GWT - QUICK GUIDE

What is GWT?

- Google Web Toolkit (GWT) is a development toolkit to create RICH Internet Application(RIA).
- GWT provides developers option to write client side application in JAVA.
- GWT compiles the code written in JAVA to JavaScript code.
- Application written in GWT is cross-browser compliant. GWT automatically generates javascript code suitable for each browser.
- GWT is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache License version 2.0.

Overall, GWT is a framework to build large scale and high performance web appliation while keeping them as easy-to-maintain.

Why to use GWT?

- Being Java based, you can use JAVA IDEs like Eclipse to develop GWT applcation. Developers can use code auto-complete/refactoring/navigation/project management and all features of IDEs.
- GWT provides full debugging capability. Developers can debug the client side application just as an Java Application.
- GWT provides easy integration with Junit and Maven.
- Again being Java based, GWT has a low learning curve for Java Developers.
- GWT generates optimized javascript code, produces browser's specific javascript code by self.
- GWT provides Widgets library provides most of tasks required in an application.
- GWT is extensible and custom widget can be created to cater to application needs.

On top of everything, GWT applications can run on all major browsers and smart phones including Android and iOS based phones/tablets.

Disadvantages of GWT

Though GWT comes with lots of plus points but same time we should consider the following points:

- **Not indexable** : Web pages generated by GWT would not be indexed by search engines because these applications are generated dynamically.
- **Not degradable**: If your application user disables Javascript then user will just see the basic page and nothing more.
- **Not designer's friendly**: GWT is not suitable for web designers who prefer using plain HTML with placeholders for inserting dynamic content at later point in time.

The GWT Components

The GWT framework can be divided into following three major parts:

- **GWT Java to JavaScript compiler** : This is the most important part of GWT which makes it a powerful tool for building RIAs. The GWT compiler is used to translate all the application code written in Java into JavaScript.
- **JRE Emulation library** : Google Web Toolkit includes a library that emulates a subset of the Java runtime library. The list includes *java.lang, java.lang.annotation, java.math, java.io, java.sql, java.util* and *java.util.logging*
- **GWT UI building library** : This part of GWT consists of many subparts which includes the actual UI components, RPC support, History management, and much more.

GWT also provides a **GWT Hosted Web Browser** which lets you run and execute your GWT applications in hosted mode, where your code runs as Java in the Java Virtual Machine without compiling to JavaScript.

GWT - ENVIRONMENT SETUP

This tutorial will guide you on how to prepare a development environment to start your work with GWT Framework. This tutorial will also teach you how to setup JDK, Tomcat and Eclipse on your machine before you setup GWT Framework:

System Requirement

GWT requires JDK 1.6 or higher so the very first requirement is to have JDK installed in your machine.

JDK	1.6 or above.
Memory	no minimum requirement.
Disk Space	no minimum requirement.
Operating System	no minimum requirement.

Follow the given steps to setup your environment to start with GWT application development.

Step 1 - Verify Java installation on your machine

Now open console and execute the following **java** command.

OS	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version
Mac	Open Terminal	machine:~ joseph$ java -version

Let's verify the output for all the operating systems:

OS	Generated Output
Windows	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Mac	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing)

Step 2 - Setup Java Development Kit (JDK):

If you do not have Java installed then you can install the Java Software Development Kit (SDK) from Oracle's Java site: Java SE Downloads. You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Finally set PATH and JAVA_HOME environment variables to refer to the directory that contains java and javac, typically java_install_dir/bin and java_install_dir respectively.

Set the **JAVA_HOME** environment variable to point to the base directory location where Java is installed on your machine. For example

OS	Output
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA_HOME=/usr/local/java-current
Mac	export JAVA_HOME=/Library/Java/Home

Append Java compiler location to System Path.

OS	Output
Windows	Append the string ;%JAVA_HOME%\bin to the end of the system variable, Path.
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac	not required

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, compile and run a simple program to confirm that the IDE knows where you installed Java, otherwise do proper setup as given document of the IDE.

Step 3 - Setup Eclipse IDE

All the examples in this tutorial have been written using Eclipse IDE. So I would suggest you should have latest version of Eclipse installed on your machine based on your operating system.

To install Eclipse IDE, download the latest Eclipse binaries from http://www.eclipse.org/downloads/. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\eclipse on windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

Eclipse can be started by executing the following commands on windows machine, or you can simply double click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine:

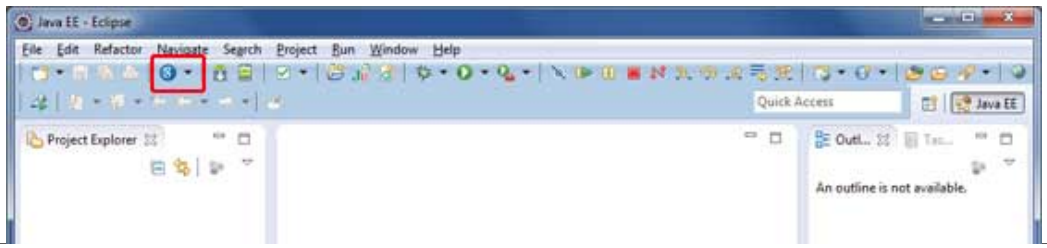
```
$/usr/local/eclipse/eclipse
```

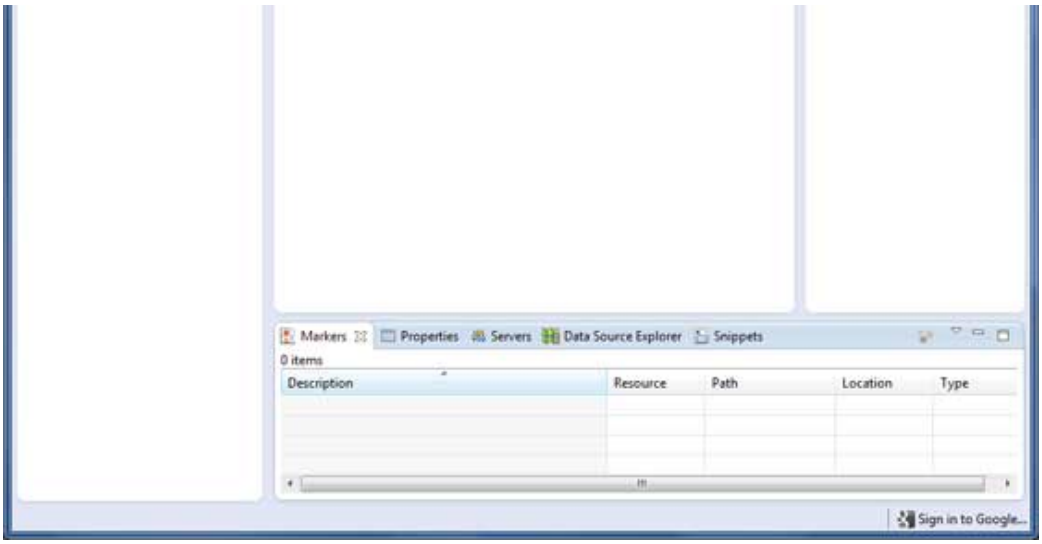
After a successful startup, if everything is fine then it should display following result:

Step 4: Install GWT SDK & Plugin for Eclipse

Follow the instructions given at the link Plugin for Eclipse (incl. SDKs) to install GWT SDK & Plugin for Eclipse version installed on your machine.

After a successful setup for the GWT plugin, if everything is fine then it should display following screen with google icon marked with red rectangle:





Step 5: Setup Apache Tomcat:

You can download the latest version of Tomcat from http://tomcat.apache.org/. Once you downloaded the installation, unpack the binary distribution into a convenient location. For example in C:\apache-tomcat-6.0.33 on windows, or /usr/local/apache-tomcat-6.0.33 on Linux/Unix and set CATALINA_HOME environment variable pointing to the installation locations.

