JAVA 19 FEATURES
(WITH EXAMPLES)

Sven Woltmann

Last update: November 27, 2024

Java 19 has been released on September 20, 2022. You can download Java 19 here.

The most exciting new feature for me is virtual threads, which have been under development for several years within Project Loom and are now finally included as a preview in the JDK.

Virtual threads are a prerequisite for Structured Concurrency, another exciting new incubator feature in Java 19.

For those who need to access non-Java code (e.g., the C standard library), there is also good news: The Foreign Function & Memory API has reached the preview stage after five incubator rounds.

Contents [hide]

New Methods to Create Preallocated HashMaps

If we want to create an *ArrayList* for a known number of elements (e.g., 120), we can do it as follows since ever:

```
List<String> list = new ArrayList<>(120);
```

Thus the array underlying the *ArrayList* is allocated directly for 120 elements and does not have to be enlarged several times (i.e., newly created and copied) to insert the 120 elements.

Similarly, we have always been able to generate a *HashMap* as follows:

```
Map<String, Integer> map = new HashMap<>(120);
```

Intuitively, one would think that this *HashMap* offers space for 120 mappings.

However, this is not the case!

This is because the *HashMap* is initialized with a default load factor of 0.75. This means that as soon as the HashMap is 75% full, it is rebuilt ("rehashed") with double the size. This ensures that the elements are distributed as evenly as possible across the *HashMap*'s buckets and that as few buckets as possible contain more than one element.

Thus, the *HashMap* initialized with a capacity of 120 can only hold 120 × 0.75 = 90 mappings.

To create a *HashMap* for 120 mappings, you had to calculate the capacity by dividing the number of mappings by the load factor: 120 ÷ 0.75 = 160.

So a *HashMap* for 120 mappings had to be created as follows:

```
// for 120 mappings: 120 / 0.75 = 160
Map<String, Integer> map = new HashMap<>(160);
```

Java 19 makes it easier for us – we can now write the following instead:

```
Map<String, Integer> map = HashMap.newHashMap(120);
```

If we look at the source code of the new methods, we see that they do the same as we did before:

```
public static <K, V> HashMap<K, V> newHashMap(int numMappings) {
    return new HashMap<>(calculateHashMapCapacity(numMappings));
}

static final float DEFAULT_LOAD_FACTOR = 0.75f;

static int calculateHashMapCapacity(int numMappings) {
    return (int) Math.ceil(numMappings / (double) DEFAULT_LOAD_FACTOR);
}
```

The *newHashMap()* method has also been added to *LinkedHashMap* and *WeakHashMap*.

There is no JDK enhancement proposal for this extension.

Preview- und Incubator-Features

Java 19 provides us with six preview and incubator features, i.e., features that have not yet been completed but can already be tested by the developer community. The feedback from the community is usually incorporated into the further development and completion of these features.

Pattern Matching for switch (Third Preview) – JEP 427

Let's start with a feature that has already gone through two rounds of previews. First introduced in Java 17, "Pattern Matching for switch" allowed us to write code like the following:

```
switch (obj) {
  case String s && s.length() > 5 → System.out.println(s.toUpperCase()
  case String s                   → System.out.println(s.toLowerCase()
```

```
    case Integer i                  -> System.out.println(i * i);

    default -> {}
  }
```

We can check within a *switch* statement if an object is of a particular class and if it has additional characteristics (like in the example: longer than five characters).

In Java 19, JDK Enhancement Proposal 427 changed the syntax of the so-called "Guarded Pattern" (in the example above "*String s && s.length() > 5*"). Instead of *&&*, we now have to use the new keyword *when*.

The example from above is notated in Java 19 as follows:

```
switch (obj) {
  case String s when s.length() > 5 -> System.out.println(s.toUpperCase(
  case String s                     -> System.out.println(s.toLowerCase(

  case Integer i                    -> System.out.println(i * i);

  default -> {}
}
```

when is a so-called "contextual keyword" and therefore only has a meaning within a *case* label. If you have variables or methods with the name "when" in your code, you don't need to change them.

Record Patterns (Preview) – JEP 405

We stay with the topic "pattern matching" and come to "record patterns". If the subject "records" is new to you, I recommend reading the article "Records in Java" first.

