Stop writing so much code!

Learn the benefits of code reuse with these four Apache Commons Lang classes

Skill Level: Intermediate

Andrew Glover (ajglover@gmail.com)
Author and developer

16 Dec 2008

Write less code by leveraging a battle-hardened collection of open source utilities from the Apache Commons project's Lang library. Reusing other people's reliable code helps you get your software to market more quickly, with fewer defects.

Section 1. Before you start

About this tutorial

Commons Lang is a component of Apache Commons, a macro project whose many subprojects relate to various aspects of software development in the Java™ language. Commons Lang extends the standard `java.lang` API with string-manipulation methods, basic numerical methods, object reflection, creation and serialization, and `System` properties. It also contains an inheritable `enum` type, support for multiple types of nested `Exceptions`, enhancements to `java.util.Date`, and utilities that help with building methods such as `hashCode`, `toString`, and `equals`. I've found Commons Lang to be helpful across a wide variety of application verticals. By using Commons Lang, you'll end up writing less code, which lets you deliver production-ready software faster and with fewer defects. This tutorial guides you step-by-step through the fundamental concepts of using a few different Commons Lang classes and leveraging their code so that you don't have to write so much of it yourself.

Objectives

You'll learn how to:

- Implement object contracts such as `equals` and `hashCode`.
- Verify their proper functionality.
- Implement the `Comparable` interface's `compareTo` method.

When you are done with the tutorial, you will understand the benefits of the Commons Lang library and learn how to write less code.

Prerequisites

To get the most from this tutorial, you should be familiar with Java syntax and the basic concepts of object-oriented development on the Java platform. You should be familiar with refactoring and normal unit testing as well.

System requirements

To follow along and try out the code for this tutorial, you need:

- A working installation of either:
  - Sun's JDK 1.5.0_09 (or later).
  - IBM® Developer Kit for Java technology 1.5.0 SR3.
- The current release of the Commons Lang project (2.4 as of this writing). After you download and unpack the distribution, include commons-lang-2.4.jar in your classpath.

The recommended system configuration for this tutorial is:

- A system supporting either the Sun JDK 1.5.0_09 (or later) or the IBM JDK 1.5.0 SR3 with at least 500MB of main memory.
- At least 20MB of disk space to install the software components and examples covered.

The instructions and examples in the tutorial are based on a Microsoft® Windows® operating system. All the tools covered in the tutorial also work on Linux® and UNIX® systems.

Section 2. The benefits of code reuse

In the early days of software development, a developer's productivity was thought to be directly proportional to the quantity of code he or she wrote. At the time, it seemed like a plausible metric: code ultimately yields a presumably working binary asset, so someone who appears to be writing a lot of code must be working diligently toward producing a working application. The metric seems to apply to other industries. An accountant who handles a lot of tax returns, or a barista who makes a lot of espresso drinks, must be productive, right? Both apparently generate more revenue for their respective businesses because they produce a lot of the items they are supposed to.

But we learned some time ago that more lines of code don't correlate with productivity. Lots of code certainly indicates activity, but activity doesn't necessarily correlate with progress. Accountants who produce a lot of incorrect tax returns per day are highly active but are producing little value for their clients and employers. A barista who serves up coffee at lightning speeds but gets the orders wrong is definitely performing a lot of activity but not being productive.

More code can mean more defects

Thankfully, the software industry generally accepts that too much code can be a bad thing. Two studies have found that the average application contains anywhere from 20 to 250 bugs per 1,000 lines of code (see Resources)! This metric is known as *defect density*. You can draw a major conclusion from this data: *fewer lines of code means fewer defects*.

Of course, code still needs to be written. We aren't at the point yet where applications can write themselves, but we are at the point where we can borrow a lot of code. We haven't yet realized reuse in business components — the vision that one developer can reuse another's `Account` object, for example — but reuse in platform concerns is already here. A proliferation of open source frameworks and supporting code that is easy to reuse can help you write an `Account` object (for example) with as few lines of code as possible.