Tomcat can be started by executing the following commands on windows machine, or you can simply double click on startup.bat

```
%CATALINA_HOME%\bin\startup.bat

or

C:\apache-tomcat-6.0.33\bin\startup.bat
```

Tomcat can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine:

```
$CATALINA_HOME/bin/startup.sh

or

/usr/local/apache-tomcat-6.0.33/bin/startup.sh
```

After a successful startup, the default web applications included with Tomcat will be available by visiting **http://localhost:8080/**. If everything is fine then it should display following result:

Further information about configuring and running Tomcat can be found in the documentation included here, as well as on the Tomcat web site: http://tomcat.apache.org

Tomcat can be stopped by executing the following commands on windows machine:

```
%CATALINA_HOME%\bin\shutdown

or

C:\apache-tomcat-5.5.29\bin\shutdown
```

Tomcat can be stopped by executing the following commands on Unix (Solaris, Linux, etc.) machine:

```
$CATALINA_HOME/bin/shutdown.sh

or

/usr/local/apache-tomcat-5.5.29/bin/shutdown.sh
```

Before we start with creating actual *HelloWorld* application using GWT, let us see what are the actual parts of a GWT application. A GWT application consists of following four important parts out of which last part is optional but first three parts are mandatory:

- **Module descriptors**
- **Public resources**
- **Client-side code**
- **Server-side code**

Sample locations of different parts of a typical gwt application **HelloWord** will be as shown below:

Name	Location
Project root	HelloWorld/
Module descriptor	src/com/tutorialspoint/HelloWorld.gwt.xml
Public resources	src/com/tutorialspoint/war/
Client-side code	src/com/tutorialspoint/client/
Server-side code	src/com/tutorialspoint/server/

Module Descriptors

A module descriptor is the configuration file in the form of XML which is used to configure a GWT application. A module descriptor file extension is ***.gwt.xml**, where * is the name of the application and this file should reside in the project's root. Following will be a default module descriptor **HelloWorld.gwt.xml** for a **HelloWorld** application:

```
<?xml version="1.0" encoding="utf-8"?>
<module rename-to='helloworld'>
   <!-- inherit the core web toolkit stuff.                      -->
   <inherits name='com.google.gwt.user.user'/>

   <!-- inherit the default gwt style sheet.                     -->
   <inherits name='com.google.gwt.user.theme.clean.Clean'/>

   <!-- specify the app entry point class.                       -->
   <entry-point class='com.tutorialspoint.client.HelloWorld'/>

   <!-- specify the paths for translatable code                  -->
   <source path='...'/>
   <source path='...'/>

   <!-- specify the paths for static files like html, css etc.   -->
   <public path='...'/>
   <public path='...'/>

   <!-- specify the paths for external javascript files          -->
   <script src="js-url" />
   <script src="js-url" />

   <!-- specify the paths for external style sheet files          -->
   <stylesheet  src="css-url" />
   <stylesheet  src="css-url" />
</module>
```

Following is the brief detail about different parts used in module descriptor.

S.N.	Nodes & Description
1	**<module rename-to="helloworld">** This provides name of the application.

2	**<inherits name="logical-module-name" />**This adds other gwt module in application just like import does in java applications. Any number of modules can be inherited in this manner.
3	**<entry-point />**This specifies the name of class which will start loading the GWT Application. Any number of entry-point classes can be added and they are called sequentially in the order in which they appear in the module file. So when the onModuleLoad() of your first entry point finishes, the next entry point is called immediately.
4	**<source path="path" />**This specifies the names of source folders which GWT compiler will search for source compilation.
5	**<public path="path" />**The public path is the place in your project where static resources referenced by your GWT module, such as CSS or images, are stored. The default public path is the public subdirectory underneath where the Module XML File is stored.
6	**<script src="js-url" />**Automatically injects the external JavaScript file located at the location specified by src.
7	**<stylesheet src="css-url" />**Automatically injects the external CSS file located at the location specified by src.

Public resources

These are all files referenced by your GWT module, such as Host HTML page, CSS or images. The location of these resources can be configured using <public path="path" /> element in module configuration file. By default, it is the public subdirectory underneath where the Module XML File is stored.

When you compile your application into JavaScript, all the files that can be found on your public path are copied to the module's output directory.

The most important public resource is host page which is used to invoke actual GWT application. A typical HTML host page for an application might not include any visible HTML body content at all but it is always expected to include GWT application via a <script.../> tag as follows:

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>Hello World</h1>
<p>Welcome to first GWT application</p>

</body>
</html>
```

Following is the sample style sheet which we have included in our host page:

```
body {
   text-align: center;
   font-family: verdana, sans-serif;
}
h1 {
  font-size: 2em;
  font-weight: bold;
  color: #777777;
  margin: 40px 0px 70px;
  text-align: center;
}
```

Client-side code

This is the actual Java code written implementing the business logic of the application and that the

GWT compiler translates into JavaScript, which will eventually run inside the browser. The location of these resources can be configured using <source path="path" /> element in module configuration file.

For example **Entry Point** code will be used as client side code and its location will be specified using <source path="path" />. A module **entry-point** is any class that is assignable to **EntryPoint** and that can be constructed without parameters. When a module is loaded, every entry point class is instantiated and its **EntryPoint.onModuleLoad()** method gets called. A sample HelloWorld Entry Point class will be as follows:

```
public class HelloWorld implements EntryPoint {
   public void onModuleLoad() {
      Window.alert("Hello, World!");
   }
}
```

Server-side code

This is the server side part of your application and its very much optional. If you are not doing any backend processing with-in your application then you do not need this part, but if there is some processing required at backend and your client-side application interact with the server then you will have to develop these components.

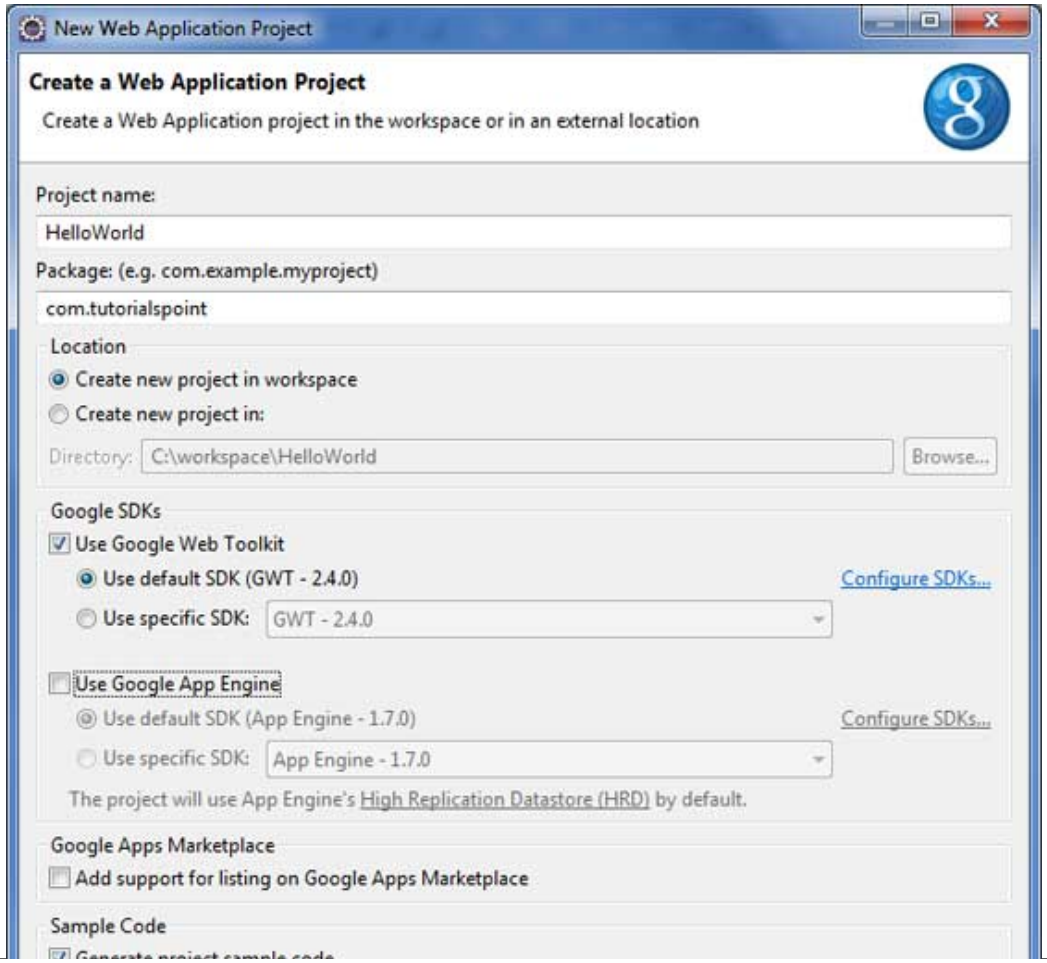
Next chapter will make use of all the above mentioned concepts to create HelloWorld application using Eclipse IDE.

GWT - CREATE APPLICATION

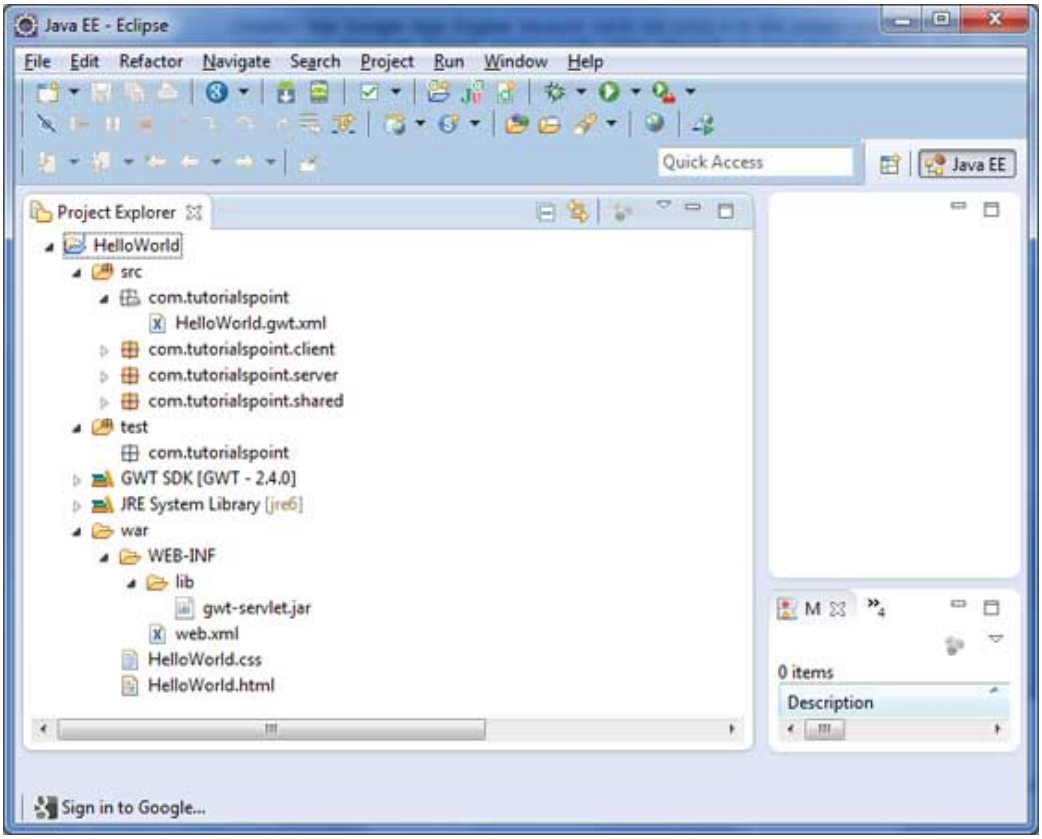
As power of GWT lies in **Write in Java, Run in JavaScript**, we'll be using Java IDE Eclipse to demonstrate our examples. Let's start with a simple *HelloWorld* application:

Step 1 - Create Project

The first step is to create a simple Web Application Project using Eclipse IDE. Launch project wizard using the option **Google Icon > New Web Application Project...**. Now name your project as *HelloWorld* using the wizard window as follows:



Unselect **Use Google App Engine** because we're not using it in this project and leave other default values (keep **Generate Sample project code** option checked) as such and click Finish Button. Once your project is created successfully, you will have following content in your Project Explorer:



Here is brief description of all important folders:

Folder	Location
src	• Source code (java classes) files. • Client folder containing the client-side specific java classes responsible for client UI display. • Server folder containing the server-side java classes responsible for server side processing. • Shared folder containing the java model class to transfer data from server to client and vice versa. • HelloWorld.gwt.xml, a module descriptor file required for GWT compiler to compile the HelloWorld project.
test	• Test code (java classes) source files. • Client folder containing the java classes responsible to test gwt client side code.
war	• This is the most important part, it represents the actual deployable web

application.

- WEB-INF containing compiled classes, gwt libraries, servlet libraries.
- HelloWorld.css, project style sheet.
- HelloWorld.html, hots HTML which will invoke GWT UI Application.

Step 2 - Modify Module Descriptor: HelloWorld.gwt.xml

GWT plugin will create a default module descriptor file *src/com.tutorialspoint/HelloWorld.gwt.xml* which is given below. For this example we are not modifying it, but you can modify it based on your requirement.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.  You can change       -->
  <!-- the theme of your GWT application by uncommenting          -->
  <!-- any one of the following lines.                            -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>
  <!-- <inherits name='com.google.gwt.user.theme.chrome.Chrome'/> -->
  <!-- <inherits name='com.google.gwt.user.theme.dark.Dark'/>     -->

  <!-- Other module inherits                                      -->

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Step 3 - Modify Style Sheet: HelloWorld.css

GWT plugin will create a default Style Sheet file *war/HelloWorld.css*. Let us modify this file to keep our example at simplest level of understaning:

```
body {
   text-align: center;
   font-family: verdana, sans-serif;
}
h1 {
  font-size: 2em;
  font-weight: bold;
  color: #777777;
  margin: 40px 0px 70px;
  text-align: center;
}
```

Step 4 - Modify Host File: HelloWorld.html

GWT plugin will create a default HTML host file *war/HelloWorld.html*. Let us modify this file to keep our example at simplest level of understaning:

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>Hello World</h1>
<p>Welcome to first GWT application</p>
```

```
</body>
</html>
```

You can create more static files like HTML, CSS or images in the same source directory or you can create further sub-directories and move files in those sub-directories and configure those sub-directories in module descriptor of the application.

Step 5 - Modify Entry Point: HelloWorld.java

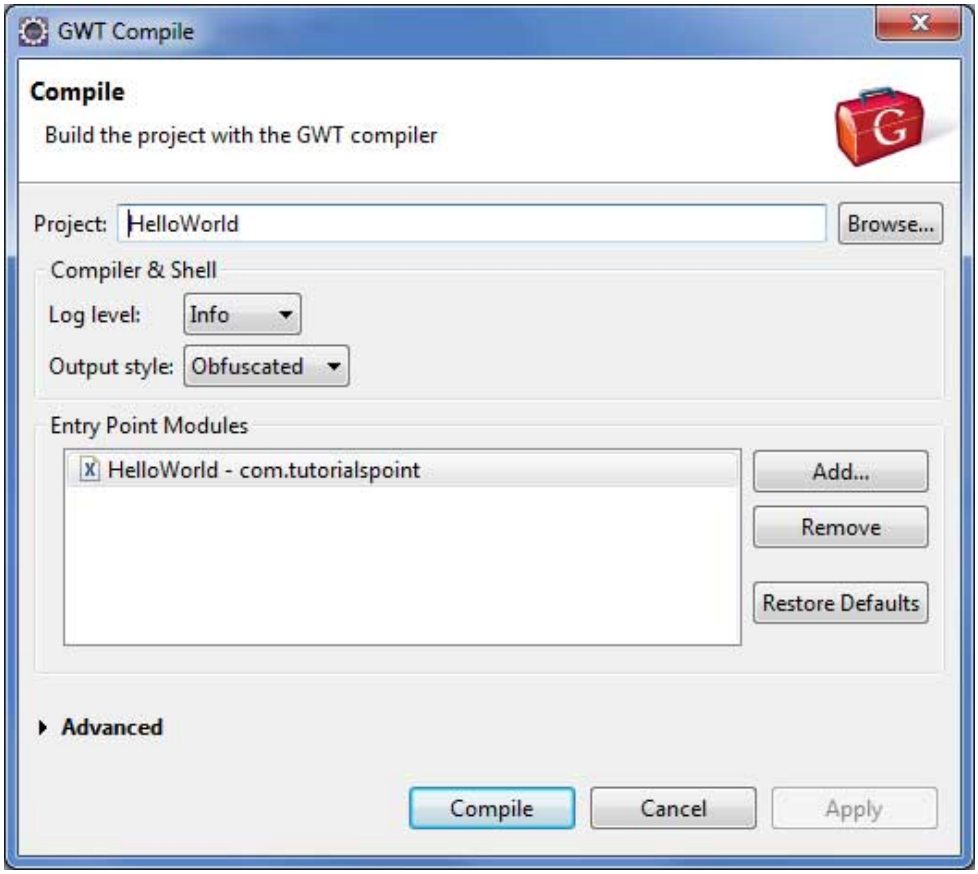
GWT plugin will create a default Java file *src/com.tutorialspoint/HelloWorld.java*, which keeps an entry point for the application. Let us modify this file to display "Hello,World!":

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.Window;

public class HelloWorld implements EntryPoint {
   public void onModuleLoad() {
    Window.alert("Hello, World!");
   }
}
```

You can create more Java files in the same source directory to define either entry points or to define helper routines.

Step 6 - Compile Application

Once you are ready with all the changes done, its time to compile the project. Use the option **Google Icon > GWT Compile Project...** to launch GWT Compile dialogue box as shown below:



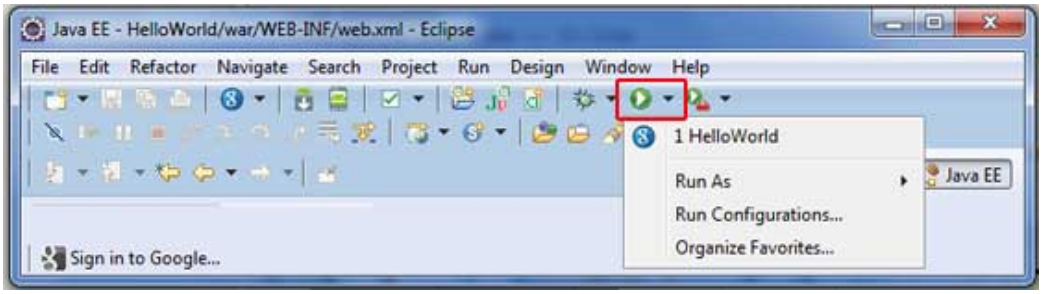
Keep default values intact and click Compile button. If everything goes fine, you will see following output in Eclipse console

