these methods when it encounters add(4, 5); in source code. The compiler would report a "redefinition" error.

Return statement

Sometimes, a method must terminate its execution before finishing. For example, a method for plotting a pixel might detect negative coordinates. Other methods need to return a value to their callers. For both situations, Java provides the return statement to terminate method execution and return control to the method's caller. This statement has the following syntax:

```
return [ expression ] ;
```

You can specify return without an expression to prematurely exit a method or a constructor. For example, consider a copy() method that copies bytes from the standard input stream (obtained via System.in.read() method calls) to the standard output stream (via System.out.print() method calls). This method is shown below:

```
static void copy() throws java.io.IOException // I'll discuss throws and exceptions
{                                             // in a future article.
  while (true)
  {
    int _byte = System.in.read();
    if (_byte == -1)
      return;
    System.out.print((char) _byte);
  }
}
```

System.in.read() reads bytes from the standard input stream, which defaults to the keyboard but can be redirected to a file. When the stream is redirected to a file, -1 is returned when there are no more bytes to read. When copy() detects this situation, it executes return to return from the infinite loop to its caller.

You can specify return with an expression to return a value to the method's caller. (Constructors don't support this version of return because they don't have return types.) For example, you might return early from a method that is searching for a specific array value when the value is found. This scenario is demonstrated in the following example:

```
static int search(int[] values, int srchValue)
{
  for (int i = 0; i < values.length; i++)
    if (values[i] == srchValue)
      return i; // return index of found value
  return -1; // -1 is an invalid index, so it's useful for indicating "value not found"
}
```

Constructors: Initializing objects

As well as explicitly assigning values to fields, a class can declare one or more blocks of code for more extensive object initialization. Each code block is a *constructor*. Its declaration

consists of a header followed by a brace-delimited body. The header consists of a class name (a constructor doesn't have its own name) followed by an optional parameter list:

```
className ( [parameterList] )
{
  // constructor body
}
```

The *className* must match the name of the class in which the constructor is declared. The *parameterList* is a comma-separated list of parameters. A brace-delimited body containing code to execute when the constructor is called follows. Unlike a method, a constructor doesn't have a return type because it doesn't return any value.

Default no-argument constructor

When a class doesn't declare any constructors, the compiler generates a default no-argument constructor. This default constructor does nothing.
The following example declares a constructor in the Book class. The constructor initializes a Book object's title and pubYear fields to the arguments that were passed to the constructor's _title and _pubYear parameters when the object was created. The constructor also increments the count class field:

```
class Book
{
  // ...

  Book(String _title, int _pubYear)
  {
    title = _title;
    pubYear = _pubYear;
    ++count;
  }

  // ...
}
```

The parameter names have leading underscores to prevent a problem with the assignments. For example, if you renamed _title to title and specified title = title;, you would have merely assigned the parameter's value to the parameter, which accomplishes nothing. However, you can avoid this problem by prefixing the field names with this.:

```
class Book
{
  // ...
```

```
  Book(String title, int pubYear)
  {
    this.title = title;
    this.pubYear = pubYear;
    ++count;
  }

  // ...

  void setTitle(String title)
  {
    this.title = title;
  }

  void setPubYear(int pubYear)
  {
    this.pubYear = pubYear;
  }

  // ...
}
```

A parameter (or local variable) name that's identical to an instance field name *shadows* (meaning hides or masks) the field. Keyword this represents the current object (actually, its reference). Prepending this. to the field name removes the shadowing by accessing the field name instead of the same-named parameter.

Shadowing fields

A parameter or local variable name can shadow a class field name. You can prepend this. in instance contexts to the class field name to access this field, but it's clearer to prepend the class name and member access operator.

Although you can initialize fields such as title and pubYear through the assignments shown above, it's preferable to perform the assignments via setter methods such as setTitle() and setPubYear(), as demonstrated below:

```
class Book
{
  // ...

  Book(String title, int pubYear)
  {
    setTitle(title);
    setPubYear(pubYear);
    ++count;
```

```
  }


  //...
}
```

Note that in the future these methods might perform additional initialization tasks; why duplicate this code in the constructor?

Constructor calling

Classes can declare multiple constructors. For example, consider a Book constructor that accepts a title argument only and sets the publication year to -1 to indicate that the year of publication is unknown. This extra constructor along with the original constructor are shown below:

```
class Book
{
  //...

  Book(String title)
  {
    setTitle(title);
    setPubYear(-1);
    ++count;
  }

  Book(String title, int pubYear)
  {
    setTitle(title);
    setPubYear(pubYear);
    ++count;
  }

  //...
}
```

But there is a problem with this new constructor: it duplicates code (setTitle(title);)) located in the existing constructor. Duplicate code adds unnecessary bulk to the class. Java provides a way to avoid this duplication by offering this() syntax for having one constructor call another:

```
class Book
{
  //...
```

```
  Book(String title)
  {
    this(title, -1);

    // Do not include ++count; here because it already
    // executes in the second constructor and would
    // execute here after this() returns. You would end
    // up with one extra book in the count.
  }

  Book(String title, int pubYear)
  {
    setTitle(title);
    setPubYear(pubYear);
    ++count;
  }

  // ...
}
```

The first constructor uses keyword this followed by a bracketed argument list to call the second constructor. The single parameter value is passed unchanged as the first argument, and -1 is passed as the second argument. When using this(), remember that it must be the first piece of code in a constructor; otherwise, the compiler reports an error.

