Examples of Usage

Preamble

The simplicity in JOpt "Simple" arises from two guiding principles:

- Stick as often as possible to supporting conventional Unix option syntaxes.
- Keep the surface area of the published API as small and simple as possible.

To the first principle: You will not see support in JOpt Simple for option "groups", alternative option prefixes (+, /), enforced multiplicity of option arguments, etc. JOpt Simple believes you can create a useful and understandable CLI without all that stuff. If you feel as though you need any of those features, there are lots of other choices out there. The author of JOpt Simple believes you'll want to leverage its easy configuration, parsing, and option interrogation APIs instead of using more feature-laden, but perhaps more confusing libraries.

To the second principle: JOpt Simple will make every attempt to keep the API free of clutter. The API is well factored, making it intuitive to use, and the entire library is well tested, making it more reliable and predictable. If you cannot look at the Javadoc and quickly get a sense of what you need to do to use JOpt Simple, then JOpt Simple has failed. So by all means, let the author know what needs improved.

With that said, let's take a tour through JOpt Simple's features.

Options

JOpt Simple supports short options and long options, using a syntax that attempts to take from the best of POSIX getopt() and GNU getopt_long().

Short Options

Short options begin with a single hyphen (-) followed by a single letter or digit, or question mark (?), or dot (.).

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class ShortOptionsTest {
    @Test
    public void supportsShortOptions() {
        OptionParser parser = new OptionParser( "aB?*." );

        OptionSet options = parser.parse( "-a", "-B", "-?" );

        assertTrue( options.has( "a" ) );
        assertTrue( options.has( "B" ) );
        assertTrue( options.has( "?" ) );
        assertFalse( options.has( "." ) );
    }
}
```

When you construct an OptionParser with a string of short option characters, you configure that parser to recognize the options with those characters.

Arguments of Options

Short options can accept single arguments. The argument can be made required or optional. When you construct an OptionParser with a string of short option characters, append a single colon (:) to an option character to configure that option to require an argument. Append two colons (::) to an option character to configure that option to accept an optional argument. Append an asterisk (*) to an option character, but before any "argument" indicators, to configure that option as a "help" option.

The syntax of the option specification string given to the OptionParser constructor should look familiar to you if you have used GNU's getopt() before.

```
package joptsimple.examples;

import static java.util.Arrays.*;
import static java.util.Collections.*;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class ShortOptionsWithArgumentsTest {
    @Test
    public void allowsOptionsToAcceptArguments() {
        OptionParser parser = new OptionParser( "fc:q::" );

        OptionSet options = parser.parse( "-f", "-c", "foo", "-q" );

        assertTrue( options.has( "f" ) );

        assertTrue( options.has( "c" ) );
        assertTrue( options.hasArgument( "c" ) );
        assertEquals( "foo", options.valueOf( "c" ) );
        assertEquals( asList( "foo" ), options.valuesOf( "c" ) );

        assertTrue( options.has( "q" ) );
        assertFalse( options.hasArgument( "q" ) );
        assertNull( options.valueOf( "q" ) );
        assertEquals( emptyList(), options.valuesOf( "q" ) );
    }
}
```

Specifying Arguments for a Short Option on the Command Line

A short option's argument can occur:

- in the position on the command line after the option
- right up against the option
- right up against the option separated by an equals sign (=)

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class ShortOptionsWithArgumentPositioningTest {
    @Test
    public void allowsDifferentFormsOfPairingArgumentWithOption() {
        OptionParser parser = new OptionParser( "a:b:c::" );

        OptionSet options = parser.parse( "-a", "foo", "-bbar", "-c=baz" );
```

```
        assertTrue( options.has( "a" ) );
        assertTrue( options.hasArgument( "a" ) );
        assertEquals( "foo", options.valueOf( "a" ) );

        assertTrue( options.has( "b" ) );
        assertTrue( options.hasArgument( "b" ) );
        assertEquals( "bar", options.valueOf( "b" ) );

        assertTrue( options.has( "c" ) );
        assertTrue( options.hasArgument( "c" ) );
        assertEquals( "baz", options.valueOf( "c" ) );
    }
}
```