```
Compiling module com.tutorialspoint.HelloWorld
   Compiling 6 permutations
      Compiling permutation 0...
      Compiling permutation 1...
      Compiling permutation 2...
```

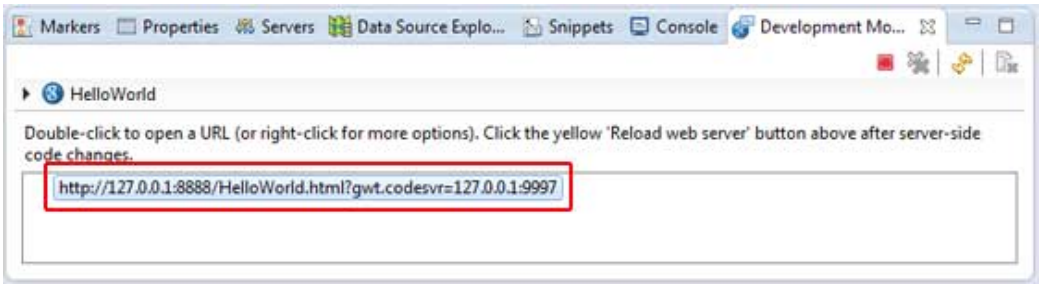
```
      Compiling permutation 3...
      Compiling permutation 4...
      Compiling permutation 5...
   Compile of permutations succeeded
Linking into C:\workspace\HelloWorld\war\helloworld
   Link succeeded
   Compilation succeeded -- 33.029s
```

Step 6 - Run Application

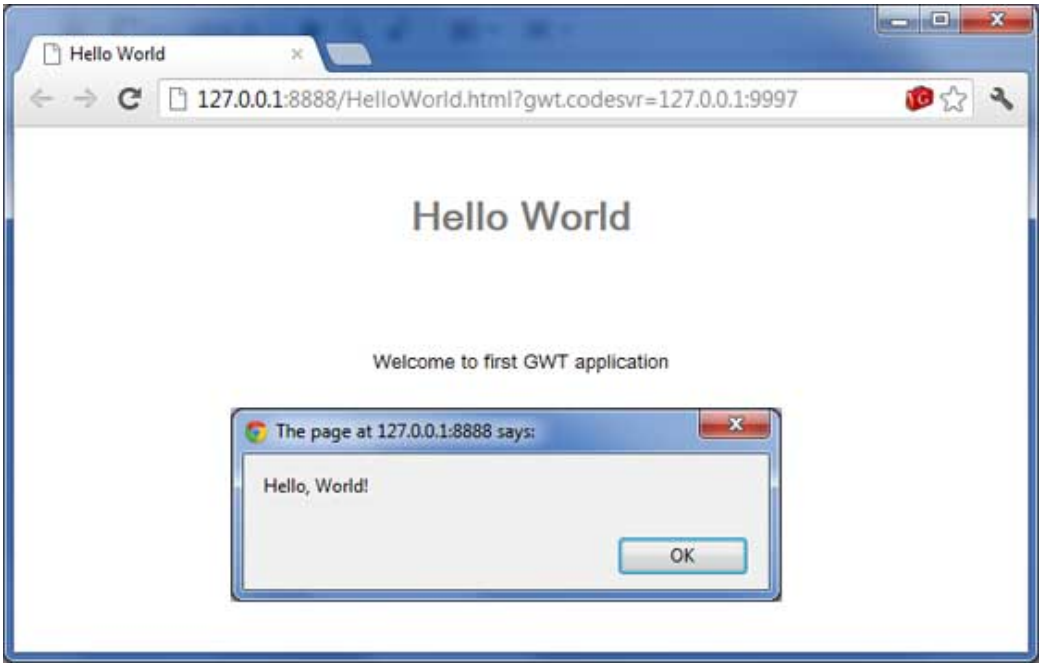
Now click on
Run application menu and select **HelloWorld** application to run the application.



If everything is fine, you must see GWT Development Mode active in Eclipse containing a URL as shown below. Double click the URL to open the GWT application.



Because you are running your application in development mode, so you will need to install GWT plugin for your browser. Simply follow the onscreen instructions to install the plugin. If you already have GWT plugin set for your browser, then you should be able to see the following output:



Congratulations! you have implemented your first application using Google Web Toolkit (GWT).

GWT - DEPLOY APPLICATION

This tutorial will explain you how to create an application **war** file and how to deploy that in Apache Tomcat Webseve root. If you understood this simple example then you will also be able to

deploy a complex GWT application following the same steps.

Let us have working Eclipse IDE along with GWT plug in place and follow the following steps to create a GWT application:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to make sure business logic is working as per the requirements.
4	Finally, zip the content of the **war** folder of the application in the form of war file and deploy it in Apache Tomcat Webserver.
5	Launch your web application using appropriate URL as explained below in the last step.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body {
   text-align: center;
   font-family: verdana, sans-serif;
}
h1 {
  font-size: 2em;
  font-weight: bold;
  color: #777777;
  margin: 40px 0px 70px;
  text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**.

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>Hello World</h1>
<div ></div>

</body>
```

```
</html>
```

I modified HTML a little bit from previous example. Here I created a placeholder ... where we will insert some content using our entry point java class. So let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java**.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.RootPanel;

public class HelloWorld implements EntryPoint {
   public void onModuleLoad() {
      HTML html = new HTML("<p>Welcome to GWT application</p>");

      RootPanel.get("gwtContainer").add(html);
   }
}
```

Here we created on basic widgest HTML and added it inside the div tag having . We will study different GWT widgets in coming chapters.

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:

Create WAR File

Now our applictaion is working fine and we are ready to export it as a war file. Follow the following steps:

- Go into your project's **war** directory **C:\workspace\HelloWorld\war**
- Select all the files & folders available inside war directory.
- Zip all the selected files & folders in a file called *HelloWorld.zip*.
- Rename *HelloWorld.zip* to *HelloWorld.war*.

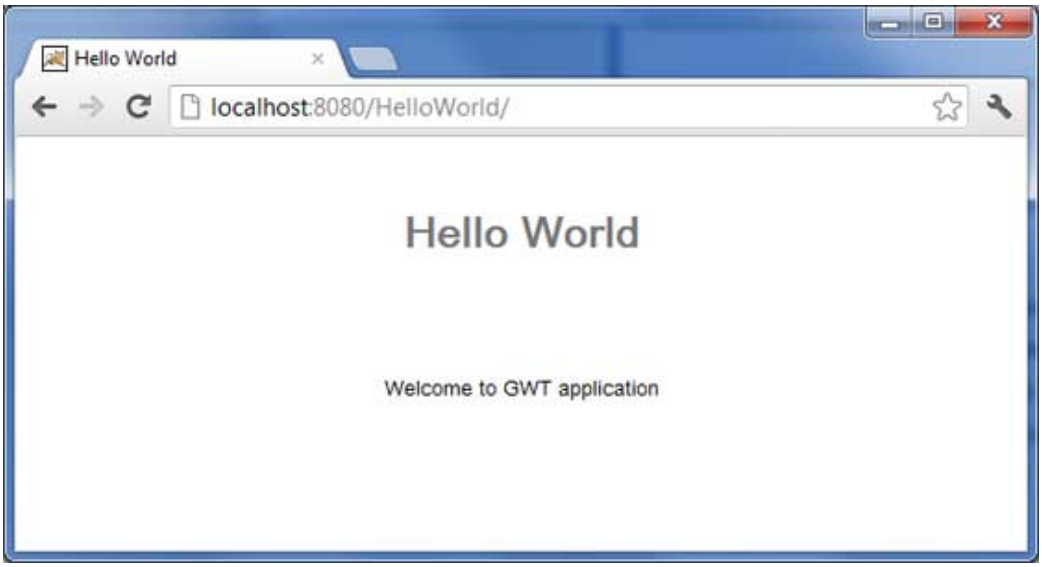
Deploy WAR file

- Stop the tomcat server.
- Copy the *HelloWorld.war* file to **tomcat installation directory > webapps folder.**
- Start the tomcat server.
- Look inside webapps directory, there should be a folder **helloworld** got created.
- Now HelloWorld.war is successfully deployed in Tomcat Webserver root.

Run Application

Enter a url in web browser: **http://localhost:8080/HelloWorld** to launch the application

Server name (localhost) and port (8080) may vary as per your tomcat configuration.



GWT - STYLE WITH CSS

GWT widgets rely on cascading style sheets (CSS) for visual styling. By default, the class name for each component is **gwt-<classname>**. For example, the Button widget has a default style of *gwt-Button* and similar way TextBox widgest has a default style of *gwt-TextBox*. In order to give all buttons and text boxes a larger font, you could put the following rule in your application's CSS file:

```
.gwt-Button  { font-size: 150%; }
.gwt-TextBox { font-size: 150%; }
```

By default, neither the browser nor GWT creates default **id** attributes for widgets. You must explicitly create a unique id for the elements which you can use in CSS. In order to give a particular button with id **my-button-id** a larger font, you could put the following rule in your application's CSS file:

```
#my-button-id { font-size: 150%; }
```

To set the id for a GWT widget, retrieve its DOM Element and then set the id attribute as follows:

```
Button b = new Button();
DOM.setElementAttribute(b.getElement(), "id", "my-button-id")
```

CSS Styling APIs

There are many APIs available to hangle CSS setting for any GWT widget. Following are few important APIs which will help you in your day to day web programming using GWT:

S.N.	API & Description
1	**public void setStyleName(java.lang.String style)** This method will clear any existing styles and set the widget style to the new CSS class provided using *style*.
2	**public void addStyleName(java.lang.String style)** This method will add a secondary or dependent style name to the widget. A secondary style name is an additional style name that is,so if there were any previous style names applied they are kept.
3	**public void removeStyleName(java.lang.String style)** This method will remove given style from the widget and leaves any others associated with the widget.
4	**public java.lang.String getStyleName()**

	This method gets all of the object's style names, as a space-separated list.
5	**public void setStylePrimaryName(java.lang.String style)** This method sets the object's primary style name and updates all dependent style names.

For example, let's define two new styles which we will apply to a text:

```
.gwt-Big-Text{
   font-size:150%;
}
.gwt-Small-Text{
   font-size:75%;
}
.gwt-Red-Text{
   color:red;
}
```

Now you can use *setStyleName(Style)* to change the default setting to new setting. After applying the below rule, a text's font will become large:

```
txtWidget.setStyleName("gwt-Big-Text");
```

We can apply a secondary CSS rule on the same widget to change its color as follows:

```
txtWidget.addStyleName("gwt-Red-Text");
```

Using above method you can add as many styles as you like to apply on a widget. If you remove first style from the button widget then second style will still remain with the text:

```
txtWidget.removeStyleName("gwt-Big-Text");
```

Primary & Secondary Styles

By default, the *primary style* name of a widget will be the default style name for its widget class for example *gwt-Button* for Button widgets. When we add and remove style names using AddStyleName() method, those styles are called secondary styles.

The final appearance of a widget is determined by the sum of all the secondary styles added to it, plus its primary style. You set the primary style of a widget with the *setStylePrimaryName(String)* method. To illustrate, let's say we have a Label widget. In our CSS file, we have the following rules defined:

```
.MyText {
  color: blue;
}

.BigText {
  font-size: large;
}

.LoudText {
  font-weight:  bold;
}
```

Let's suppose we want a particular label widget to always display blue text, and in some cases, use a larger, bold font for added emphasis. We could do something like this:

```
// set up our primary style
Label someText = new Label();
someText.setStylePrimaryName("MyText");
...

// later on, to really grab the user's attention
someText.addStyleName("BigText");
someText.addStyleName("LoudText");
...

// after the crisis is over
```

```
someText.removeStyleName("BigText");
someText.removeStyleName("LoudText");
```

Associating CSS Files

There are multiple approaches for associating CSS files with your module. Modern GWT applications typically use a combination of CssResource and UiBinder. We are using only first approach in our examples.

- Using a <link> tag in the host HTML page.
- Using the <stylesheet> element in the module XML file.
- Using a **CssResource** contained within a **ClientBundle**.
- Using an inline <ui:style> element in a **UiBinder** template.

GWT CSS Example

This example will take you through simple steps to apply different CSS rules on your GWT widgest. Let us have working Eclipse IDE along with GWT plug in place and follow the following steps to create a GWT application:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
.gwt-Button{
   font-size: 150%;
   font-weight: bold;
   width:100px;
```

```
   height:100px;
}
.gwt-Big-Text{
   font-size:150%;
}
.gwt-Small-Text{
   font-size:75%;
}
```

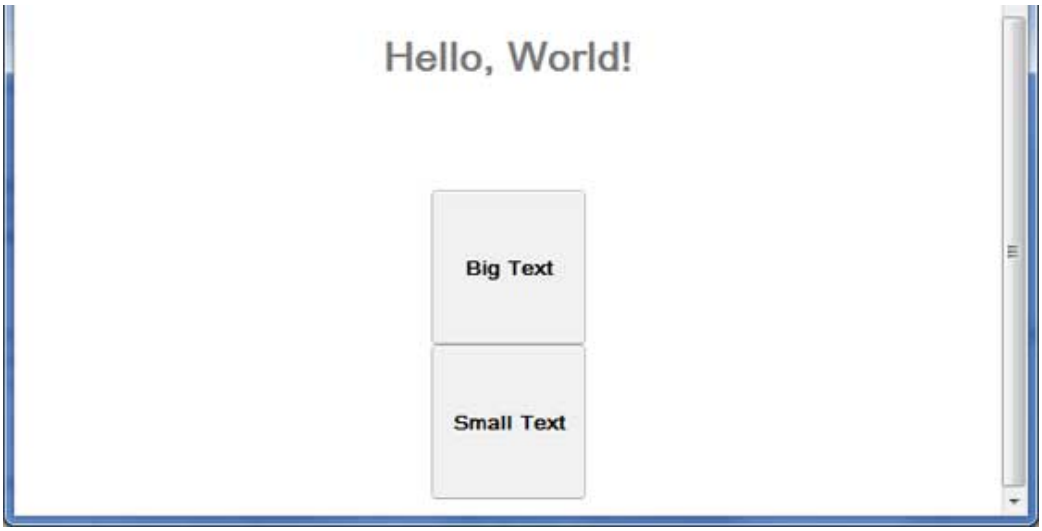
Following is the content of the modified HTML host file **war/HelloWorld.html** to accomodate two buttons.

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<div ><h1>Hello, World!</h1></div>
<div ></div>
<div ></div>

</body>
</html>
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** which will take care of adding two buttons in HTML and will apply custom CSS style.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.RootPanel;

public class HelloWorld implements EntryPoint {
   public void onModuleLoad() {

   // add button to change font to big when clicked.
   Button Btn1 = new Button("Big Text");
   Btn1.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
         RootPanel.get("mytext").setStyleName("gwt-Big-Text");
      }
   });

   // add button to change font to small when clicked.
   Button Btn2 = new Button("Small Text");
   Btn2.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
         RootPanel.get("mytext").setStyleName("gwt-Small-Text");
      }
   });

   RootPanel.get("gwtGreenButton").add(Btn1);
   RootPanel.get("gwtRedButton").add(Btn2);
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:



Now try clicking on the two buttons displayed and observe "Hello, World!" text which keeps changing its font upon clicking on the two buttons.

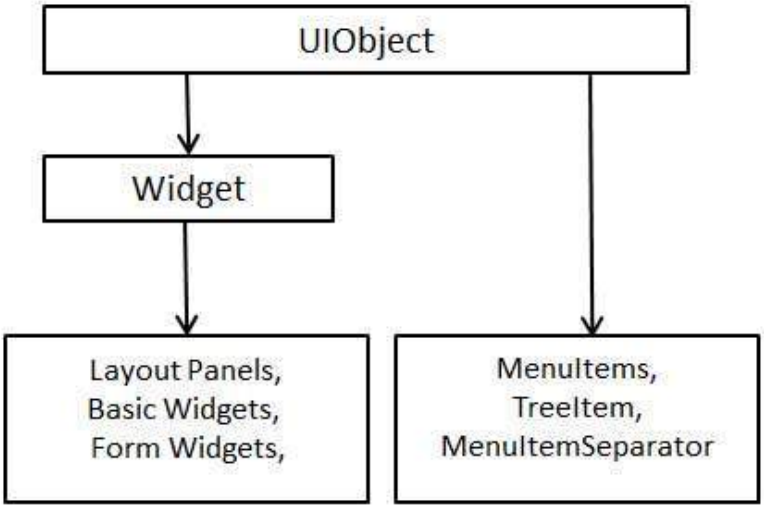
GWT - WIDGETS

Every user interface considers the following three main aspects:

- **UI elements** : Thes are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex which we will cover in this tutorial.
- **Layouts:** They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface). This part will be covered in Layout chapter.
- **Behavior:** These are events which occur when the user interacts with UI elements. This part will be covered in Event Handling chapter.

GWT UI Elements:

The GWT library provides classes in a well-defined class hierarchy to create complex web-based user interfaces. All classes in this component hierarchy has been derived from the **UIObject** base class as shown below:



Every Basic UI widget inherits properties from Widget class which in turn inherits properties from UIObject. Tree and Menu will be covered in complex widgets tutorial.

S.N.	Widget & Description
1	GWT UIObject Class This widget contains text, not interpreted as HTML using a element, causing it to

	be displayed with block layout.
2	GWT Widget Class This widget can contain HTML text and displays the html content using a element, causing it to be displayed with block layout.

Basic Widgets

Following are few important *Basic Widgets*:

S.N.	Widget & Description
1	Label This widget contains text, not interpreted as HTML using a element, causing it to be displayed with block layout.
2	HTML This widget can contain HTML text and displays the html content using a element, causing it to be displayed with block layout.
3	Image This widget displays an image at a given URL.
4	Anchor This widget represents a simple <a> element.

Form Widgets

Form widgets allows users to input data and provides them interaction capability with the application. Every Form widget inherits properties from Widget class which in turn inherits properties from UIObject and Wigdet classes.