Objects: Working with class instances

Once you have declared a class, you can create objects from it. An *object* is nothing more than a class instance. For example, now that the Book class has been declared, you can create one or more Book objects. Accomplish this task by specifying the new operator followed by a Book constructor, as follows:

```
Book book = new Book("A Tale of Two Cities", 1859);
```

new loads Book into memory and then calls its constructor with arguments "A Tale of Two Cities" and 1859. The object is initialized to these values. When the constructor returns from its execution, new returns a *reference* (some kind of pointer to an object) to the newly initialized Book object. This reference is then assigned to the book variable.

Constructor return type

If you were wondering why a constructor doesn't have a return type, the answer is that there is no way to return a constructor value. After all, the new operator is already returning a reference to the newly-created object.

Accessing fields

After creating a Book object, you can access its instance fields by using the member access operator (.) with the Book reference:

```
System.out.println(book.title); // Output: A Tale of Two Cities

book.pubYear = 2019;
System.out.println(book.pubYear); // Output: 2019
```

You don't have to create any Book objects to access class fields. Instead, you prepend the class name and member access operator to the class method's name when accessing these fields:

```
System.out.println(Book.count); // Output: 1

Book.count = 0;
System.out.println(Book.count); // Output: 0
```

Calling methods

After creating a Book object, you can call its getTitle() and getPubYear() methods to return the instance field values. Also, you can call setTitle() and setPubYear() to set new values. In either case, you use the member access operator with the Book reference to accomplish this task:

```
System.out.println(book.getTitle()); // Output: A Tale of Two Cities

System.out.println(book.getPubYear()); // Output: 1859

book.setTitle("Moby Dick");

book.setPubYear(1851);

System.out.println(book.getTitle()); // Output: Moby Dick
System.out.println(book.getPubYear()); // Output: 1851
```

Messaging objects

Calling a method on an object is equivalent to sending a *message* to the object. The name of the method and its arguments are conceptualized as a message that is being sent to the object on which the method is called.

You don't have to create any Book objects to call class methods. Instead, you prepend the class name and member access operator to the class method's name when calling these methods:

```
Book.showCount(); // Output: count = 1
```

I previously mentioned that instance methods affect only the objects on which they are called; they don't affect other objects. The following example reinforces this truth by creating two Book objects and then accessing each object's title, which is subsequently output:

```
Book book1 = new Book("A Tale of Two Cities", 1859);
```

```
Book book2 = new Book("Moby Dick", 1851);
Book book3 = new Book("Unknown");
System.out.println(book1.getTitle()); // Output: A Tale of Two Cities
System.out.println(book2.getTitle()); // Output: Moby Dick
System.out.println(book3.getPubYear()); // Output: -1
Book.showCount(); // Output: count = 3
```

Calling varargs methods

When calling a method that takes one or more array arguments, you either pass the name of an array or specify extra syntax that tends to clutter source code. For example, I previously presented the void average(double[] values) class method that calculates the average value from an array of double precision floating-point values, and then outputs the average. The following code fragment shows the two ways you could call this method:

```
double[] values = { 1.0, 2.0, 3.0, 4.0 };
average(values);
average(new double[] { 1.0, 2.0, 3.0, 4.0 });
```

Java 5 introduced a *variable arguments (varargs)* feature to reduce the clutter when passing an array to a method or constructor. To use varargs, declare the method or constructor with ... after the rightmost parameter type name in the method's/constructor's parameter list:

```
void average(double... values) { /* ... */ }
```

The compiler treats ... as syntactic sugar for declaring an array. In the example, we have simply replaced [] with .... However, this syntactic sugar allows us to specify 1.0, 2.0, 3.0, 4.0 without the surrounding new double[] { } clutter when calling this method:

```
average(1.0, 2.0, 3.0, 4.0);
```

In spite of the ... syntax, the values parameter is still an array of type double[]. You don't have to modify the code within average()'s body to interact with values.

Information hiding and access levels

A class's body is composed of interface and implementation. The *interface* is that part of the class that's accessible to code located outside of the class. The *implementation* is that part of the class that exists to support the interface. Implementation should be hidden from external code so that it can be changed to meet evolving requirements.

Consider the Book class. Constructor and method headers form this class's interface. The code within the constructors and methods, and the various fields are part of the implementation. There is no need to access these fields because they can be read or written via the getter and setter methods.

However, because no precautions have been taken, it's possible to directly access these fields. Using the previous book reference variable, you could

specify book.title and book.pubYear, and that would be okay with the Java compiler. To prevent access to these fields (or at least determine who can access them), you need to take advantage of access levels.