Multiple Arguments for a Single Option

To specify *n* arguments for a single option, specify the option *n* times on the command line, once for each argument. JOpt Simple reports the arguments given to the option in the order in which they were encountered on the command line.

```
package joptsimple.examples;

import static java.util.Arrays.*;

import joptsimple.OptionException;
import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

import static org.junit.Assert.*;
import static org.junit.rules.ExpectedException.*;

public class ShortOptionsWithMultipleArgumentsForSingleOptionTest {
    @Rule public final ExpectedException thrown = none();

    @Test
    public void allowsMultipleValuesForAnOption() {
        OptionParser parser = new OptionParser( "a:" );

        OptionSet options = parser.parse( "-a", "foo", "-abar", "-a=baz" );

        assertTrue( options.has( "a" ) );
        assertTrue( options.hasArgument( "a" ) );
        assertEquals( asList( "foo", "bar", "baz" ), options.valuesOf( "a" ) );

        thrown.expect( OptionException.class );
        options.valueOf( "a" );
    }
}
```

Clustering Short Options

Short options can be *clustered* in a single argument.

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;
```

```
public class ShortOptionsClusteringTest {
    @Test
    public void allowsClusteringShortOptions() {
        OptionParser parser = new OptionParser( "aBcd" );

        OptionSet options = parser.parse( "-cdBa" );

        assertTrue( options.has( "a" ) );
        assertTrue( options.has( "B" ) );
        assertTrue( options.has( "c" ) );
        assertTrue( options.has( "d" ) );
    }
}
```

If one of the short options can accept an argument, the remaining characters are interpreted as the argument for that option.

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class ShortOptionsClusteringWithArgumentTest {
    @Test
    public void allowsClusteringShortOptionsThatAcceptArguments() {
        OptionParser parser = new OptionParser();
        parser.accepts( "a" );
        parser.accepts( "B" );
        parser.accepts( "c" ).withRequiredArg();

        OptionSet options = parser.parse( "-aBcfoo" );

        assertTrue( options.has( "a" ) );
        assertTrue( options.has( "B" ) );
        assertTrue( options.has( "c" ) );
        assertEquals( "foo", options.valueOf( "c" ) );
    }
}
```

Long Options/Fluent Interface

Long options begin with two hyphens (--), followed by multiple letters, digits, hyphens, question marks, or dots. A hyphen cannot be the first character of a long option specification when configuring the parser.

Whereas short options can be configured using a constructor argument to OptionParser, both long and short options can be configured using a "fluent interface" API, that enables some very descriptive and powerful features.

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class LongOptionsTest {
    @Test
    public void acceptsLongOptions() {
        OptionParser parser = new OptionParser();
        parser.accepts( "flag" );
        parser.accepts( "verbose" );
```

```
        OptionSet options = parser.parse( "--flag" );

        assertTrue( options.has( "flag" ) );
        assertFalse( options.has( "verbose" ) );
    }
}
```

Arguments of Options

Like short options, long options can accept single arguments. The argument can be made required or optional. Use the methods withRequiredArg() and withOptionalArg() on the return value of OptionParser.accepts() to signal that an option takes a required or optional argument.

```
package joptsimple.examples;

import static java.util.Arrays.*;
import static java.util.Collections.*;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class LongOptionsWithArgumentsTest {
    @Test
    public void supportsLongOptionsWithArgumentsAndAbbreviations() {
        OptionParser parser = new OptionParser();
        parser.accepts( "flag" );
        parser.accepts( "count" ).withRequiredArg();
        parser.accepts( "level" ).withOptionalArg();

        OptionSet options = parser.parse( "-flag", "--co", "3", "--lev" );

        assertTrue( options.has( "flag" ) );

        assertTrue( options.has( "count" ) );
        assertTrue( options.hasArgument( "count" ) );
        assertEquals( "3", options.valueOf( "count" ) );
        assertEquals( asList( "3" ), options.valuesOf( "count" ) );

