SOFTWARE ENGINEERING

What's new in Java 8: Lambdas

A hands-on introduction to Java 8's most exciting new feature

By Madhusudhan Konda. April 14, 2014

Greek lowercase letter Lambda (source: Wikimedia Commons)

Java 8 is here, and, with it, come lambdas. Although long overdue, lambdas are a remarkable new feature that could make us rethink our programming styles and strategies. In particular, they offer exciting new possibilities for functional programming.

The O'Reilly Software Architecture Conference in San Jose 2019, June 10-13

Learn more

While lambdas are the most prominent addition to Java 8, there are many other new features, such as functional interfaces, virtual methods, class and method references, new time and date API, JavaScript support, and so on. In this post, I will focus mostly on lambdas and their associated features, as understanding this feature is a must for any Java programmer on-boarding to Java 8.

All of the code examples mentioned in this post can be found in this github repo.

What are lambdas?

Lambdas are succinctly expressed single method classes that represent behavior. They can either be assigned to a variable or passed around to other methods just like we pass data as arguments.

You'd think we'd need a new function type to represent this sort of expression. Instead, Java designers cleverly used existing interfaces with one single abstract method as the lambda's type.

Before we go into detail, let's look at a few examples.

Example Lambda Expressions

Here are a few examples of lambda expressions:

```
// Concatenating strings
(String s1, String s2) -> s1+s2;

// Squaring up two integers
(i1, i2) -> i1*i2;

// Summing up the trades quantity
(Trade t1, Trade t2) -> {
  t1.setQuantity(t1.getQuantity() + t2.getQuantity());
  return t1;
};

// Expecting no arguments and invoking another method
() -> doSomething();
```

Have a look at them once again until you familiarize yourself with the syntax. It may seem a bit strange at first. We will discuss the syntax in the next section.

You might wonder what the *type* is for these expressions. The type of any lambda is a *functional interface*, which we discuss below.

Lambda Syntax

There is a special syntax to create and represent a lambda expression. Just like a normal Java method, a lambda expression has input arguments, a body, and an optional return value. This is illustrated here below:

```
input arguments -> body
```

Each lambda expression has two parts separated by an arrow token: the left hand side is our method arguments, and the right hand side is what we do with those arguments (that is, applying business logic). The body can either be an expression or block of code, returning a result or void.

So, in the first lambda expression, (String s1, String s2) → s1+s2, the left hand side of the arrow (→) token is our method argument list. The arguments to the method are supplied as two strings. Coming to the right hand side part, the logic that we are applying in the method is what we are expressing here.

So, from our example, all we need to do is—given two strings, concatenate them up! Whatever logic we put in a method, can go here in this block. In our example, it is adding the both input arguments: s1+s2. The right hand side is the body where we can write statements, expressions, code blocks, calling other methods, etc.

Lambda's Type: the Functional Interface

I briefly mentioned earlier that the type of a lambda is a functional interface. As Java is a strongly typed language, it is usually mandatory to declare types; otherwise, the compiler laughs at us. Note, however, that we omitted types when declaring the above lambda expressions. So, what is the type of a lambda expression? Is it a string, an

object, or a new functional type?

Fortunately, there is no new addition to the type system—the designers of Java did not introduce any special type to represent lambda expressions. Instead, they cleverly re-used the existing anonymous method strategy. We're already familiar with anonymous classes, so picking up this new representation should be relatively straight forward.

A functional interface is a special interface with one and only one abstract method. It is exactly the same as our normal interface, but with two additional characteristics:

- It has one and only one abstract method.
- It can be decorated with an optional @FunctionalInterface annotation to be used as a lambda expression. (this is strongly suggested)

Java has numerous single method interfaces. These are all retrofitted, making them functional interfaces. If we want to create our own, all we need to do is to define an interface with one abstract method and stick a @FunctionalInterface annotation on top of it!

For example, the following snippet defines an `IAddable` interface. This is a functional interface whose job is to simply add two identical items of type T.

```
@FunctionalInterface
public interface IAddable<T> {
    // To add two objects
    public T add(T t1, T t2);
}
```

Because this interface has one and only one abstract method and it is annotated with @FunctionalInterface, it can be used as a type for representing lambda functions.