An *access level* is an indicator of who can access a field, method, or constructor. Java supports four access levels: private, public, protected, and package (the default). Java provides three keywords that correspond to the first three access levels:

1. private: Only code in the same class as the member can access the member.
2. public: Any code in any class in any package can access the member.
3. protected: Any code in the same class or its subclasses can access the member.

If there is no keyword then package access is implied. Package access is similar to public access in that code outside of the class can access the member. However, unlike public access, the code must be located in a class that belongs to the same package (discussed later) as the class containing the member that is to be accessed.

You can prevent external code from accessing Book's title and pubYear fields so that any attempt to access these fields from beyond Book will result in a compiler error message. Accomplish this task by prepending private to their declarations, as demonstrated below:

```
class Book
{
  // fields

  private String title;
  private int pubYear; // publication year

  // ...
}
```

To show you why it's a good idea to hide access to fields, I've created another example that expands the Book class to include an author field, getter/setter methods for accessing this field, and another constructor that lets you also specify an author's name. The following example shows the modified parts of the updated Book class:

```
class Book
{
  // ...

  private String author;

  // ...

  Book(String title, int pubYear)
  {
```

```
    this(title, pubYear, "");
  }

  Book(String title, int pubYear, String author)
  {
    setTitle(title);
    setPubYear(pubYear);
    setAuthor(author);
    ++count;
  }

  // ...

  String getAuthor()
  {
    return author;
  }

  // ...

  void setAuthor(String author)
  {
    this.author = author;
  }

  // ...
}
```

If you recall, the `Book(String title)` constructor executes `this(title, -1);` to avoid code duplication. Similarly, I've designed `Book(String title, int pubYear)` to execute `this(title, pubYear, "");`, which calls the newest constructor, to avoid code duplication.

You can easily create a new `Book` object that includes the author name and obtain this name via the `getAuthor()` method:

```
Book book = new Book("A Tale of Two Cities", 1859, "Charles Dickens");
System.out.println(book.getAuthor()); // Output: Charles Dickens
```

Consider that the upgraded `Book` class doesn't allow for multiple authors. You cannot change the constructor or `getAuthor()`/`setAuthor()` headers to make this enhancement because doing so would change the interface and cause external code that relies on this interface to stop working. Instead, you would need to rework the implementation, as follows:

```
class Book
```

```
{
  // ...

  private String[] authors;

  // ...

  Book(String title, int pubYear, String author)
  {
    this(title, pubYear, new String[] { author });
  }

  Book(String title, int pubYear, String[] authors)
  {
    setTitle(title);
    setPubYear(pubYear);
    setAuthors(authors);
    ++count;
  }

  // ...

  String getAuthor()
  {
    return authors[0];
  }

  String[] getAuthors()
  {
    return authors;
  }

  // ...

  void setAuthor(String author)
  {
    setAuthors(new String[] { author });
  }

  void setAuthors(String[] authors)
  {
```

```
    this.authors = authors;
  }


  // ...
}
```

I changed the previous `author` field to an `authors` field. Also, I changed its type from `String` to the `String[]` array type.

I modified the `Book(String title, int pubYear, String author)` constructor to call the `Book(String title, int pubYear, String[] authors)` constructor. Because that constructor's third parameter is of type `String[]`, the `this()` call converts its `String author` argument to a single-element array consisting of itself via `new String[] { author }`. The `new` operator is used to create a single-element `String` array and assign the `author` string reference to this element.

The original `getAuthor()` method accesses the first element in the `authors` array. I modified the original `setAuthor()` method to convert its single `String` argument to a single-element `String` array, then call `setAuthors()` with this array as the argument.

Finally, I added new `getAuthors()` and `setAuthors()` methods to return the `authors` array reference or assign a new reference to `authors`.

You can create a new `Book` object that includes multiple author names by calling the new constructor. You can then obtain only the first name via `getAuthor()` or obtain all names via `getAuthors()`:

```
Book book = new Book("Grimms' Fairy Tales", 1812,
                     new String[] { "Jacob Grimm", "Wilhelm Grimm" });
System.out.println(book.getTitle()); // Output: Grimms' Fairy Tales
System.out.println(book.getPubYear()); // Output: 1812
System.out.println(book.getAuthor()); // Output: Jacob Grimm
String[] authors = book.getAuthors();
for (int i = 0; i < authors.length; i++)
  System.out.println(authors[i]); // Output: Jacob Grimm Wilhelm Grimm (on successive lines)
```

If the original `author` field wasn't marked `private`, you wouldn't be able to expand `Book` this way. Furthermore, external code would be able to directly access this field (e.g., `book.author`) and would break when you changed the field name.

Hiding constructors or methods

As well as hiding fields, you might need to hide constructors or methods. You typically hide a constructor to prevent a utility class from being instantiated. You typically hide a method when it only exists to service another method (or a constructor). Such a method is known as a *helper method*. Below is a short example:

```
// compute permutation of n as n! / (n - r)!.

public long perm(long n)
{
  return fact(n) / fact(n - r);
}

// fact() is hidden because it has no use outside
// of the class that computes the permutation.

private long fact(long n)
{
  // code that computes and returns n!
}
```