For instance, Hibernate and Spring are ubiquitous in the Java community. Taking the `Account` object example, teams embarking today on a greenfield development project to build an online ordering application (requiring an `Account` object) would gain tremendous advantages from leveraging Hibernate or a worthy competitive object-relational mapping (ORM) framework, as opposed to writing an ORM

framework from scratch. The same goes for other aspects of the application, such as unit testing (you'd use something like JUnit, right?) or dependency injection (Spring is an obvious candidate). That's reuse. It's just different from what we once thought it would look like.

By borrowing or reusing these frameworks, you end up having to write less code and can focus more appropriately on the business problem at hand. The frameworks themselves have a lot of code, but the point is you don't need to write or maintain it. That's the beauty of successful open source projects: other people are doing that for you. And they may well be better at it than you.

Less is better

Fewer lines of code can result in a quicker time to market with fewer defects. But reuse is important not only because it means writing less code, but also because it means leveraging what you could call the "wisdom of crowds." Popular open source frameworks and tools — such as Hibernate, Spring, JUnit, and the Apache Web server — are being used by a multitude of people across the globe in varying applications. This battle-hardened and tested software isn't defect-free, but you can safely assume that any issues that do arise will be found and fixed swiftly and at no cost.

The Apache Commons project has been around for years and is stable. The latest release contains roughly 90 classes and almost 1,800 unit tests. Although coverage information isn't published (and certainly one could argue that this project could have low code coverage), the numbers speak for themselves. That's essentially 20 tests per class. I'm willing to bet that the project's code is at least as well tested as yours.

Section 3. Object contracts

The Commons Lang library comes with a handy set of classes, collectively known as *builders*. In this section, you'll learn how to use one of them to build the `java.lang.Object equals` method and help reduce the amount of code you write.

Method-implementation challenges

All Java classes implicitly inherit from `java.lang.Object`. And as you probably already know, the `Object` class has three methods that are generally meant to be

overridden:

- `equals`
- `hashCode`
- `toString`

The `equals` and `hashCode` methods are special in that other aspects of the Java platform, such as collections and even persistence frameworks (including Hibernate), depend on these two methods to be properly implemented.

If you have never implemented `equals` and `hashCode`, you might assume that this is a simple task — but you'd be wrong. Joshua Bloch's book *Effective Java* (see Resources) devotes more than 10 pages to the particulars of implementing the `equals` method. And if you do end up implementing the `equals` method, you are required to implement the `hashCode` method also (because the `equals` contract states that two equal objects must have identical hash codes). Bloch spends an additional 6 pages explaining the `hashCode` method. That's at least 16 pages of detailed information on properly implementing 2 apparently simple methods.

The challenge of implementing the `equals` method is with the contract that this method must follow. `equals` must:

- Be reflexive:
  - For some object, `foo` (that isn't `null`), `foo.equals(foo)` must return `true`.
- Be symmetric:
  - For objects `foo` and `bar` (that aren't `null`), if `foo.equals(bar)` returns `true`, then `bar.equals(foo)` must also return `true`.
- Be transitive:
  - For objects `foo`, `bar`, and `baz` (that aren't `null`), if `foo.equals(bar)` is `true` and `bar.equals(baz)` equals `true`, then `foo.equals(baz)` must also return `true`.
- Be consistent:
  - For objects `foo` and `bar`, if `foo.equals(bar)` returns `true`, then the `equals` method should always return `true` regardless of how many times the `equals` method is invoked (provided that neither object actually changes).
- Handle `null`s properly:
  - `foo.equals(null)` should return `false`.

After reading this and even perhaps studying *Effective Java*, you might feel you are up for the challenge of properly implementing the `equals` method on your `Account` object. But remember the point I made earlier regarding productivity and activity.

You are supposed to be building an online Web application for your business, and the sooner this application is live, the sooner you and your business can make money. Armed with this simple fact, do you now spend a few hours (or days?) properly implementing and *testing* the `equals` contract on your objects — or does it make sense to reuse someone else's code?

Building equals

When it comes to implementing the `equals` method, the Commons Lang `EqualsBuilder` is useful. This class is simple to grasp. It essentially has two methods you need to know about: `append` and `isEquals`. The `append` method takes two properties: one of the underlying object and the same one of the object being compared. Because the `append` method returns an instance of the `EqualsBuilder`, you can chain successive calls and compare all desired properties of an object. And you can finish the chain by calling the `isEquals` method.