I'll best explain what a record pattern is with an example. Let's assume we have defined the following record:

```
public record Position(int x, int y) {}
```

We also have a *print()* method that can print any object, including positions:

```
private void print(Object object) {
  if (object instanceof Position position) {
    System.out.println("object is a position, x = " + position.x()
                                          + ", y = " + position.y());
  }
  // else ...
}
```

If you stumble over the notation used – it was introduced in Java 16 as "Pattern Matching for instanceof".

Record Pattern with instanceof

As of Java 19, JDK Enhancement Proposal 405 allows us to use a so-called "record pattern". This allows us to write the code as follows:

```
private void print(Object object) {
  if (object instanceof Position(int x, int y)) {
    System.out.println("object is a position, x = " + x + ", y = " + y)
  }
  // else ...
}
```

Instead of matching on "*Position position*" and accessing *position* in the following code, we now match on "*Position(int x, int y)*" and can then access *x* and *y* directly.

Record Pattern with switch

Since Java 17, we can also write the original example as a *switch* statement:

```
private void print(Object object) {
  switch (object) {
    case Position position
        -> System.out.println("object is a position, x = " + position.x(
                                                + ", y = " + position.y(
    // other cases ...
  }
}
```

We can now also use a record pattern in the *switch* statement:

```
private void print(Object object) {
  switch (object) {
    case Position(int x, int y)
        -> System.out.println("object is a position, x = " + x + ", y =

    // other cases ...
  }
}
```

Nested Record Patterns

It is also possible to match nested records – let me demonstrate this with another example.

We first define a second record, *Path*, with a start position and a destination position:

```
public record Path(Position from, Position to) {}
```

Our *print()* method can now use a record pattern to print all the path's X and Y coordinates easily:

```
private void print(Object object) {
  if (object instanceof Path(Position(int x1, int y1), Position(int x2,
    System.out.println("object is a path, x1 = " + x1 + ", y1 = " + y1
                                      + ", x2 = " + x2 + ", y2 = " + y2)
  }
  // else ...
}
```

We can also write this alternatively as a *switch* statement:

```
private void print(Object object) {
  switch (object) {
    case Path(Position(int x1, int y1), Position(int x2, int y2))
        → System.out.println("object is a path, x1 = " + x1 + ", y1 = "
                                          + ", x2 = " + x2 + ", y2 = "
    // other cases ...
  }
}
```

Record patterns thus provide us with an elegant way to access the record's elements after a type check.

Virtual Threads (Preview) – JEP 425

The most exciting innovation in Java 19 for me is "Virtual Threads". Virtual threads have been developed in Project Loom for several years and could only be tested with a self-compiled JDK so far.

With JDK Enhancement Proposal 425, virtual threads finally make their way into the official JDK – and they do so directly in the preview stage, so no more significant changes to the API are expected.

To find out why we need virtual threads, what they are, how they work, and how to use them, check out the main article on virtual threads. You definitely shouldn't miss it.

Structured Concurrency (Incubator) – JEP 428

Also developed in Project Loom and initially released as an incubator feature in Java 19 with JDK Enhancement Proposal 428 is the so-called "Structured Concurrency."

When a task consists of several subtasks that can be processed in parallel, Structured Concurrency allows us to implement this in a particularly readable and maintainable way.

You can learn more about how this works in the main article about Structured Concurrency.

Foreign Function & Memory API (Preview) – JEP 424

In Project Panama, a replacement for the cumbersome, error-prone, and slow Java Native Interface (JNI) has been in the works for a long time.

The "Foreign Memory Access API" and the "Foreign Linker API" were already introduced in Java 14 and Java 16 – both initially individually in the incubator stage. In Java 17, these APIs were combined to form the "Foreign Function & Memory API" (FFM API), which remained in the incubator stage until Java 18.

In Java 19, JDK Enhancement Proposal 424 finally promoted the new API to the preview stage, which means that only minor changes and bug fixes will be made. So it's time to introduce the new API!

The Foreign Function & Memory API enables access to native memory (i.e., memory outside the Java heap) and access to native code (e.g., C libraries) directly from Java.

I will show how this works with an example. However, I won't go too deep into the topic here since most Java developers rarely (or never) need to access native memory and code.