        assertTrue( options.has( "level" ) );
        assertFalse( options.hasArgument( "level" ) );
        assertNull( options.valueOf( "level" ) );
        assertEquals( emptyList(), options.valuesOf( "level" ) );
    }
}
```

Abbreviating Long Options

Notice in the example above that the command line uses abbreviations of command line options. You can abbreviate options so long as the abbreviation is unambiguous. Even though you can abbreviate the options on the command line, you cannot address the OptionSet using those abbreviations.

Using Single Hyphen on Long Options

As demonstrated in the example above, you can use a single hyphen instead of a double hyphen to specify a long option -- but be careful that doing so doesn't introduce ambiguity.

Specifying Arguments for a Long Option on the Command Line

A long option's argument can occur:

- in the position on the command line after the option
- right up against the option separated by an equals sign (=)

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class LongOptionsWithArgumentPositioningTest {
    @Test
    public void allowsDifferentFormsOfPairingArgumentWithOption() {
        OptionParser parser = new OptionParser();
        parser.accepts( "count" ).withRequiredArg();
        parser.accepts( "level" ).withOptionalArg();

        OptionSet options = parser.parse( "--count", "4", "--level=3" );

        assertTrue( options.has( "count" ) );
        assertTrue( options.hasArgument( "count" ) );
        assertEquals( "4", options.valueOf( "count" ) );

        assertTrue( options.has( "level" ) );
        assertTrue( options.hasArgument( "level" ) );
        assertEquals( "3", options.valueOf( "level" ) );
    }
}
```

Multiple Arguments for a Single Option

Specify multiple arguments for a long option in the same manner as for short options (see above).

Alternative Form of Long Options

The option `-W` is reserved. If you tell the parser to recognize alternative long options, then it will treat, for example, `-W foo=bar` as the long option `foo` with argument bar, as though you had written `--foo=bar`.

You can specify `-W` as a valid short option, or use it as an abbreviation for a long option, but recognizing alternative long options will always supersede this behavior.

To recognize alternative long options, either construct an `OptionParser` with a string of short option characters containing the sequence `W;` (a capital W followed by a semicolon), or call the method `OptionParser.recognizeAlternativeLongOptions()`.

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class AlternativeLongOptionsTest {
    @Test
    public void handlesAlternativeLongOptions() {
        OptionParser parser = new OptionParser( "W;" );
```

```
        parser.recognizeAlternativeLongOptions( true );  // same effect as above
        parser.accepts( "level" ).withRequiredArg();

        OptionSet options = parser.parse( "-W", "level=5" );

        assertTrue( options.has( "level" ) );
        assertTrue( options.hasArgument( "level" ) );
        assertEquals( "5", options.valueOf( "level" ) );
    }
}
```

Other Features

Converting Option Arguments to Other Types

Without action other than the with*Arg() methods, arguments of options are returned as Strings. For backwards compatibility, OptionSet.valueOf(String) and OptionSet.valuesOf(String) return Object and List<?>, respectively, so to get the values out as Strings, you will need to downcast the results of those methods.

You can tell JOpt Simple to convert the arguments of options to different Java types via the ofType() method on the return value of with*Arg(). The Class argument of ofType() must represent a Java class that has either:

- a public static method called valueOf() which accepts a single String argument and whose return type is the type itself, or
- a public constructor which takes a single String argument.

If the class has both, the valueOf() method is used.

Note that enums have a valueOf() method.

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class OptionArgumentValueTypeTest {
    @Test
    public void convertsArgumentsToJavaValueTypes() {
        OptionParser parser = new OptionParser();
        parser.accepts( "flag" );
        parser.accepts( "count" ).withRequiredArg().ofType( Integer.class );
        parser.accepts( "level" ).withOptionalArg().ofType( Level.class );

        OptionSet options = parser.parse( "--count", "3", "--level", "DEBUG" );

        assertTrue( options.has( "count" ) );
        assertTrue( options.hasArgument( "count" ) );
        assertEquals( Integer.valueOf( 3 ), options.valueOf( "count" ) );

        assertTrue( options.has( "level" ) );
        assertTrue( options.hasArgument( "level" ) );
        assertEquals( Level.DEBUG, options.valueOf( "level" ) );
    }
}
```

Another way to convert arguments of options is to specify a converter object via withValuesConvertedBy(). This is useful when the desired type for the arguments does not meet the requirements that ofType() sets forth. Such objects may not perform any "conversion" at all, but rather can validate that arguments conform to certain restrictions before passing through as-is.

```
package joptsimple.examples;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.joda.time.DateMidnight;
import org.junit.Test;

import static joptsimple.util.DateConverter.*;
import static joptsimple.util.RegexMatcher.*;
import static org.junit.Assert.*;

public class OptionArgumentConverterTest {
    @Test
    public void usesConvertersOnOptionArgumentsWhenTold() {
        OptionParser parser = new OptionParser();
        parser.accepts( "birthdate" ).withRequiredArg().withValuesConvertedBy( datePattern( "MM/dd/yy" ) );
        parser.accepts( "ssn" ).withRequiredArg().withValuesConvertedBy( regex( "\\d{3}-\\d{2}-\\d{4}" ));

        OptionSet options = parser.parse( "--birthdate", "02/24/05", "--ssn", "123-45-6789" );

        assertEquals( new DateMidnight( 2005, 2, 24 ).toDate(), options.valueOf( "birthdate" ) );
        assertEquals( "123-45-6789", options.valueOf( "ssn" ) );
    }
}
```