For instance, create an `Account` object as I've done in Listing 1:

Listing 1. A simple Account object

```
import org.apache.commons.lang.builder.CompareToBuilder;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

import java.util.Date;

public class Account implements Comparable {
 private long id;
 private String firstName;
 private String lastName;
 private String emailAddress;
 private Date creationDate;

 public Account(long id, String firstName, String lastName,
   String emailAddress, Date creationDate) {
  this.id = id;
  this.firstName = firstName;
  this.lastName = lastName;
  this.emailAddress = emailAddress;
  this.creationDate = creationDate;
 }

 public long getId() {
  return id;
 }

 public String getFirstName() {
  return firstName;
```

```
 }

 public String getLastName() {
  return lastName;
 }

 public String getEmailAddress() {
  return emailAddress;
 }

 public Date getCreationDate() {
  return creationDate;
 }
}
```

The `Account` object, for our purposes, is quite simple and isolated. At this point, you can run a quick test, as shown in Listing 2, to see if you can rely on the default implementation of `equals`:

Listing 2. Testing the Account object's default equals method

```
import org.junit.Test;
import org.junit.Assert;
import com.acme.app.Person;

import java.util.Date;

public class AccountTest {
 @Test
 public void verifyEquals(){
  Date now = new Date();
  Account acct1 = new Account(1, "Andrew", "Glover", "ajg@me.com", now);
  Account acct2 = new Account(1, "Andrew", "Glover", "ajg@me.com", now);

  Assert.assertTrue(acct1.equals(acct2));
 }
}
```

As you can see in Listing 2, I've created two identical `Account` objects, each with its own reference (thus == would return `false`). When I try to see if they are equal, JUnit kindly informs me that `false` is returned instead.

Remember, the `equals` method is leveraged by various aspects of the Java platform, including the Java language's collection classes. So it makes a lot of sense to implement a working version of this method. Accordingly, I'll override the `equals` method.

Remember, the `equals` contract can't work with `null` objects. Also, two different types of objects (`Accounts` and `Persons`, for example) can't be equal. Lastly, in Java code, the `equals` method is distinctly different than the == operator (which, if you remember, returns `true` if two objects share the *same* reference; consequently, those two objects must be equal). Two objects could be equal (and thus return `true` for `equals`) but not share the same reference.

Accordingly, the first aspect of an `equals` method can be coded as in Listing 3:

Listing 3. Quick conditionals in equals

```
if (this == obj) {
 return true;
}
if (obj == null || this.getClass() != obj.getClass()) {
 return false;
}
```

In Listing 3, I create two conditionals that should be verified before I try to compare respective properties of the underlying object and the `obj` parameter passed in.

Next, because the `equals` method takes an `Object` type, it makes sense to cast the `obj` parameter to `Account`, as shown in Listing 4:

Listing 4. Casting the obj parameter

```
Account account = (Account) obj;
```

Assuming the `equals` logic has made it this far, it's time to leverage the `EqualsBuilder` object. Remember, this object is designed to compare similar properties of the underlying object (`this`) with the type passed into the `equals` method, using the `append` method. Because these methods can be chained, you can finish the chain with the `isEquals` method, which returns `true` or `false`. Consequently, you can write essentially one line of code, as I've done in Listing 5:

Listing 5. Reusing EqualsBuilder

```
return new EqualsBuilder().append(this.id, account.id)
  .append(this.firstName, account.firstName)
   .append(this.lastName, account.lastName)
    .append(this.emailAddress, account.emailAddress)
     .append(this.creationDate, account.creationDate)
      .isEquals();
```

Putting it all together yields an `equals` method like the one in Listing 6:

Listing 6. equals fully coded

```
public boolean equals(Object obj) {
 if (this == obj) {
  return true;
 }
 if (obj == null || this.getClass() != obj.getClass()) {
  return false;
 }

 Account account = (Account) obj;

 return new EqualsBuilder().append(this.id, account.id)
  .append(this.firstName, account.firstName)
```

```
    .append(this.lastName, account.lastName)
     .append(this.emailAddress, account.emailAddress)
      .append(this.creationDate, account.creationDate)
       .isEquals();
}
```

Now rerun the previously failing test (see Listing 2). You should see a successful result.