Here is a simple example that stores a string in off-heap memory and calls the "strlen" function of the C standard library on it:

```
1  public class FFMTest {
2    public static void main(String[] args) throws Throwable {
3      // 1. Get a lookup object for commonly used libraries
4      SymbolLookup stdlib = Linker.nativeLinker().defaultLookup();
5
6      // 2. Get a handle to the "strlen" function in the C standard
7      MethodHandle strlen = Linker.nativeLinker().downcallHandle(
8          stdlib.lookup("strlen").orElseThrow(),
9          FunctionDescriptor.of(JAVA_LONG, ADDRESS));
10
11     // 3. Convert Java String to C string and store it in off-heap
12     MemorySegment str = implicitAllocator().allocateUtf8String("Ha
13
14     // 4. Invoke the foreign function
15     long len = (long) strlen.invoke(str);
16
17     System.out.println("len = " + len);
18   }
19 }
```

Interesting is the *FunctionDescriptor* in line 9: it expects as the first parameter the return type of the function and as additional parameters the function's arguments. The *FunctionDescriptor* ensures that all Java types are adequately converted to C types and vice versa.

Since the FFM API is still in the preview stage, we must specify a few additional parameters to compile and start it:

```
$ javac --enable-preview --source 19 FFMTest.java
$ java --enable-preview FFMTest
```

Anyone who has worked with JNI – and remembers how much Java and C boilerplate code you had to write and keep in sync – will realize that the effort required to call the native function has been reduced by orders of magnitude.

If you want to delve deeper into the matter: you can find more complex examples in the JEP.

Vector API (Fourth Incubator) – JEP 426

The new Vector API has nothing to do with the *java.util.Vector* class. In fact, it is about a new API for mathematical vector computation and its mapping to modern SIMD (Single-Instruction-Multiple-Data) CPUs.

The Vector API has been part of the JDK since Java 16 as an incubator and was further developed in Java 17 and Java 18.

With JDK Enhancement Proposal 426, Java 19 delivers the fourth iteration in which the API has been extended to include new vector operations – as well as the ability to store vectors in and read them from memory segments (a feature of the Foreign Function & Memory API).

Incubator features may still be subject to significant changes, so that I won't present the API in detail here. I will do that as soon as the Vector API has moved to the preview stage.

Deprecations and Deletions

In Java 19, some functions have been marked as "deprecated" or made inoperable.

Deprecation of Locale class constructors

In Java 19, the public constructors of the *Locale* class were marked as "deprecated".

Instead, we should use the new static factory method *Locale.of()*. This ensures that there is only one instance per *Locale* configuration.

The following example shows the use of the factory method compared to the constructor:

```
Locale german1 = new Locale("de"); // deprecated
Locale germany1 = new Locale("de", "DE"); // deprecated

Locale german2 = Locale.of("de");
Locale germany2 = Locale.of("de", "DE");

System.out.println("german1  == Locale.GERMAN  = " + (german1 == Locale
System.out.println("germany1 == Locale.GERMANY = " + (germany1 == Locale
System.out.println("german2  == Locale.GERMAN  = " + (german2 == Locale
System.out.println("germany2 == Locale.GERMANY = " + (germany2 == Locale
```

When you run this code, you will see that the objects supplied via the factory method are identical to the *Locale* constants – those created via constructs logically are not.

java.lang.ThreadGroup is degraded

In Java 14 and Java 16, several *Thread* and *ThreadGroup* methods were marked as "deprecated for removal". The reasons are explained in the linked sections.

The following of these methods have been decommissioned in Java 19:

- *ThreadGroup.destroy()* – invocations of this method will be ignored.
- *ThreadGroup.isDestroyed()* – always returns *false*.

- *ThreadGroup.setDaemon()* – sets the *daemon* flag, but this has no effect anymore.
- *ThreadGroup.getDaemon()* – returns the value of the unused *daemon* flags.
- *ThreadGroup.suspend()*, *resume()*, and *stop()* throw an *UnsupportedOperationException*.

Other Changes in Java 19

In this section, you will find changes/enhancements that might not be relevant for all Java developers.

Automatic Generation of the CDS Archive

Application Class Data Sharing (short: " "Application CDS" or "AppCDS") was introduced in Java 10, the configuration was significantly simplified in Java 13.

Application CDS makes it possible to load the classes of an application into the memory once when operating several JVMs on one machine and to share this memory area with all JVMs. This saves memory and time for loading the *.jar* and *.class* files and converting them into a platform-specific binary format.