Retrieving Arguments of Options in a Type-Safe Manner

In the previous examples, we have been discarding the return values of the methods of JOpt Simple's fluent interface. If instead you retain them in variables of type OptionSpec, you can use them to retrieve arguments of options in a type-safe manner.

```
package joptsimple.examples;

import java.io.File;

import static java.util.Arrays.*;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import joptsimple.OptionSpec;
import org.junit.Test;

import static org.junit.Assert.*;

public class TypesafeOptionArgumentRetrievalTest {
    @Test
    public void allowsTypesafeRetrievalOfOptionArguments() {
        OptionParser parser = new OptionParser();
        OptionSpec<Integer> count = parser.accepts( "count" ).withRequiredArg().ofType( Integer.class );
        OptionSpec<File> file = parser.accepts( "file" ).withOptionalArg().ofType( File.class );
        OptionSpec<Void> verbose = parser.accepts( "verbose" );

        OptionSet options = parser.parse( "--count", "3", "--file", "/tmp", "--verbose" );

        assertTrue( options.has( verbose ) );

        assertTrue( options.has( count ) );
        assertTrue( options.hasArgument( count ) );
        Integer expectedCount = 3;
        assertEquals( expectedCount, options.valueOf( count ) );
        assertEquals( expectedCount, count.value( options ) );
        assertEquals( asList( expectedCount ), options.valuesOf( count ) );
        assertEquals( asList( expectedCount ), count.values( options ) );

        assertTrue( options.has( file ) );
```

```
        assertTrue( options.hasArgument( file ) );
        File expectedFile = new File( "/tmp" );
        assertEquals( expectedFile, options.valueOf( file ) );
        assertEquals( expectedFile, file.value( options ) );
        assertEquals( asList( expectedFile ), options.valuesOf( file ) );
        assertEquals( asList( expectedFile ), file.values( options ) );
    }
}
```

Default Values for Option Arguments

Often it is convenient to specify default values for the arguments of certain command line options. To do this, call the defaultsTo() method.

```
package joptsimple.examples;

import java.io.File;

import joptsimple.OptionException;
import joptsimple.OptionParser;
import joptsimple.OptionSet;
import joptsimple.OptionSpec;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

import static joptsimple.examples.Level.*;
import static org.junit.Assert.*;
import static org.junit.rules.ExpectedException.*;

public class DefaultValuesForOptionArgumentsTest {
    @Rule public final ExpectedException thrown = none();

    @Test
    public void allowsSpecificationOfDefaultValues() throws Exception {
        File tempDir = new File( System.getProperty( "java.io.tmpdir" ) );
        File tempFile = File.createTempFile( "aFile", ".txt" );
        OptionParser parser = new OptionParser();
        OptionSpec<File> infile =
            parser.accepts( "infile" ).withRequiredArg().ofType( File.class ).defaultsTo( tempFile );
        OptionSpec<File> outdir =
            parser.accepts( "outdir" ).withRequiredArg().ofType( File.class ).defaultsTo( tempDir );
        OptionSpec<Integer> bufferSize =
            parser.accepts( "buffer-size" ).withOptionalArg().ofType( Integer.class ).defaultsTo( 4096 );
        OptionSpec<Level> level =
            parser.accepts( "level" ).withOptionalArg().ofType( Level.class ).defaultsTo( INFO );
        OptionSpec<Integer> count =
            parser.accepts( "count" ).withOptionalArg().ofType( Integer.class ).defaultsTo( 10 );

        OptionSet options = parser.parse( "--level", "WARNING", "--count", "--infile", "/etc/passwd" );

        assertEquals( new File( "/etc/passwd" ), infile.value( options ) );
        assertTrue( options.has( infile ) );
        assertTrue( options.hasArgument( infile ) );
        assertEquals( tempDir, outdir.value( options ) );
        assertFalse( options.has( outdir ) );
        assertFalse( options.hasArgument( outdir ) );
        assertEquals( Integer.valueOf( 4096 ), bufferSize.value( options ) );
        assertFalse( options.has( bufferSize ) );
        assertFalse( options.hasArgument( bufferSize ) );
        assertEquals( WARNING, level.value( options ) );
        assertTrue( options.has( level ) );
        assertTrue( options.hasArgument( level ) );
        assertEquals( Integer.valueOf( 10 ), count.value( options ) );
        assertTrue( options.has( count ) );
        assertFalse( options.hasArgument( count ) );
```

```
        thrown.expect( OptionException.class );

        parser.parse( "--outdir" );
    }
}
```