You didn't spend any time trying to write your own version of `equals`. If you're still curious how you would write a proper `equals` method, suffice it to say that it involves a lot of conditionals. For instance, a slight snippet of a non-`EqualsBuilder` implemented `equals` method could compare the `creationDate` property, as shown in Listing 7:

Listing 7. A snippet of your own equals method

```
if (creationDate != null ? !creationDate.equals(
   person.creationDate) : person.creationDate != null){
 return false;
}
```

Note, using a ternary in this case makes the code a bit more precise, but arguably at the cost of comprehension. Nevertheless, the point is that I could either write a series of conditionals comparing the various aspects of each object's properties or I could leverage `EqualsBuilder` (which does the same exact thing). Which would you choose?

Note too that if you really want to streamline your `equals` method and write as little code as possible (which means less code to maintain), you can leverage the power of reflection and write the code in Listing 8:

Listing 8. Using EqualsBuilder's reflection API

```
public boolean equals(Object obj) {
 return EqualsBuilder.reflectionEquals(this, obj);
}
```

How's that for a reduction in code?

Listing 8 does have a downside. The `EqualsBuilder` must sneakily turn off access control in the underlying object (in order to compare `private` fields). This can fail if your VM is configured with security in mind. And, the heavy reflection Listing 8 uses can affect the `equals` method's run-time performance. On the plus side, though, if you add new properties and you are using the reflection API, you don't need to update your `equals` method (as you would if you stuck with the nonreflection way).

With the `EqualsBuilder`, you can leverage the power of reuse. It gives you two

choices for implementing the `equals` method. Which one you choose is up to you and depends on your particular situation. The one-liner style is simple and sweet, but as you now understand, not without risks.

Section 4. Hashing away at objects

Now that you've knocked out a proper `equals` method without having to write too much code, you can't forget to override `hashCode` too. This section shows you how.

Building hashCode

The `hashCode` method also has a contract but it isn't as formal as the `equals` contract. Nevertheless, it's important that you get it correct. As with `equals`, the result must be consistent. And if two objects, `foo` and `bar`, return `true` for `foo.equals(bar)`, then both `foo` and `bar`'s `hashCode` methods must return the same value. If `foo` and `bar` are not equal, they are not required to return different hash codes; however, the Javadocs suggest that if those objects do have different results, things will generally operate better.

Of note also is that, as you've probably noticed before, `hashCode` returns a seemingly random integer even if you fail to override it. This is because the underlying platform typically coverts the underlying object's address location into an integer; despite that, the documentation states this isn't a requirement and thus could change. Regardless, if you do end up overriding the `equals` method, it makes sense to also override the `hashCode` method. (Remember, Joshua Bloch's *Effective Java* spends six pages on properly implementing the `hashCode` method even though it appears to work out of the box.)

The Commons Lang library provides a `HashCodeBuilder` class that operates almost identically to the `EqualsBuilder`. But instead of comparing two properties, it appends a single property to generate an integer that obeys the contract I just described.

In your `Account` object, go ahead and override the `hashCode` method, as shown in Listing 9:

Listing 9. A default hashCode method

```
public int hashCode() {
 return 0;
}
```

Because there's nothing to compare when you generate a hash code, using the `HashCodeBuilder` is a one-liner. What's important is that you initialize the `HashCodeBuilder` correctly. The constructor takes two `ints`, which it uses to create a hash code. These two `ints` must be odd. The `append` method takes a property and, as before, these methods can be chained. The chain can be completed with a call to the `toHashCode` method.