Below are some example representations that use the above IAddable functional interface here:

```
// Our interface implementations using Lambda expressions
// Joining two strings—note the interface is a generic type

IAddable<String> stringAdder = (String s1, String s2) -> s1+s2;

// Squaring the number
IAddable<Integer> square = (i1, i2) -> i1*i2;

// Summing up the trades quantity
IAddable<Trade> tradeAdder = (Trade t1, Trade t2) -> {
  t1.setQuantity(t1.getQuantity() + t2.getQuantity());
  return t1;
};
```

Note that the `IAddable` is a generic type interface, therefore we're using it to add different types, as in the above example.

To summarize, the lambda expression's type is the functional interface that we intended to implement via the lambda expression.

Once we have the implementations ready, we can use them in our class by simply invoking the respective method. See the example below, which shows how the above implementations are being used.

```
// A lambda expression for adding two strings.
IAddable<String> stringAdder = (s1, s2) -> s1+s2;

// this method adds the two strings using the first lambda expression
private void addStrings(String s1, String s2) {
  log("Concatenated Result: " + stringAdder.add(s1, s2));
}
```

Before we move on, let's take a moment to understand what is achieved here. The noticeable thing is that we are now treating the business logic as a function that can be tossed around for execution. We can define numerous variations of the business logic on the fly (kind of!) without having to create and instantiate classes as we used to do before.

Now that we understand a bit about lambda expressions and their types, let's run through a complete example using lambdas. We'll also compare and contrast with the pre-Java 8 version to understand the differences.

Lambda Example

Our requirement is to create business logic for merging two trades issued by the same issuer. This example should solve the requirement by using both pre-Java 8 and Java 8 approaches.

Pre-Java 8 Implementation

Prior to Java 8, we were expected to implement the interface with concrete definitions, as shown in the test class below:

```
public void testPreJava8() {
  IAddable<Trade> tradeMerger = new IAddable<Trade>() {
  @Override
    public Trade add(Trade t1, Trade t2) {
      t1.setQuantity(t1.getQuantity() + t2.getQuantity());
      return t1;
    }
  };
}
```

Here, we created a concrete class implementing the interface and invoked the add method on the instantiated object. We used an anonymous strategy for creating the code shown above.

Although the actual logic of merging the trades is our core logic, we are forced to do some additional tasks such

as implementing the interface with a class identity, overriding the abstract method, creating an instance of it and finally doing something with the instance. This "excess baggage" has always attracted critics and made developers uneasy—it's a lot of boiler plate code and meaningless implementations and instantiations for a piece of business logic.

Once we have the instance of the class, we follow the usual process of invoking the respective methods on the instance, as shown below:

```
IAddable addable = ....;

Trade t1 = new Trade(1, "GOOG", 12000, "NEW");
Trade t2 = new Trade(2, "GOOG", 24000, "NEW");

// using the conventional anonymous class..
Trade mergedTrade = tradeMerger.add(t1,t2);
```

The business logic is intertwined with the technical garbage. The core logic is very much tied to the implementing class. For example, instead of returning a merged trade as in the above case, if I may have to compare and return the big trade, I have to sigh a bit, grab a coffee, sneeze loudly, moan, groan and roll up my sleeves and get ready to rewrite the code logic.

Note also that all our test cases will start failing too once we change the logic!

What if I have to support dozens of such requirements? Well, either I have to hack the current method to create the additional logic path using a control statement (if-else or switch) or create a new class for each piece of logic. Tightly coupling your business logic to its implementing class is asking for trouble—especially if we have fickle-minded business analysts and project managers. Surely there must be a better way to make it work.

Java 8 Implementation

Creating multiple behaviors using anonymous classes is not impossible, but it's not ideal. We can simplify this problem by using lambda functions to represent various behaviors. So, for example, we can write a lambda expression to sum up the trade's quantities, another for returning a large trade, another expression for encrypting trade data, etc. All we do is write a lambda expression for each of the requirements and send it over to the class that expects the lambda.

