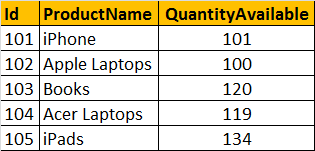
**In**[**Part 5**](http://csharp-video-tutorials.blogspot.com/2012/10/sql-injection-tutorial-part-5.html)**, we have learn about, what can cause sql injection. In this session we will learn about**   
1. Preventing sql injection using parametrized queries and stored procedures.  
2. How to execute stored procedures and parameterized queries using ADO.NET command object  
  
**The table, this demo is based on, is shown below.**   
   
  
   
  
**The following ADO.NET code is from Part 5**. This is the code, that let's sql injection happen.   
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
using (SqlConnection con = new SqlConnection(CS))  
{  
    string Command = "Select \* from tblProductInventory where ProductName like '" + TextBox1.Text + "%'";  
    SqlCommand cmd = new SqlCommand(Command, con);  
    con.Open();  
    GridView1.DataSource = cmd.ExecuteReader();  
    GridView1.DataBind();  
}  
  
**The above code can be easily re-written using parameterized queries** to prevent sql injection attack. The re-written code is shown below. Notice, that the query now uses parameter - **@ProductName**. The value for this parameter is then provided using the **AddWithValue**() method. The parameter is associated with the command object using **Prameters** collection property of the **command** object.  
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
using (SqlConnection con = new SqlConnection(CS))  
{  
    // Parameterized query. @ProductName is the parameter  
    string Command = "Select \* from tblProductInventory where ProductName like @ProductName" ;  
    SqlCommand cmd = new SqlCommand(Command, con);  
    // Provide the value for the parameter  
    cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");  
    con.Open();  
    GridView1.DataSource = cmd.ExecuteReader();  
    GridView1.DataBind();  
}   
  
   
  
**Sql injection can also be prevented using stored procedures.** So, first let's write a stored procedure, that returns the list of products. This stored procedure takes an input parameter **@ProductName.**  
Create Procedure spGetProductsByName  
@ProductName nvarchar(50)  
as  
Begin  
 Select \* from tblProductInventory   
 where ProductName like @ProductName + '%'  
End  
  
**To test this procedure execute the follwing command in sql server management studio.**  
Execute spGetProductsByName 'ip'  
  
**Now, let's re-write the code, to use stored procedure spGetProductsByName.**  
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
using (SqlConnection con = new SqlConnection(CS))  
{  
    // The command, that we want to execute is a stored procedure,  
    // so specify the name of the procedure as cmdText  
    SqlCommand cmd = new SqlCommand("spGetProductsByName", con);  
    // Specify that the T-SQL command is a stored procedure  
    cmd.CommandType = System.Data.CommandType.StoredProcedure;  
    // Associate the parameter and it's value with the command object  
    cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");  
    con.Open();  
    GridView1.DataSource = cmd.ExecuteReader();  
    GridView1.DataBind();  
}  
  
**If you type the following input into the TextBox**, the entire content of the TextBox is now treated as a value for the parameter - @ProductName not as a seperate sql statement.  
**i'; Delete from tblProductInventory --**  
  
**So the conclusion is that, always used parameterized queries or stored procedures, to avoid sql injection attacks.**