Following are few important *Form Widgets*:

S.N.	Widget & Description
1	Button This widget represents a standard push button.
2	PushButton This widget represents a normal push button with custom styling.
3	ToggleButton This widget represents a stylish stateful button which allows the user to toggle between up and down states.
4	CheckBox This widget represents a standard check box widget. This class also serves as a base class for RadioButton.
5	RadioButton This widget represents a mutually-exclusive selection radio button widget.
6	ListBox This widget represents a list of choices to the user, either as a list box or as a drop-down list.
7	SuggestBox This widget represents a text box or text area which displays a pre-configured set of selections that match the user's input. Each SuggestBox is associated with a single SuggestOracle. The SuggestOracle is used to provide a set of selections given a specific query string.
8	TextBox This widget represents a single line text box.

9	PasswordTextBox This widget represents a text box that visually masks its input to prevent eavesdropping..
10	TextArea This widget represents a text box that allows multiple lines of text to be entered.
11	RichTextArea This widget represents a rich text editor that allows complex styling and formatting.
12	FileUpload This widget wraps the HTML <input type='file'> element.
13	Hidden This widget represets a hidden field in an HTML form.

Complex Widgets

Complex widgets allows users to advanced interaction capability with the application. Every Complex widget inherits properties from Widget class which in turn inherits properties from UIObject.

Following are few important *Complex Widgets*:

S.N.	Widget & Description
1	Tree This widget represents a standard hierarchical tree widget. The tree contains a hierarchy of TreeItems that the user can open, close, and select.
2	MenuBar This widget represents a standard menu bar widget. A menu bar can contain any number of menu items, each of which can either fire a Command or open a cascaded menu bar.
3	DatePicker This widget represents a standard GWT date picker.
4	CellTree This widget represents a view of a tree. This widget will only work in standards mode, which requires that the HTML page in which it is run have an explicit <!DOCTYPE> declaration.
5	CellList This widget represents a single column list of cells.
6	CellTable This widget represents a tabular view that supports paging and columns.
7	CellBrowser This widget represents a **browsable** view of a tree in which only a single node per level may be open at one time. This widget will only work in standards mode, which requires that the HTML page in which it is run have an explicit <!DOCTYPE> declaration.

GWT - LAYOUT PANELS

Layout Panels can contain other widgets. These panels controls the way widgets to be shown on User Interface. Every Panel widget inherits properties from Panel class which in turn inherits properties from Widget class and which in turn inherits properties from UIObject class.

S.N.	Widget & Description
1	GWT UIObject Class This widget contains text, not interpreted as HTML using a element, causing it to be displayed with block layout.
2	GWT Widget Class

	This widget can contain HTML text and displays the html content using a element, causing it to be displayed with block layout.
3	GWT Panel Class This is an is the abstract base class for all panels, which are widgets that can contain other widgets.

Layout Panels

Following are few important *Layout Panels*:

S.N.	Widget & Description
1	FlowPanel This widget represents a panel that formats its child widgets using the default HTML layout behavior.
2	HorizontalPanel This widget represents a panel that lays all of its widgets out in a single horizontal column.
3	VerticalPanel This widget represents a panel that lays all of its widgets out in a single vertical column.
4	HorizontalSplitPanel This widget represents a panel that arranges two widgets in a single horizontal row and allows the user to interactively change the proportion of the width dedicated to each of the two widgets. Widgets contained within a HorizontalSplitPanel will be automatically decorated with scrollbars when necessary.
5	VerticalSplitPanel This widget represents a A panel that arranges two widgets in a single vertical column and allows the user to interactively change the proportion of the height dedicated to each of the two widgets. Widgets contained within a VertialSplitPanel will be automatically decorated with scrollbars when necessary.
6	FlexTable This widget represents a flexible table that creates cells on demand. It can be jagged (that is, each row can contain a different number of cells) and individual cells can be set to span multiple rows or columns.
7	Grid This widget represents a A rectangular grid that can contain text, html, or a child Widget within its cells. It must be resized explicitly to the desired number of rows and columns.
8	DeckPanel panel that displays all of its child widgets in a 'deck', where only one can be visible at a time. It is used by TabPanel.
9	DockPanel This widget represents a panel that lays its child widgets out "docked" at its outer edges, and allows its last widget to take up the remaining space in its center.
10	HTMLPanel This widget represents a panel that contains HTML, and which can attach child widgets to identified elements within that HTML.
11	TabPanel This widget represents a panel that represents a tabbed set of pages, each of which contains another widget. Its child widgets are shown as the user selects the various tabs associated with them. The tabs can contain arbitrary HTML.
12	Composite This widget represents a type of widget that can wrap another widget, hiding the wrapped widget's methods. When added to a panel, a composite behaves exactly as if the widget it wraps had been added.
13	SimplePanel This widget represents a Base class for panels that contain only one widget.

14	ScrollPanel This widget represents a simple panel that wraps its contents in a scrollable area
15	FocusPanel This widget represents a simple panel that makes its contents focusable, and adds the ability to catch mouse and keyboard events.
16	FormPanel This widget represents a simple panel that makes its contents focusable, and adds the ability to catch mouse and keyboard events.
17	PopupPanel This widget represents a panel that can **pop up** over other widgets. It overlays the browser's client area (and any previously-created popups).
18	DialogBox This widget represents a form of popup that has a caption area at the top and can be dragged by the user. Unlike a PopupPanel, calls to PopupPanel.setWidth(String) and PopupPanel.setHeight(String) will set the width and height of the dialog box itself, even if a widget has not been added as yet.

GWT - EVENT HANDLING

GWT provides a event handler model similar to Java AWT or SWING User Interface frameworks.

- A listener interface defines one or more methods that the widget calls to announce an event. GWT provides a list of interfaces corresponding to various possible events.
- A class wishing to receive events of a particular type implements the associated handler interface and then passes a reference to itself to the widget to subscribe to a set of events.

For example, the **Button** class publishes **click** events so you will have to write a class to implement *ClickHandler* to handle **click** event.

Event Handler Interfaces

All GWT event handlers have been extended from *EventHandler* interface and each handler has only a single method with a single argument. This argument is always an object of associated event type. Each **event** object have a number of methods to manipulate the passed event object. For example for click event you will have to write your handler as follows:

```
/**
 * create a custom click handler which will call
 * onClick method when button is clicked.
 */
public class MyClickHandler implements ClickHandler {
   @Override
   public void onClick(ClickEvent event) {
      Window.alert("Hello World!");
   }
}
```

Now any class wishing to receive click events will call **addClickHandler()** to register an event handler as follows:

```
/**
 * create button and attach click handler
 */
Button button = new Button("Click Me!");
button.addClickHandler(new MyClickHandler());
```

Each widget supporting an event type will have a method of the form HandlerRegistration add**Foo**Handler(**Foo**Event) where **Foo** is the actual event like Click, Error, KeyPress etc.

Following is the list of important GWT event handlers and associated events and handler registration methods:

S.N.	Event Interface	Event Method & Description
1	BeforeSelectionHandler<I>	**void onBeforeSelection(BeforeSelectionEvent<I> event);** Called when BeforeSelectionEvent is fired.
2	BlurHandler	**void onBlur(BlurEvent event);** Called when BlurEvent is fired.
3	ChangeHandler	**void onChange(ChangeEvent event) ;** Called when a change event is fired.
4	ClickHandler	**void onClick(ClickEvent event);** Called when a native click event is fired.
5	CloseHandler<T>	**void onClose(CloseEvent<T> event) ;** Called when CloseEvent is fired.
6	ContextMenuHandler	**void onContextMenu(ContextMenuEvent event);** Called when a native context menu event is fired.
7	DoubleClickHandler	**void onDoubleClick(DoubleClickEvent event);** Called when a DoubleClickEvent is fired.
8	ErrorHandler	**void onError(ErrorEvent event);** Called when ErrorEvent is fired.
9	FocusHandler	**void onFocus(FocusEvent event) ;** Called when FocusEvent is fired.
10	FormPanel.SubmitCompleteHandler	**void onSubmitComplete(FormPanel.SubmitCompleteEvent event) ;** Fired when a form has been submitted successfully.
11	FormPanel.SubmitHandler	**void onSubmit(FormPanel.SubmitEvent event);** Fired when the form is submitted.
12	KeyDownHandler	**void onKeyDown(KeyDownEvent event);** Called when KeyDownEvent is fired.
13	KeyPressHandler	**void onKeyPress(KeyPressEvent event) ;** Called when KeyPressEvent is fired.
14	KeyUpHandler	**void onKeyUp(KeyUpEvent event) ;** Called when KeyUpEvent is fired.
15	LoadHandler	**void onLoad(LoadEvent event);** Called when LoadEvent is fired.
16	MouseDownHandler	**void onMouseDown(MouseDownEvent event) ;** Called when MouseDown is fired.
17	MouseMoveHandler	**void onMouseMove(MouseMoveEvent event);** Called when MouseMoveEvent is fired.
18	MouseOutHandler	**void onMouseOut(MouseOutEvent event) ;** Called when MouseOutEvent is fired.
19	MouseOverHandler	**void onMouseOver(MouseOverEvent event);** Called when MouseOverEvent is fired.
20	MouseUpHandler	**void onMouseUp(MouseUpEvent event) ;** Called when MouseUpEvent is fired.
21	MouseWheelHandler	**void onMouseWheel(MouseWheelEvent event) ;** Called when MouseWheelEvent is fired.
22	ResizeHandler	**void onResize(ResizeEvent event) ;** Fired when the widget is resized.
23	ScrollHandler	**void onScroll(ScrollEvent event) ;**

		Called when ScrollEvent is fired.
24	SelectionHandler<I>	**void onSelection(SelectionEvent<I> event) ;** Called when SelectionEvent is fired.
25	ValueChangeHandler<I>	**void onValueChange(ValueChangeEvent<I> event) ;** Called when ValueChangeEvent is fired.
26	Window.ClosingHandler	**void onWindowClosing(Window.ClosingEvent event) ;** Fired just before the browser window closes or navigates to a different site.
27	Window.ScrollHandler	**void onWindowScroll(Window.ScrollEvent event) ;** Fired when the browser window is scrolled.