For example, take a look at this list of lambda expressions for our requirements:

```
// Summing up the trades quantity
IAddable<Trade> aggregatedQty = (t1, t2) -> {
  t1.setQuantity(t1.getQuantity() + t2.getQuantity());
  return t1;
};

// Return a large trade
IAddable<Trade> largeTrade = (t1, t2) -> {
 if (t1.getQuantity() > 1000000)
   return t1;
 else
   return t2;
};

// Encrypting the trades (Lambda uses an existing method)
IAddable<Trade> encryptTrade = (t1, t2) -> encrypt(t1,t2);
```

As you can see, we have declared lambdas for each our respective functionalities. The method can now be modeled to expect an expression, as shown below:

```
//A generic method with an expected lambda
public void applyBehaviour(IAddable addable, Trade t1, Trade t2){
  addable.add(t1, t2);
}
```

Note that the method is generic enough that it can apply functionality to any two trades with the given behavior using the given lambda expression (the IAddable interface).

Now, the client has the control of creating the behaviors and pass it on to the remote server for application of them. This way, the client cares about *what* to do, while the server cares about*how* to do it. As long as the interface is designed to accept the lambda expression, the client can create a number of those expressions according to its requirements and invoke the method.

Before we sum up, let's take an existing Runnable interface, which comes with lambda support, and see how it can be used.

Runnable Functional interface

The most popular Runnable interface has one method, run, which takes no arguments and returns nothing. Perhaps the reasoning behind this is that a piece of logic is to be run in a separate thread to speed up the process.

The new definition of Runnable interface, followed by an example implementation using an anonymous class, is given below:

```
// The functional interface
@FunctionalInterface
public interface Runnable {
  public void run();
}

// example implementation
new Thread(new Runnable() {
  @Override
  public void run() {
    sendAnEmail();
  }
}).start();
```

As you can see the above way of creating and using the anonymous class is very verbose—and unsightly too. Apart from sendAnEmail() in the run method, everything else is redundant boilerplate code.

The same Runnable can now be re-written to use a lambda expression, as shown below:

```
// The constructor now takes in a lambda
new Thread( () -> sendAnEmail() ).start();
```

The lambda expression () → sendAnEmail() highlighted above is passed to the constructor of the thread. Note that this expression is effectively a piece of code (an instance of Runnable) that carries certain behavior (make sure to always send an email in a new thread).

Looking at the expression, we can deduce the type of the lambda—in this case it is Runnable, as we all know that the thread constructor accepts a Runnable. If you've noticed the redefined interface definition, Runnable is now a functional interface and hence tagged with the @FunctionalInterface annotation. Now the lambda expression can be assigned to a class's variable as Runnable r = () → sendAnEmail() just like how we declare and assign variables.

This is the power of lambdas—they allow us to pass a behavior to a method, assigned to a variable (free floating) from another method, just like we do when passing arguments that carry data.

Let's suppose we have a server side class called AsyncManager whose sole purpose is to execute requests on different threads. It has a single method, runAsync, which accepts a Runnable, as shown below:

```
public class AsyncManager{
  public void runAsync(Runnable r) {
    new Thread(r);
  }
}
```

The client now has the ability to create a plethora of lambda expressions based on his requirements. For example, see the various lambda expressions that can be passed on to the server-side class:

```
manager.runAsync(() -> System.out.println("Running in Async mode"));
manager.runAsync(() -> sendAnEmail());
manager.runAsync(() -> {
  persistToDatabase();
  goToMoon();
  returnFromMars();
  sendAnEmail();
});
```

Summary

In this post, we learned about the biggest change Java has ever seen. Lambdas will certainly steer the direction of Java and make it more appealing to varied sections of the programmer community.