You can see that defaultsTo() should relieve you of the burden of having to check has() and/or hasArgument() on an OptionSet for a given option, and has no bearing on the return values of those methods. Specifying a default value for an option with a required argument does not mean that you can elide an argument for the option on the command line.

The type of values defaultsTo() expects is dictated by the class given by a previous call to ofType() or withValuesConvertedBy(); if no such call has been made, the type is String.

"Required" Options

You can indicate that a given option must be present on the command line via the required() method. Only options that accept arguments can be made "required".

An option designated as a "help" option via forHelp(), when present on the command line, causes missing "required" options not to reject the command line.

```
package joptsimple.examples;

import joptsimple.OptionException;
import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class RequiredOptionsTest {
    @Test( expected = OptionException.class )
    public void allowsSpecificationOfRequiredOptions() {
        OptionParser parser = new OptionParser() {
            {
                accepts( "userid" ).withRequiredArg().required();
                accepts( "password" ).withRequiredArg().required();
            }
        };

        parser.parse( "--userid", "bob" );
    }

    @Test
    public void aHelpOptionMeansRequiredOptionsNeedNotBePresent() {
        OptionParser parser = new OptionParser() {
            {
                accepts( "userid" ).withRequiredArg().required();
                accepts( "password" ).withRequiredArg().required();
                accepts( "help" ).forHelp();
            }
        };

        OptionSet options = parser.parse( "--help" );
        assertTrue( options.has( "help" ) );
    }

    @Test( expected = OptionException.class )
    public void missingHelpOptionMeansRequiredOptionsMustBePresent() {
        OptionParser parser = new OptionParser() {
```

```
            {
                accepts( "userid" ).withRequiredArg().required();
                accepts( "password" ).withRequiredArg().required();
                accepts( "help" ).forHelp();
            }
        };

        parser.parse( "" );
    }
}
```

"Required" Dependent Options

You can indicate that a given option must be present on the command line if some other option is present on the command line via the requiredIf() method. Any option can be made "required if".

An option designated as a "help" option via forHelp(), when present on the command line, causes missing "required if" options not to reject the command line.

```
package joptsimple.examples;

import joptsimple.OptionParser;

public class RequiredIfExample {
    public static void main( String[] args ) {
        OptionParser parser = new OptionParser();
        parser.accepts( "ftp" );
        parser.accepts( "username" ).requiredIf( "ftp" ).withRequiredArg();
        parser.accepts( "password" ).requiredIf( "ftp" ).withRequiredArg();

        parser.parse( "--ftp" );
    }
}
```

Synonyms of Options

Sometimes it is useful to allow many different options to share the same meaning in the program that uses them. To specify that options are to be treated as synonymous, use the acceptsAll() method of OptionParser.

```
package joptsimple.examples;

import java.util.List;

import static java.util.Arrays.*;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class OptionSynonymTest {
    @Test
    public void supportsOptionSynonyms() {
        OptionParser parser = new OptionParser();
        List<String> synonyms = asList( "message", "blurb", "greeting" );
        parser.acceptsAll( synonyms ).withRequiredArg();
        String expectedMessage = "Hello";

        OptionSet options = parser.parse( "--message", expectedMessage );

        for ( String each : synonyms ) {
```

```
            assertTrue( each, options.has( each ) );
            assertTrue( each, options.hasArgument( each ) );
            assertEquals( each, expectedMessage, options.valueOf( each ) );
            assertEquals( each, asList( expectedMessage ), options.valuesOf( each ) );
        }
    }
}
```