Event Methods

As mentioned earlier, each handler has a single method with a single argument which holds the event object, for example *void onClick(ClickEvent event)* or *void onKeyDown(KeyDownEvent event)*. The event objects like *ClickEvent* and *KeyDownEvent* has few common methods which are listed below:

S.N.	Method & Description
1	**protected void dispatch(ClickHandler handler)** This method Should only be called by HandlerManager
2	**DomEvent.Type <FooHandler> getAssociatedType()** This method returns the type used to register **Foo** event.
3	**static DomEvent.Type<FooHandler> getType()** This method gets the event type associated with **Foo** events.
4	**public java.lang.Object getSource()** This method returns the source that last fired this event.
5	**protected final boolean isLive()** This method returns whether the event is live.
6	**protected void kill()** This method kills the event

Example

This example will take you through simple steps to show usage of a **Click** Event and **KeyDown** Event handling in GWT. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
```

```
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**.

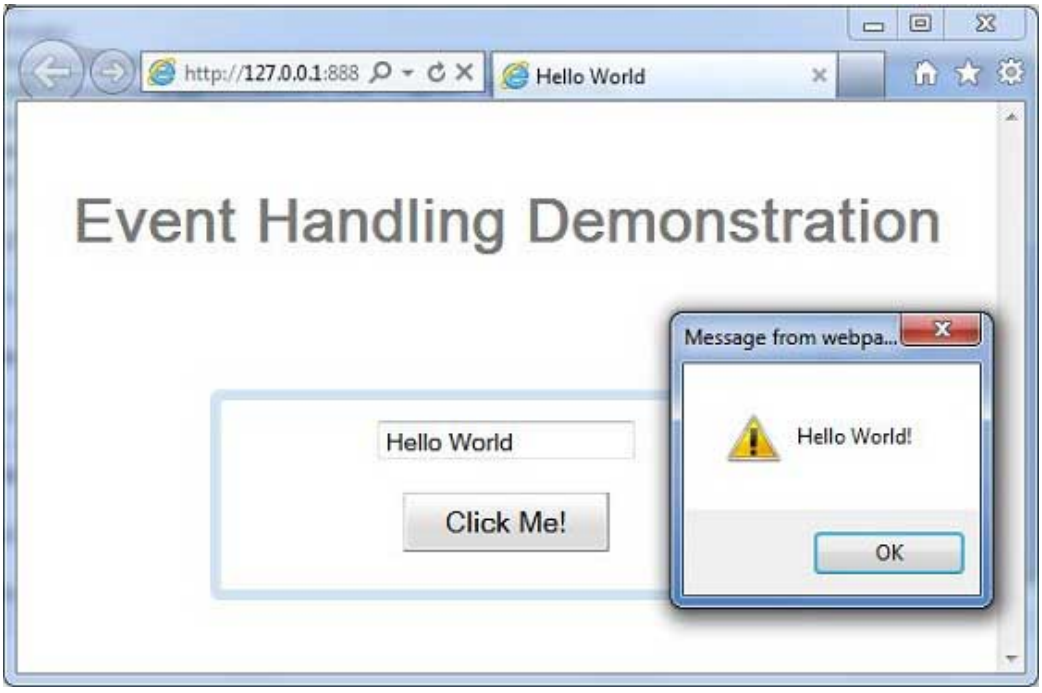
```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>Event Handling Demonstration</h1>
<div ></div>

</body>
</html>
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** which will demonstrate use of Event Handling in GWT.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyDownEvent;
import com.google.gwt.event.dom.client.KeyDownHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DecoratorPanel;
import com.google.gwt.user.client.ui.HasHorizontalAlignment;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

public class HelloWorld implements EntryPoint {
   public void onModuleLoad() {
      /**
       * create textbox and attach key down handler
       */
      TextBox textBox = new TextBox();
      textBox.addKeyDownHandler(new MyKeyDownHandler());

      /*
       * create button and attach click handler
       */
      Button button = new Button("Click Me!");
```

```
      button.addClickHandler(new MyClickHandler());

      VerticalPanel panel = new VerticalPanel();
      panel.setSpacing(10);
      panel.setHorizontalAlignment(HasHorizontalAlignment.ALIGN_CENTER);
      panel.setSize("300", "100");
      panel.add(textBox);
      panel.add(button);

      DecoratorPanel decoratorPanel = new DecoratorPanel();
      decoratorPanel.add(panel);
      RootPanel.get("gwtContainer").add(decoratorPanel);
   }

   /**
    * create a custom click handler which will call
    * onClick method when button is clicked.
    */
   private class MyClickHandler implements ClickHandler {
      @Override
      public void onClick(ClickEvent event) {
         Window.alert("Hello World!");
      }
   }

   /**
    * create a custom key down handler which will call
    * onKeyDown method when a key is down in textbox.
    */
   private class MyKeyDownHandler implements KeyDownHandler {
      @Override
      public void onKeyDown(KeyDownEvent event) {
         if(event.getNativeKeyCode() == KeyCodes.KEY_ENTER){
            Window.alert(((TextBox)event.getSource()).getValue());
         }
      }
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:



GWT - CUSTOM WIDGETS

GWT provides three ways to create custom user interface elements. There are three general strategies to follow:

- **Create a widget by extending Composite Class:** This is the most common and easiest

way to create custom widgets. Here you can use existing widgets to create composite view with custom properties.

- **Create a widget using GWT DOM API in JAVA:** GWT basic widgets are created in this way. Still its a very complicated way to create custom widget and should be used cautiously.
- **Use JavaScript and wrap it in a widget using JSNI:** This should generally only be done as a last resort. Considering the cross-browser implications of the native methods, it becomes very complicated and also becomes more difficult to debug.

Create Custom Widget with Composite Class

This example will take you through simple steps to show creation of a Custom Widget in GWT. Follow the following steps to update the GWT application we created in *GWT - Basic Widgets* chapter:

Here we are going to create a custom widget by extending Composite class, which is the easiest way to build custom widgets.

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**.

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
```