Given those bits of information, you can then implement a `hashCode` method as I've done in Listing 10:

Listing 10. Implementing a hashCode method with HashCodeBuilder

```
public int hashCode() {
 return new HashCodeBuilder(11, 21).append(this.id)
  .append(this.firstName)
   .append(this.lastName)
    .append(this.emailAddress)
     .append(this.creationDate)
      .toHashCode();
}
```

Note that I pass in an 11 and a 21 in the constructor. These are completely randomly chosen odd numbers for this object. Open up the `AccountTest` from earlier (see Listing 2). Add a quick check to verify the contract that if `equals` returns `true` for two objects, then `hashCode` should return the same number. Listing 11 shows the modified test:

Listing 11. Verifying the hashCode contract for two equal objects

```
import org.junit.Test;
import org.junit.Assert;
import com.acme.app.Account;

import java.util.Date;

public class AccountTest {
 @Test
 public void verifyAccountEquals(){
  Date now = new Date();
  Account acct1 = new Account(1, "Andrew", "Glover", "ajg@me.com", now);
  Account acct2 = new Account(1, "Andrew", "Glover", "ajg@me.com", now);

  Assert.assertTrue(acct1.equals(acct2));
  Assert.assertEquals(acct1.hashCode(), acct2.hashCode());
 }
}
```

In Listing 11, I verify that two equal objects share the same hash code. Next, in Listing 12, I also check to verify that two *different* objects have different hash codes:

Listing 12. Verifying the hashCode contract for two different objects

```
@Test
public void verifyAccountDifferentHashCodes(){
 Date now = new Date();
 Account acct1 = new Account(1, "John", "Smith", "john@smith.com", now);
 Account acct2 = new Account(2, "Andrew", "Glover", "ajg@me.com", now);

 Assert.assertFalse(acct1.equals(acct2));
 Assert.assertTrue(acct1.hashCode() != acct2.hashCode());
}
```

If, just out of curiosity, you wanted to code a `hashCode` method yourself, how would you do it? Keeping in mind the `hashCode` contract, you could code something like Listing 13:

Listing 13. Implementing your own hashCode

```
public int hashCode() {
 int result;
 result = (int) (id ^ (id >>> 32));
 result = 31 * result + (firstName != null ? firstName.hashCode() : 0);
 result = 31 * result + (lastName != null ? lastName.hashCode() : 0);
 result = 31 * result + (emailAddress != null ? emailAddress.hashCode() : 0);
 result = 31 * result + (creationDate != null ? creationDate.hashCode() : 0);
 return result;
}
```

Needless to say, this code works as a valid `hashCode` method, but which of the two `hashCode` methods would you rather maintain? Which one can you comprehend more quickly? Note how once again, ternary statements are leveraged in Listing 13 to avoid a lot of conditional logic. You can imagine that Commons Lang's `HashCodeBuilder` is probably doing something similar — but the beauty is that the Commons Lang developers are maintaining and testing it.

Like the `EqualsBuilder`, the `HashCodeBuilder` has another API that leverages reflection. With it, you don't need to add each property of the underlying object manually with an `append` method, resulting in a `hashCode` method like the one in Listing 14:

Listing 14. Using the HashCodeBuilder's reflection API

```
public int hashCode() {
 return HashCodeBuilder.reflectionHashCode(this);
}
```

Just as before, because this method applies Java reflection under the covers, security tweaking might break the functionality, and performance might be a bit slower.

Section 5. Comparatively comparable

Another interesting method that suggests a rather formal contract is the `Comparable` interface's `compareTo` method. This interface turns out to be quite important should you require specific control over how particular objects are ordered. In this section, you'll learn how to leverage the Commons Lang `CompareToBuilder`.

Ordering output

As you've probably already noticed in your past Java programming pursuits, there are default mechanisms for how objects are ordered in certain events, such as the `Collections` class's `sort` method.

For example, the `Collection` in Listing 15 is unordered and will remain this way if you don't do anything to it:

Listing 15. A list of Strings

```
ArrayList<String> list = new ArrayList<String>();
list.add("Megan");
list.add("Zeek");
list.add("Andy");
list.add("Michelle");
```

Yet, if you pass the `list` to `Collections`' `sort` method, as I do in Listing 16, the default ordering will be applied, which in this case is alphabetical. Listing 16 sorts and prints out an alphabetically ordered list of the names in Listing 15:

Listing 16. Sorting a list of Strings

```
Collections.sort(list);

for(String value : list){
 System.out.println("sorted is " + value);
}
```

Listing 17 shows the output:

Listing 17. A sorted output of Strings

```
sorted is Andy
sorted is Megan
sorted is Michelle
```

```
sorted is Zeek
```

The reason this works, of course, is that the Java `String` class implements the `Comparable` interface and thus has an implementation of the `compareTo` method that permits sorting alphabetically. In fact, practically all core classes in the Java language implement this interface.