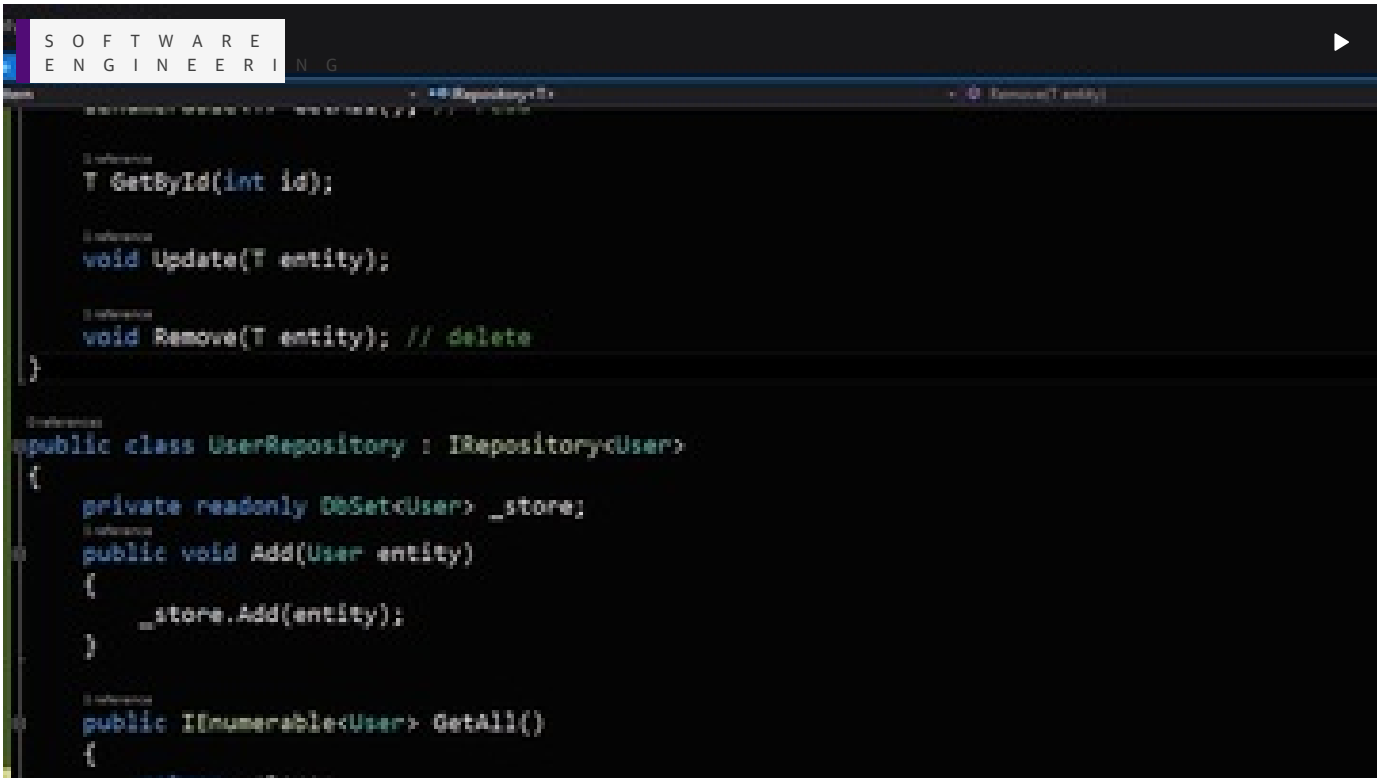
See part two of this series, in which we discuss Java 8's functional interfaces.

Article image: Greek lowercase letter Lambda (source: Wikimedia Commons).

Madhusudhan Konda

Madhusudhan Konda is an experienced Java consultant working in London, primarily with investment banks and financial organizations. Having worked in enterprise and core Java for the last 12 years, his interests lie in distributed, multi-threaded, n-tier scalable, and extensible architectures. He is experienced in designing and developing high-frequency and low-latency application architectures. He enjoys writing technical papers and is interested in mentoring.