Concise Specification of Multiple Arguments for an Option

Another way to specify multiple arguments for an option is to tell the parser to treat a single argument containing multiple delimited values as multiple arguments for the option using the withValuesSeparatedBy() method.

```
package joptsimple.examples;

import java.io.File;

import static java.io.File.*;
import static java.util.Arrays.*;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import joptsimple.OptionSpec;
import org.junit.Test;

import static joptsimple.examples.Strings.*;
import static org.junit.Assert.*;

public class MultipleDelimitedArgumentsTest {
    @Test
    public void supportsMultipleDelimitedArguments() {
        OptionParser parser = new OptionParser();
        OptionSpec<File> path = parser.accepts( "path" ).withRequiredArg().ofType( File.class )
            .withValuesSeparatedBy( pathSeparatorChar );

        OptionSet options = parser.parse( "--path", join( pathSeparatorChar, "/tmp", "/var", "/opt" ) );

        assertTrue( options.has( path ) );
        assertTrue( options.hasArgument( path ) );
        assertEquals( asList( new File( "/tmp" ), new File( "/var" ), new File( "/opt" ) ), options.valuesOf( path ) );
    }
}
```

Signalling End of Options

An argument consisting only of two hyphens (--) signals that the remaining arguments are to be treated as non-options.

An argument consisting only of a single hyphen is considered a non-option argument (though it can be an argument of an option). Many Unix programs treat single hyphens as stand-ins for the standard input or standard output stream.

Non-Option Arguments

Any arguments which are not options or arguments of options can be retrieved via method nonOptionArguments() on OptionSet. If the double hyphen is an argument, it is ignored and is not a non-option argument.

```
package joptsimple.examples;

import static java.util.Arrays.*;
```

```
import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class SignallingEndOfOptionsTest {
    @Test
    public void doubleHyphenSignalsEndOfOptions() {
        OptionParser parser = new OptionParser( "ab:c::de:f::" );

        OptionSet options = parser.parse( "-a", "-b=foo", "-c=bar", "--", "-d", "-e", "baz", "-f", "biz" );

        assertTrue( options.has( "a" ) );
        assertFalse( options.hasArgument( "a" ) );
        assertTrue( options.has( "b" ) );
        assertTrue( options.hasArgument( "b" ) );
        assertEquals( asList( "foo" ), options.valuesOf( "b" ) );
        assertTrue( options.has( "c" ) );
        assertTrue( options.hasArgument( "c" ) );
        assertEquals( asList( "bar" ), options.valuesOf( "c" ) );
        assertFalse( options.has( "d" ) );
        assertFalse( options.has( "e" ) );
        assertFalse( options.has( "f" ) );
        assertEquals( asList( "-d", "-e", "baz", "-f", "biz" ), options.nonOptionArguments() );
    }
}
```

"POSIX-ly Correct"-ness

By default, as with GNU getopt(), JOpt Simple allows intermixing of options and non-options. If, however, the parser has been created to be "POSIX-ly correct", then the first argument that does not look lexically like an option, and is not a required argument of a preceding option, signals the end of options. You can still bind optional arguments to their options using the abutting (for short options) or = syntax.

Unlike GNU getopt(), JOptSimple does not honor the environment variable POSIXLY_CORRECT. "POSIX-ly correct" parsers are configured by either:

- using the method OptionParser.posixlyCorrect()
- using the OptionParser constructor with an argument whose first character is a plus sign (+)

```
package joptsimple.examples;

import static java.util.Arrays.*;
import static java.util.Collections.*;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class PosixlyCorrectTest {
    @Test
    public void supportsPosixlyCorrectBehavior() {
        OptionParser parser = new OptionParser( "i:j::k" );
        String[] arguments = { "-ibar", "-i", "junk", "xyz", "-jixnay", "foo", "-k", "blah", "--", "bah" };

        OptionSet options = parser.parse( arguments );

        assertTrue( options.has( "i" ) );
        assertTrue( options.has( "j" ) );
        assertTrue( options.has( "k" ) );
        assertEquals( asList( "bar", "junk" ), options.valuesOf( "i" ) );
```

```
        assertEquals( asList( "ixnay" ), options.valuesOf( "j" ) );
        assertEquals( asList( "xyz", "foo", "blah", "bah" ), options.nonOptionArguments() );

        parser.posixlyCorrect( true );
        options = parser.parse( arguments );

        assertTrue( options.has( "i" ) );
        assertFalse( options.has( "j" ) );
        assertFalse( options.has( "k" ) );
        assertEquals( asList( "bar", "junk" ), options.valuesOf( "i" ) );
        assertEquals( emptyList(), options.valuesOf( "j" ) );
        assertEquals( asList( "xyz", "-jixnay", "foo", "-k", "blah", "--", "bah" ), options.nonOptionArguments() );
    }
}
```