What if you wanted to permit a collection of `Accounts` to be ordered in various ways — for example, via `id` or last name? How would you achieve this goal?

Of course, you'd first have to implement the `Comparable` interface and then implement the `compareTo` method. This method can essentially be used only for natural ordering — thus, an object is ordered according to its properties. Consequently, `compareTo` is quite similar to the `equals` method, yet it permits a collection of `Accounts` to be sorted by their properties in the order in which the properties are processed via the `compareTo` method.

If you read the documentation for implementing this method, you'll find that it is quite similar to that of `equals`; that is, getting it correct can be tricky. (*Effective Java* devotes four pages to the subject.) By now, you've no doubt figured out the pattern: you'll just leverage Commons Lang to get it right.

Building compareTo

Commons Lang provides an aptly named `CompareToBuilder` that functions almost identically to the `EqualsBuilder`. It includes a chainable `append` method, and you can ultimately return an `int` via the `toComparison` method.

Accordingly, to get the ball rolling, you must first alter the `Account` class to implement the `Comparable` interface, as shown in Listing 18:

Listing 18. Implementing the Comparable interface

```
public class Account implements Comparable {}
```

Next, you must implement the `compareTo` method, as shown in Listing 19:

Listing 19. The default implementation of compareTo

```
public int compareTo(Object obj) {
 return 0;
}
```

Implementing this method is a two-step process. First, you must cast the incoming

parameter type to your desired type (`Account` in this case). Then you leverage the `CompareToBuilder` to compare your object's properties. The Commons Lang documentation states you should compare the same properties you compared in the `equals` method; accordingly, the `Account` object's `compareTo` method should look like Listing 20:

Listing 20. Using the CompareToBuilder

```
public int compareTo(Object obj) {
 Account account = (Account) obj;
 return new CompareToBuilder().append(this.id, account.id)
  .append(this.firstName, account.firstName)
   .append(this.lastName, account.lastName)
    .append(this.emailAddress, account.emailAddress)
     .append(this.creationDate, account.creationDate)
      .toComparison();
}
```

Don't forget, if you really want to cut down on your coding, you can always leverage the reflection-style `CompareToBuilder` API, as I've done in Listing 21:

Listing 21. Using CompareToBuilder's reflection API

```
public int compareTo(Object obj) {
 return CompareToBuilder.reflectionCompare(this, obj);
}
```

Now, if you need to rely, for example, on natural ordering for a collection of `Accounts`, you can leverage `Collections.sort`, as shown in Listing 22:

Listing 22. Sorting a comparable list of Accounts

```
Date now = new Date();
ArrayList<Account> list = new ArrayList<Account>();
list.add(new Account(41, "Amy", "Glover", "ajg@me.com", now));
list.add(new Account(10, "Andrew", "Glover", "ajg@me.com", now));
list.add(new Account(1, "Andrew", "Blover", "ajg@me.com", new Date()));
list.add(new Account(2, "Andrew", "Smith", "b@bb.com", now));
list.add(new Account(0, "Andrew", "Glover", "z@zell.com", new Date()));

Collections.sort(list);

for(Account acct : list){
 System.out.println(acct);
}
```

This code outputs the objects in a natural order according to `id`, then first name, then last name, and so on. Consequently, the sorted order would be what's shown in Listing 23:

Listing 23. A sorted list of Accounts

```
new Account(0, "Andrew", "Glover", "z@zell.com", new Date())
new Account(1, "Andrew", "Blover", "ajg@me.com", new Date())
new Account(2, "Andrew", "Smith", "b@bb.com", now)
new Account(10, "Andrew", "Glover", "ajg@me.com", now)
new Account(41, "Amy", "Glover", "ajg@me.com", now)
```

Making sense of this output is another matter. In the next section, you'll see how Commons Lang can help you build more readable results.