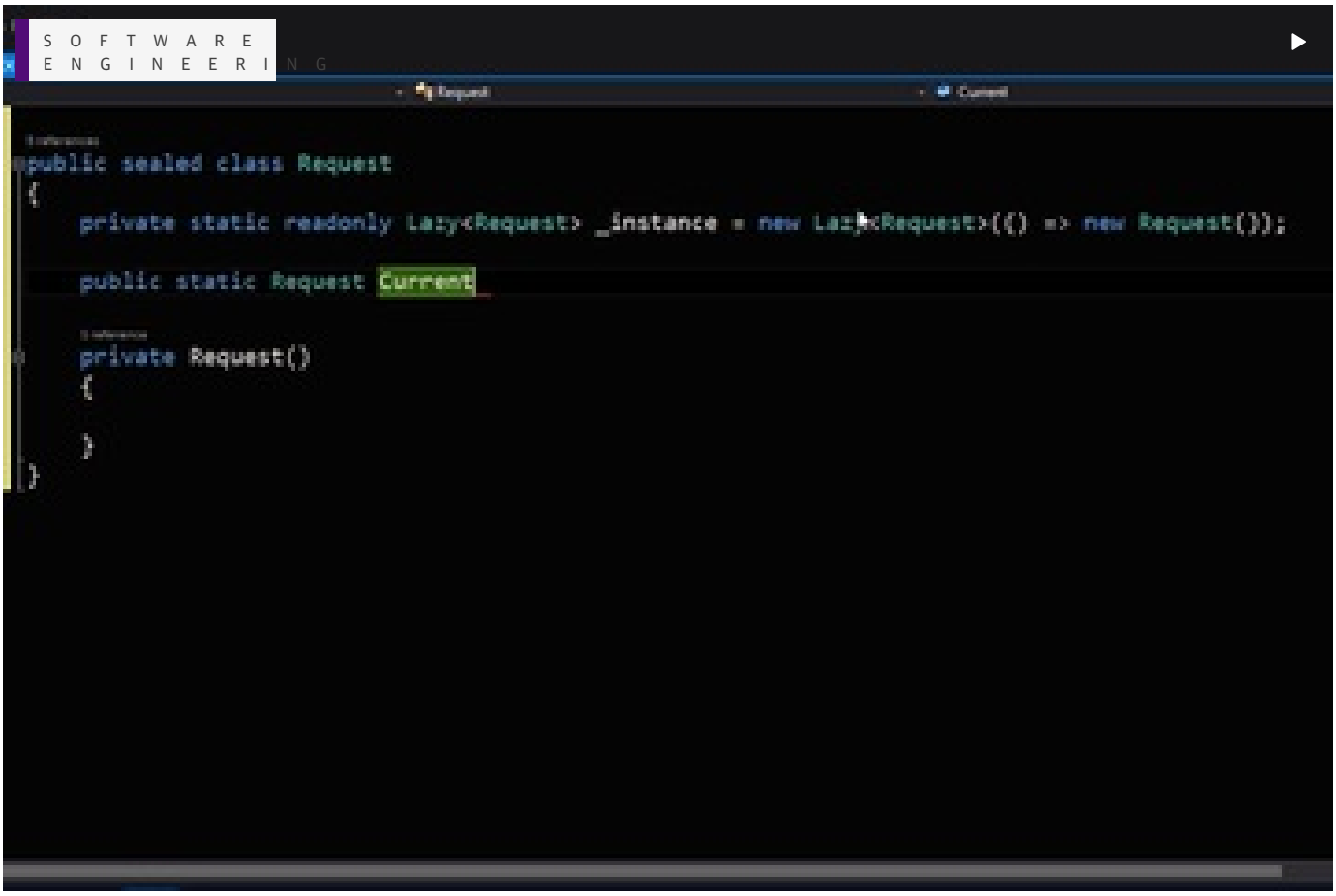
more



How can I pass parameters to a command in C#?