With Java 19, the configuration of AppCDS has been simplified once again. You can now specify the following VM parameter to automatically create or update a CDS archive.

The application from the examples in the Java 10 and 13 articles linked above can now be started as follows:

```
java -XX:+AutoCreateSharedArchive -XX:SharedArchiveFile=helloworld.jsa \
    -cp target/helloworld.jar eu.happycoders.appcds.Main
```

The shared archive will now be created if it does not exist or if it was created by an older Java version.

Linux/RISC-V Port – JEP 422

Due to the increasing use of RISC-V hardware, a port for the corresponding architecture was made available with JEP 422.

Additional Date-Time Formats

We can use the *DateTimeFormatter.ofLocalizedDate(...)*, *ofLocalizedTime(...)*, and *ofLocalizedDateTime(...)* methods and the subsequent call to *withLocale(...)* to generate a date/time formatter. We control the exact format using the *FormatStyle* enum, which can take the values *FULL*, *LONG*, *MEDIUM*, and *SHORT*.

In Java 19, the method *ofLocalizedPattern(String requestedTemplate)* was added, with which we can also define flexible formats. Here is an example:

```
LocalDate now = LocalDate.now();
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedPattern("yMMM
System.out.println("US:      " + formatter.withLocale(Locale.US).format(
System.out.println("Germany: " + formatter.withLocale(Locale.GERMANY).fo
System.out.println("Japan:   " + formatter.withLocale(Locale.JAPAN).forr
```

The code outputs the following:

```
US:      Jan 2024
Germany: Jan. 2024
Japan:   2024年1月
```

There is no JDK Enhancement Proposal for this change. You can find it in the JDK 19 Release Notes.

New System Properties for System.out and System.err

Since version 18, Java automatically uses the character encoding of the console or terminal for printing to *System.out* and *System.err*. On Linux, this is usually UTF-8 and on Windows, code page 437.

Save the following program in the file *Test.java*:

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Á é ö ß € ¼");
  }
}
```

If you start this on Linux, all characters will probably be displayed correctly:

```
$ java Test.java
Á é ö ß € ¼
```

However, if you run the program on Windows, you will most likely see the following output (a question mark instead of the Á and € characters):

```
C:\...>java Test.java
? é ö ß ? ¼
```

This is because Windows has the character encoding "code page 437" activated by default, which does not contain the corresponding characters.

You can switch the Windows console to UTF-8 as follows:

```
C:\...>chcp 65001
Active code page: 65001
```

When you start the program again, you will now see all characters correctly.

If the automatic character set recognition does not work, you can set it to UTF-8, for example, using the following VM options from Java 19 onwards:

-Dstdout.encoding=utf8 -Dstderr.encoding=utf8

If you don't want to do this every time you start the program, you can also set these settings globally by defining the following environment variable (yes, it begins with an underscore):

_JAVA_OPTIONS="-Dstdout.encoding=utf8 -Dstderr.encoding=utf8"

There is no JDK Enhancement Proposal for this change. You can find it in the JDK 19 release notes.

Complete List of All Changes in Java 19

In addition to the JDK Enhancement Proposals (JEPs) and class library changes presented in this article, there are numerous smaller changes that are beyond the scope of this article. You can find a complete list in the JDK 19 Release Notes.

Summary

In Java 19, the long-awaited virtual threads developed in Project Loom have finally found their way into the JDK (albeit in preview stage for now). I hope you are as excited as I am and can't wait to use virtual threads in your projects!

Structured Concurrency (still in the incubator stage) will build on this to greatly simplify the management of tasks that are split into parallel subtasks.

The pattern matching capabilities in *instanceof* and *switch*, which have been gradually enhanced in recent JDK versions, have been extended to include record patterns.

The preview and incubator features "Pattern Matching for switch", "Foreign Function & Memory API", and "Vector API" were sent to the next preview and incubator rounds.