Section 6. String representations of objects

The default implementation of `Object`'s `toString` method returns the fully qualified name of the object followed by a @ character and then the value of the object's hash code. And as you've probably figured out long ago, this isn't very helpful in distinguishing objects from one another. Commons Lang has a handy `ToStringBuilder` class that helps build a more readable `toString` result.

Building toString

You've probably coded a `toString` method on more than one occasion — I know I have. These methods aren't complicated and it's hard to code them incorrectly; however, they could be considered a pain in the neck. And because your `Account` object is already relying on the Commons Lang library, let's go ahead and see the `ToStringBuilder` in action.

The `ToStringBuilder` acts the same way as the other three classes I've covered. You create an instance of it, append some properties, and call `toString`. That's it.

Go ahead and override the `toString` method and add the code in Listing 24:

Listing 24. Using the ToStringBuilder

```
public String toString() {
 return new ToStringBuilder(this).append("id", this.id).
  .append("firstName", this.firstName)
   .append("lastName", this.lastName)
    .append("emailAddress", this.emailAddress)
     .append("creationDate", this.creationDate)
      .toString();
}
```

As always, you can also leverage reflection, as shown in Listing 25:

Listing 25. Using ToStringBuilder's reflection API

```
public String toString() {
 return ToStringBuilder.reflectionToString(this);
}
```

Regardless of how you choose to use `ToStringBuilder`, invoking `toString` yields a more readable `String`. For example, take the object instance in Listing 26:

Listing 26. A unique Account instance

```
new Account(10, "Andrew", "Glover", "ajg@me.com", now);
```

As you can see in Listing 27, the output is quite readable:

Listing 27. ToStringBuilder's output

```
com.acme.app.Account@47858e[
   id=10,firstName=Andrew,lastName=Glover,emailAddress=ajg@me.com,
   creationDate=Tue Nov 11 17:20:08 EST 2008]
```

If you aren't happy with this particular `String` representation of your object, the Commons Lang library has a few helper classes that can assist in custom output. If anything, using the `ToStringBuilder` enables a consistent representation of object instances in log files, for example.

Section 7. Less is more

Thankfully, over the last two decades or so, the software industry started learning that too much code can be a bad thing. The studies I've cited let us assume that fewer lines of code means fewer defects.

The enormous proliferation of open source software means that the code reuse is already being realized. Although we're still hoping the day of true component reuse will arrive, we now have frameworks and supporting code that are easy to reuse, letting you write applications with as few lines of code as possible.

So what are you waiting for? Go ahead and start using the Apache Commons Lang project. While you're at it, explore what else is in this handy library. You'll find that the time and keystrokes you'll save are well worth the effort.

Downloads

Description	Name	Size	Download method
Article sample project code	j-lessismore.zip	6.9MB	HTTP

Information about download methods

Resources

Learn

- Commons Lang: Commons Lang provides a host of helper utilities for the `java.lang` API.
- *In pursuit of code quality* (Andrew Glover, developerWorks): This series explores techniques, tools, and methods for ensuring and measuring software quality.
- *Effective Java: Programming Language Guide* (Joshua Bloch, Prentice Hall, 2001): Bloch's book covers the details of `java.lang.Object` method implementations.
- "Hashing it out" (Brian Goetz, developerWorks, May 2003): This installment of the *Java theory and practice* series covers rules and guidelines for defining `hashCode` and `equals` effectively and appropriately.
- "Why Do CMMI Assessments?" (Donna Dunaway and Marilyn Bush, InformIt, June 2005): A sample chapter from *CMMI® Assessments: Motivating Positive Change* (Addison Wesley, 2005), which outlines some defect density studies.
- "Linux: Fewer Bugs Than Rivals" (Michelle Delio, Wired, December 2004): Another article that outlines some metrics regarding defect density.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Sun JDK 1.5 or later: You'll need at least version 1.5.0_09 to follow the examples in this tutorial.
- Commons Lang: Download Commons Lang.

Discuss

- Improve your code quality: Andrew Glover's developerWorks discussion for developers focused on test-driven development, code quality, and reducing risk.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur with a passion for behavior-driven development, Continuous Integration, and Agile software development. You can keep up with him at his blog.

Trademarks