By Jeremy McPeak

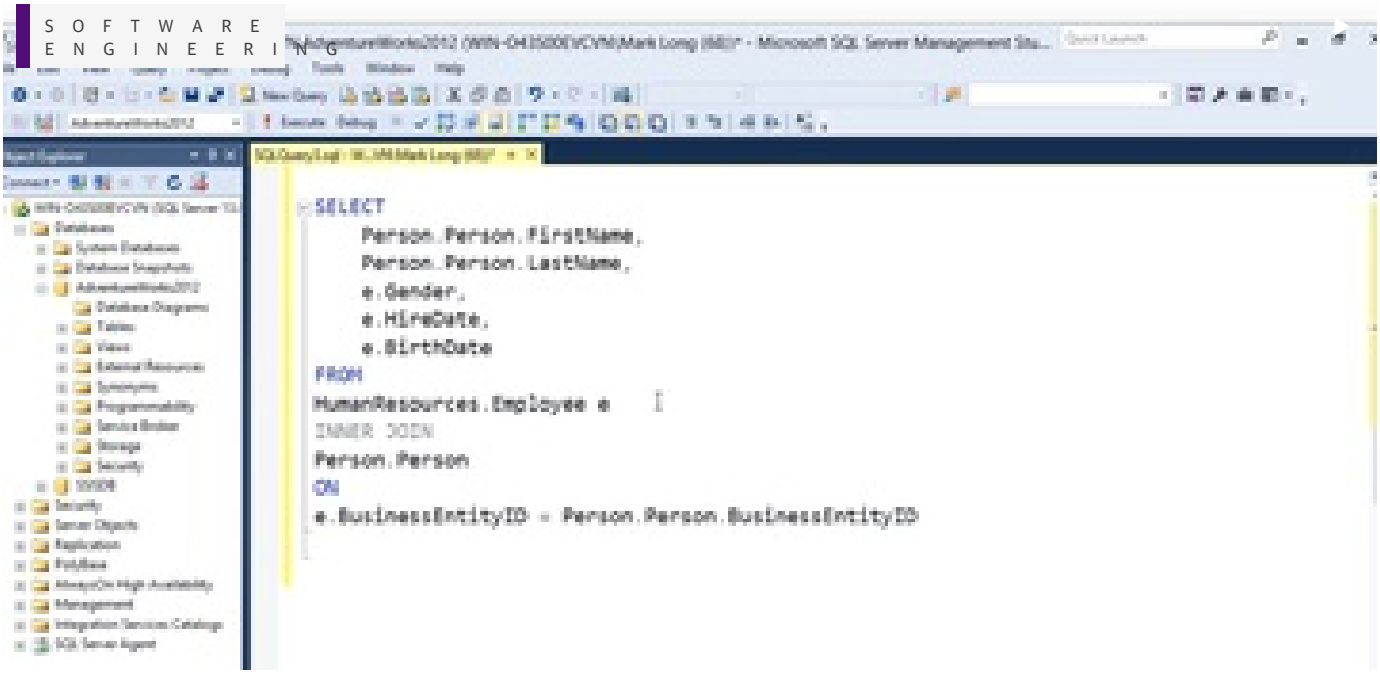
Learn how to pass data to a command without violating the command pattern in C#.

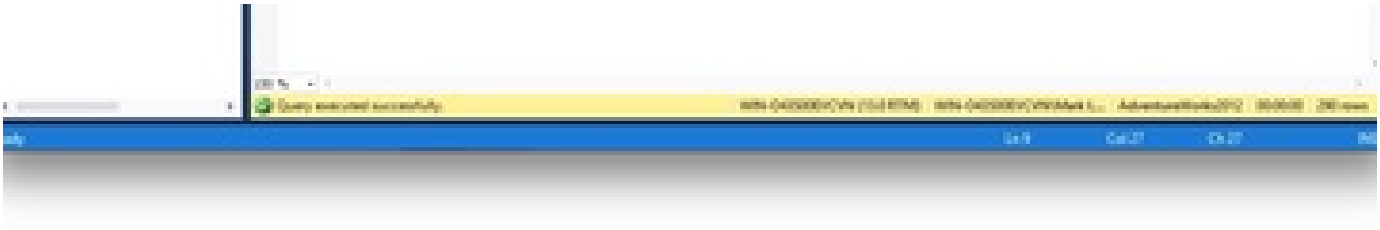


How do I use the singleton pattern in C#?

By Jeremy McPeak

Learn how to create thread-safe instances with the singleton pattern in C#.