Various other changes round off the release as usual. You can download Java 19 here.

You don't want to miss any HappyCoders.eu article and always be informed about new Java features? Then click here to sign up for the free HappyCoders newsletter.

Previous:
New features in Java 18

Next:
New features in Java 20

About the Author

I'm a freelance software developer with more than two decades of experience in scalable Java enterprise applications. My focus is on optimizing complex algorithms and on advanced topics such as concurrency, the Java memory model, and garbage collection. Here on HappyCoders.eu, I want to help you become a better Java programmer. Read more about me here.

Leave a Reply

Your email address will not be published. Required fields are marked *

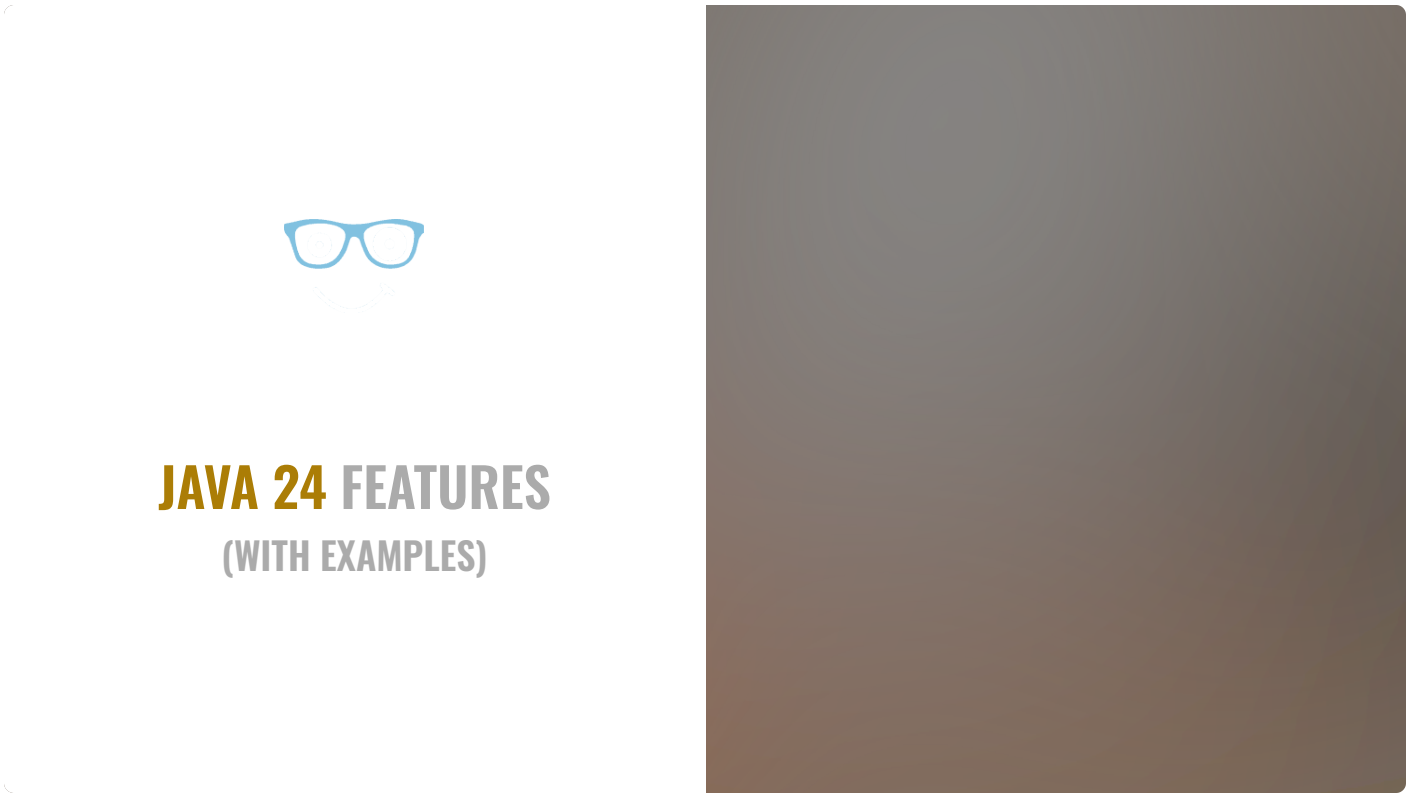
Comment *

Name *

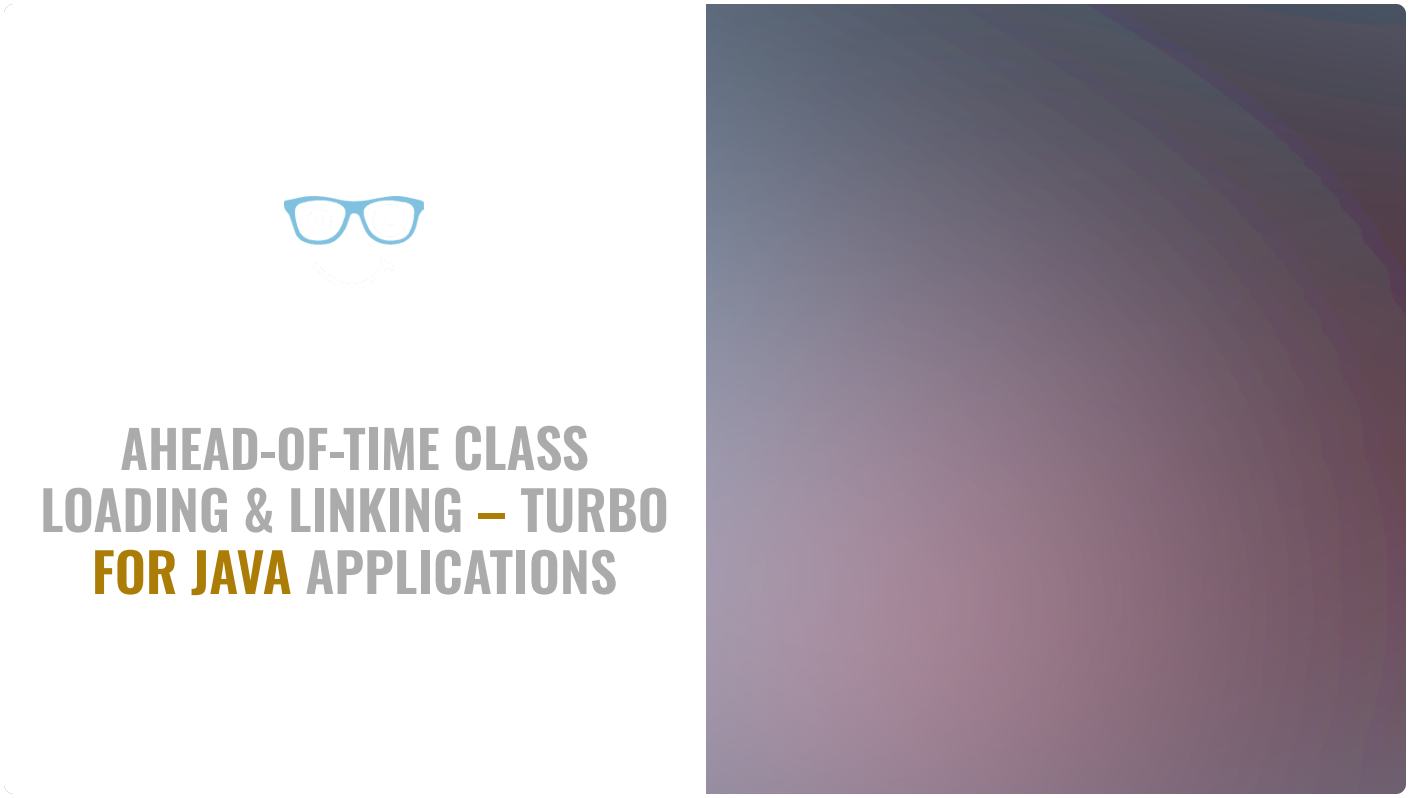
Email *

Post Comment

You might also like the following articles



Sven Woltmann December 4, 2024



Sven Woltmann December 3, 2024

Sven Woltmann December 3, 2024

Sven Woltmann December 2, 2024

Advanced Java topics, algorithms and data structures.

JOIN OUR FREE NEWSLETTER

Boost your skills: Become a better Java programmer.

CLICK HERE TO SUBSCRIBE!

Blog

Java
Algorithms and Data Structures
Software Craftsmanship
Book Recommendations

Resources

Java Versions Cheat Sheet
Big O Cheat Sheet
Newsletter
Conference Talks & Publications

About

About Sven Woltmann
HappyCoders Manifesto

Follow us

Contact Legal Notice Privacy Policy

★★★★★

13 Bewertungen auf ProvenExpert.com