Special Optional Argument Handling

If the parser detects an option whose argument is optional, and the next argument "looks like" an option, that argument is not treated as the argument to the option, but as a potentially valid option. If, on the other hand, the optional argument is typed as a derivative of Number, then that argument is treated as the negative number argument of the option, even if the parser recognizes the corresponding numeric option.

```
package joptsimple.examples;

import static java.util.Arrays.*;
import static java.util.Collections.*;

import joptsimple.OptionParser;
import joptsimple.OptionSet;
import org.junit.Test;

import static org.junit.Assert.*;

public class SpecialOptionalArgumentHandlingTest {
    @Test
    public void handlesNegativeNumberOptionalArguments() {
        OptionParser parser = new OptionParser();
        parser.accepts( "a" ).withOptionalArg().ofType( Integer.class );
        parser.accepts( "2" );

        OptionSet options = parser.parse( "-a", "-2" );

        assertTrue( options.has( "a" ) );
        assertFalse( options.has( "2" ) );
        assertEquals( asList( -2 ), options.valuesOf( "a" ) );

        options = parser.parse( "-2", "-a" );

        assertTrue( options.has( "a" ) );
        assertTrue( options.has( "2" ) );
        assertEquals( emptyList(), options.valuesOf( "a" ) );
    }
}
```

Generating Command Line Help

When you call method OptionParser.printHelpOn(), JOpt Simple will write a help screen (80-column width) describing all the options it is configured with, along with types of option arguments, whether the option is required (in angle brackets) or optional (in square brackets), etc. To give an option a description, use OptionParser.accepts*() with a description argument. To give an option argument a description, use describedAs() on the return value of with*Arg().

```
package joptsimple.examples;

import java.io.File;

import static java.io.File.*;
import static java.util.Arrays.*;

import joptsimple.OptionParser;

import static joptsimple.util.DateConverter.*;

public class HelpScreenExample {
    public static void main( String[] args ) throws Exception {
        OptionParser parser = new OptionParser() {
            {
                accepts( "c" ).withRequiredArg().ofType( Integer.class )
                    .describedAs( "count" ).defaultsTo( 1 );
                accepts( "q" ).withOptionalArg().ofType( Double.class )
                    .describedAs( "quantity" );
                accepts( "d", "some date" ).withRequiredArg().required()
                    .withValuesConvertedBy( datePattern( "MM/dd/yy" ) );
                acceptsAll( asList( "v", "talkative", "chatty" ), "be more verbose" );
                accepts( "output-file" ).withOptionalArg().ofType( File.class )
                     .describedAs( "file" );
                acceptsAll( asList( "h", "?" ), "show help" ).forHelp();
                acceptsAll( asList( "cp", "classpath" ) ).withRequiredArg()
                    .describedAs( "path1" + pathSeparatorChar + "path2:..." )
                    .ofType( File.class )
                    .withValuesSeparatedBy( pathSeparatorChar );
            }
        };

        parser.printHelpOn( System.out );
    }
}
```

Here is what the help screen looks like for the example above:

```
Option (* = required)            Description
---------------------            -----------
-?, -h                           show help
-c <Integer: count>              (default: 1)
--classpath, --cp <File: path1:
  path2:...>
* -d <MM/dd/yy>                  some date
--output-file [File: file]
-q [Double: quantity]
-v, --chatty, --talkative        be more verbose
```

If you want to create your own help screen, give method OptionParser.formatHelpWith() a HelpFormatter that builds the help screen as a String. When you call OptionParser.printHelpOn(), JOpt Simple will use your HelpFormatter to produce the help and write it to the given stream.