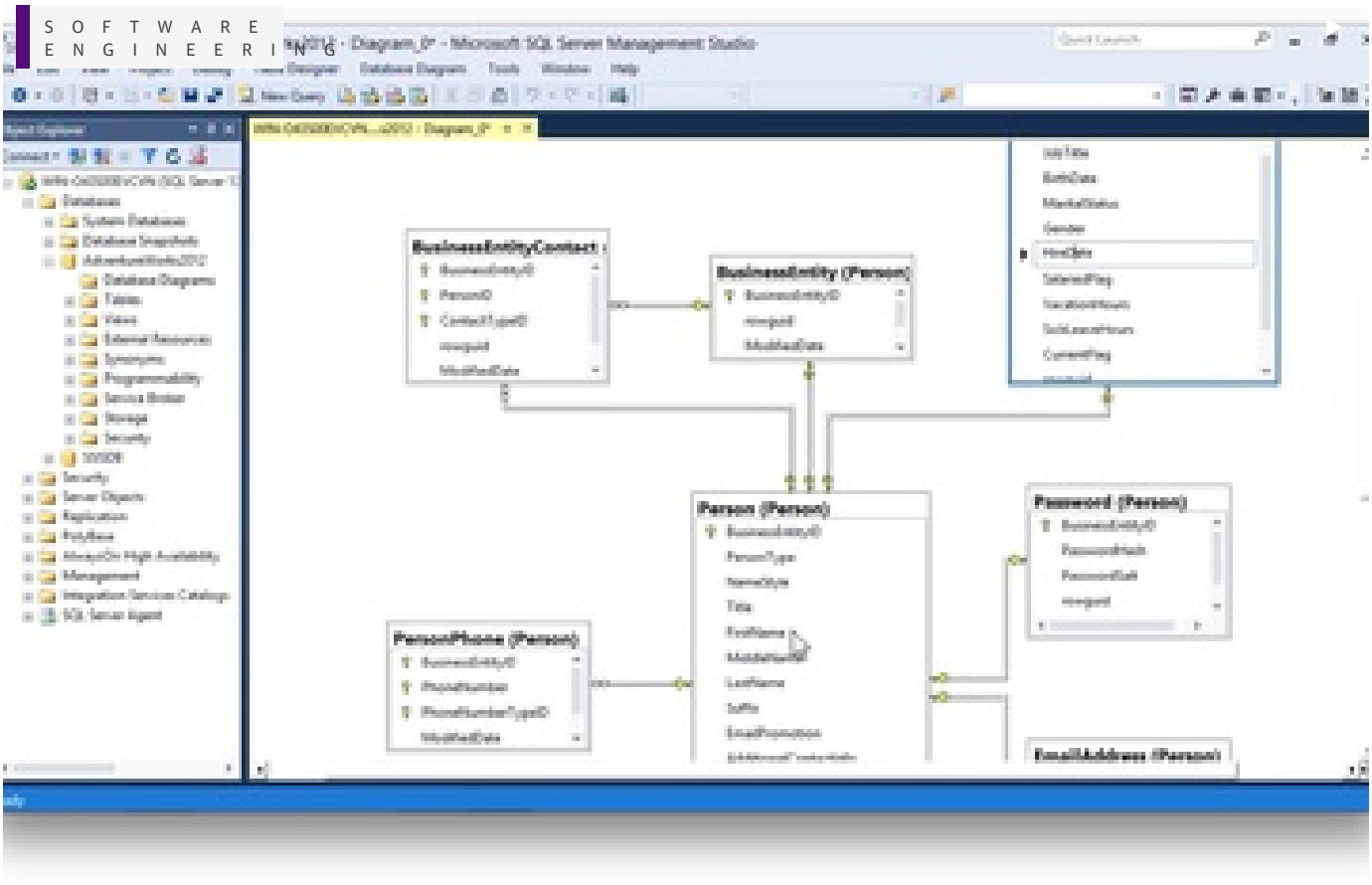




How should I format Transact-SQL queries?

By Mark Long

Learn the formatting possibilities for Transact-SQL queries and develop your own code structure.



How do I locate data in my SQL Server tables using SQL Server Management Studio diagrams?

By Mark Long

Locate data quickly and easily with the SQL Server Management Studio diagram tool.