For example, this program:

```
package joptsimple.examples;

import java.io.File;
import java.util.HashSet;
import java.util.Map;

import static java.io.File.*;
import static java.util.Arrays.*;

import joptsimple.HelpFormatter;
import joptsimple.OptionDescriptor;
```

```
import joptsimple.OptionParser;

import static joptsimple.util.DateConverter.*;

public class HelpFormatterExample {
    static class MyFormatter implements HelpFormatter {
        public String format( Map<String, ? extends OptionDescriptor> options ) {
            StringBuilder buffer = new StringBuilder();
            for ( OptionDescriptor each : new HashSet<OptionDescriptor>( options.values() ) ) {
                buffer.append( lineFor( each ) );
            }
            return buffer.toString();
        }

        private String lineFor( OptionDescriptor descriptor ) {
            StringBuilder line = new StringBuilder( descriptor.options().toString() );
            line.append( ": description = " ).append( descriptor.description() );
            line.append( ", required = " ).append( descriptor.isRequired() );
            line.append( ", accepts arguments = " ).append( descriptor.acceptsArguments() );
            line.append( ", requires argument = " ).append( descriptor.requiresArgument() );
            line.append( ", argument description = " ).append( descriptor.argumentDescription() );
            line.append( ", argument type indicator = " ).append( descriptor.argumentTypeIndicator() );
            line.append( ", default values = " ).append( descriptor.defaultValues() );
            line.append( System.getProperty( "line.separator" ) );
            return line.toString();
        }
    }

    public static void main( String[] args ) throws Exception {
        OptionParser parser = new OptionParser() {
            {
                accepts( "c" ).withRequiredArg().ofType( Integer.class )
                    .describedAs( "count" ).defaultsTo( 1 );
                accepts( "q" ).withOptionalArg().ofType( Double.class )
                    .describedAs( "quantity" );
                accepts( "d", "some date" ).withRequiredArg().required()
                    .withValuesConvertedBy( datePattern( "MM/dd/yy" ) );
                acceptsAll( asList( "v", "talkative", "chatty" ), "be more verbose" );
                accepts( "output-file" ).withOptionalArg().ofType( File.class )
                     .describedAs( "file" );
                acceptsAll( asList( "h", "?" ), "show help" ).forHelp();
                acceptsAll( asList( "cp", "classpath" ) ).withRequiredArg()
                    .describedAs( "path1" + pathSeparatorChar + "path2:..." )
                    .ofType( File.class )
                    .withValuesSeparatedBy( pathSeparatorChar );
            }
        };

        parser.formatHelpWith( new MyFormatter() );
        parser.printHelpOn( System.out );
    }
}
```

yields the following output:

```
[output-file]: description = , required = false, accepts arguments = true, requires argument = false, argument description = file, argument type indicato
[?, h]: description = show help, required = false, accepts arguments = false, requires argument = false, argument description = , argument type indicator
[q]: description = , required = false, accepts arguments = true, requires argument = false, argument description = quantity, argument type indicator = ja
[d]: description = some date, required = true, accepts arguments = true, requires argument = true, argument description = , argument type indicator = MM/
[v, chatty, talkative]: description = be more verbose, required = false, accepts arguments = false, requires argument = false, argument description = , a
[c]: description = , required = false, accepts arguments = true, requires argument = true, argument description = count, argument type indicator = java.l
[classpath, cp]: description = , required = false, accepts arguments = true, requires argument = true, argument description = path1:path2:..., argument t
```

Handling Exceptions

JOpt Simple's classes raise some derivative of OptionException if they encounter problems during parsing. These exceptions are unchecked, so you don't have to

do anything with such an exception if you don't want to. The rationale behind this decision is that you will most likely be invoking JOpt Simple's functionality from a main() method or very near to it, where a failure such as unrecognized arguments can just stop down the JVM and yield a stack trace without much user or programmer inconvenience. So, without any exception handling at all, a user would see something like this:

```
Exception in thread "main" joptsimple.UnrecognizedOptionException: 'x' is not a recognized option
        at joptsimple.OptionException.unrecognizedOption(OptionException.java:89)
        at joptsimple.OptionParser.validateOptionCharacters(OptionParser.java:538)
        at joptsimple.OptionParser.handleShortOptionCluster(OptionParser.java:463)
        at joptsimple.OptionParser.handleShortOptionToken(OptionParser.java:458)
        at joptsimple.OptionParserState$2.handleArgument(OptionParserState.java:56)
        at joptsimple.OptionParser.parse(OptionParser.java:385)
        at joptsimple.examples.ExceptionExample.main(ExceptionExample.java:9)
```

If you want to handle the exception yourself, you can just catch OptionException in your code, and do whatever you please with the contents of the exception, perhaps using the help generation facility.