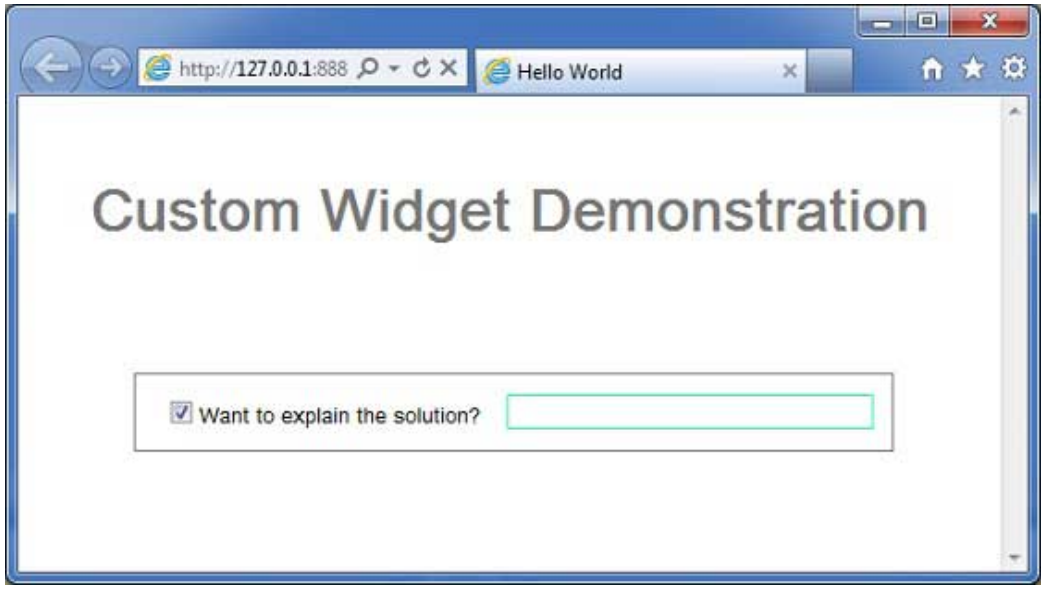
```
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>Custom Widget Demonstration</h1>
<div ></div>

</body>
</html>
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** which will demonstrate creation of a Custom widget.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.CheckBox;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;

public class HelloWorld implements EntryPoint {

   /**
   * A composite of a TextBox and a CheckBox that optionally enables it.
   */
   private static class OptionalTextBox extends Composite implements
   ClickHandler {

      private TextBox textBox = new TextBox();
      private CheckBox checkBox = new CheckBox();
      private boolean enabled = true;

      public boolean isEnabled() {
         return enabled;
      }

      public void setEnabled(boolean enabled) {
         this.enabled = enabled;
      }

      /**
      * Style this widget using .optionalTextWidget CSS class.<br/>
      * Style textbox using .optionalTextBox CSS class.<br/>
      * Style checkbox using .optionalCheckBox CSS class.<br/>
      * Constructs an OptionalTextBox with the given caption
      * on the check.
      * @param caption the caption to be displayed with the check box
      */
      public OptionalTextBox(String caption) {
         // place the check above the text box using a vertical panel.
         HorizontalPanel panel = new HorizontalPanel();
         // panel.setBorderWidth(1);
         panel.setSpacing(10);
         panel.add(checkBox);
         panel.add(textBox);

         // all composites must call initWidget() in their constructors.
         initWidget(panel);

         //set style name for entire widget
         setStyleName("optionalTextWidget");

         //set style name for text box
         textBox.setStyleName("optionalTextBox");

         //set style name for check box
         checkBox.setStyleName("optionalCheckBox");
         textBox.setWidth("200");
```

```
         // Set the check box's caption, and check it by default.
         checkBox.setText(caption);
         checkBox.setValue(enabled);
         checkBox.addClickHandler(this);
         enableTextBox(enabled,checkBox.getValue());
      }

      public void onClick(ClickEvent event) {
         if (event.getSource() == checkBox) {
            // When the check box is clicked,
            //update the text box's enabled state.
            enableTextBox(enabled,checkBox.getValue());
         }
      }

      private void enableTextBox(boolean enable,boolean isChecked){
         enable = (enable && isChecked) || (!enable && !isChecked);
         textBox.setStyleDependentName("disabled", !enable);
         textBox.setEnabled(enable);
      }
   }

   public void onModuleLoad() {
      // Create an optional text box and add it to the root panel.
      OptionalTextBox otb = new OptionalTextBox(
      "Want to explain the solution?");
      otb.setEnabled(true);
      RootPanel.get().add(otb);
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:



You can notice following points

- Creation of Custom Widget by extending Composite widget is pretty easy.
- We've created a widget with GWT inbuilt widgets, TextBox and CheckBox thus using the concept of reusability.
- TextBox get disabled/enabled depending on state of checkbox. We've provided an API to enable/disable the control.
- We've exposed internal widgets styles via documented CSS styles.

GWT - USING UIBINDER

Introduction

- The UiBinder is a framework designed to separate Functionality and View of User Interface.
- The UiBinder framework allows developers to build gwt applications as HTML pages with GWT widgets configured throughout them.
- The UiBinder framework makes easier collaboration with UI designers who are more comfortable with XML, HTML and CSS than Java source code
- The UIBinder provides a declarative way of defining User Interface.
- The UIBinder seperates the programmic logic from UI.
- The UIBinder is similar to what JSP is to Servlets.

UiBinder workflow

Step 1: Create UI Declaration XML File

Create a XML/HTML based User Interface declaration file. We've created a **Login.ui.xml** file in our example.

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
   xmlns:gwt='urn:import:com.google.gwt.user.client.ui'
   xmlns:res='urn:with:com.tutorialspoint.client.LoginResources'>
   <ui:with type="com.tutorialspoint.client.LoginResources" field="res">
   </ui:with>
   <gwt:HTMLPanel>
   ...
   </gwt:HTMLPanel>
</ui:UiBinder>
```

Step 2: Use ui:field for Later Binding

Use ui:field attribute in XML/HTML element to relate UI field in XML with UI field in JAVA file for later binding.

```
<gwt:Label ui:field="completionLabel1" />
<gwt:Label ui:field="completionLabel2" />
```

Step 3: Create Java counterpart of UI XML

Create Java based counterpart of XML based layout by extending Composite widget. We've created a **Login.java** file in our example.

```
package com.tutorialspoint.client;
...
public class Login extends Composite {
...
}
```

Step 4: Bind Java UI fields with UiField annotation

use @UiField annotation in **Login.java** to designate counterpart class members to bind to XML-based fields in **Login.ui.xml**

```
public class Login extends Composite {
   ...
   @UiField
   Label completionLabel1;

   @UiField
   Label completionLabel2;
   ...
}
```

Step 5: Bind Java UI with UI XML with UiTemplate annotation

Instruct GWT to bind java based component **Login.java** and XML based layout **Login.ui.xml** using @UiTemplate annotation

```
public class Login extends Composite {

   private static LoginUiBinder uiBinder = GWT.create(LoginUiBinder.class);

   /*
   * @UiTemplate is not mandatory but allows multiple XML templates
   * to be used for the same widget.
   * Default file loaded will be <class-name>.ui.xml
   */
   @UiTemplate("Login.ui.xml")
   interface LoginUiBinder extends UiBinder<Widget, Login> {
   }
   ...
}
```

Step 6: Create CSS File

Create an external CSS file**Login.css** and Java based Resource **LoginResources.java** file equivalent to css styles

```
.blackText {
   font-family: Arial, Sans-serif;
   color: #000000;
   font-size: 11px;
   text-align: left;
}
...
```

Step 7: Create Java based Resource File for CSS File

```
package com.tutorialspoint.client;
...
public interface LoginResources extends ClientBundle {
   public interface MyCss extends CssResource {
      String blackText();

      ...
   }

   @Source("Login.css")
   MyCss style();
}
```

Step 8: Attach CSS resource in Java UI Code file.

Attach an external CSS file**Login.css** using Contructor of Java based widget class **Login.java**

```
public Login() {
   this.res = GWT.create(LoginResources.class);
   res.style().ensureInjected();
   initWidget(uiBinder.createAndBindUi(this));
}
```

UIBinder Complete Example

This example will take you through simple steps to show usage of a UIBinder in GWT. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>
  <!-- Inherit the UiBinder module.                                -->
  <inherits name="com.google.gwt.uibinder.UiBinder"/>
  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

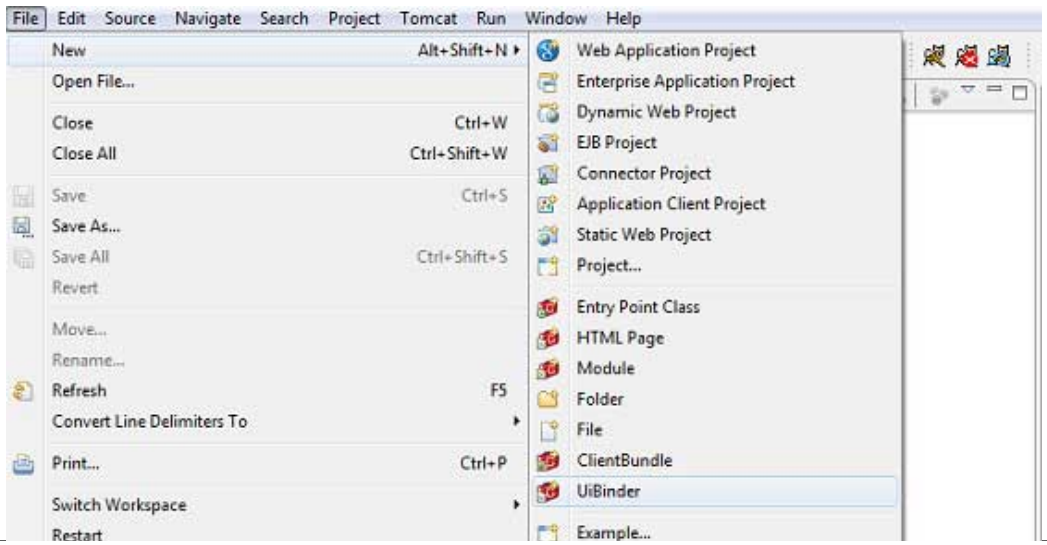
Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

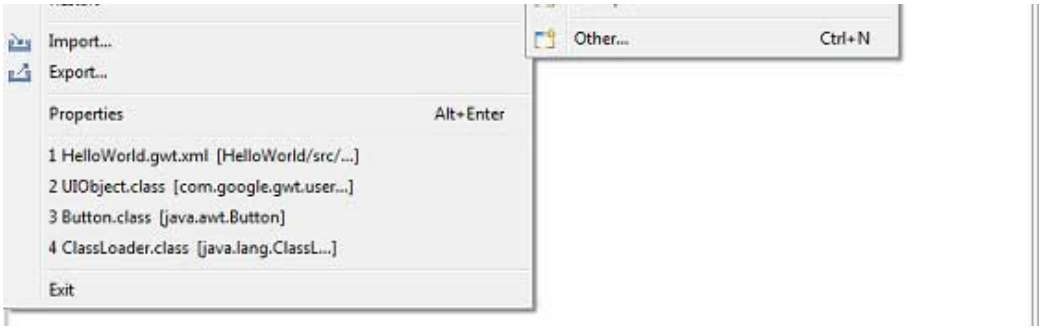
```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**.

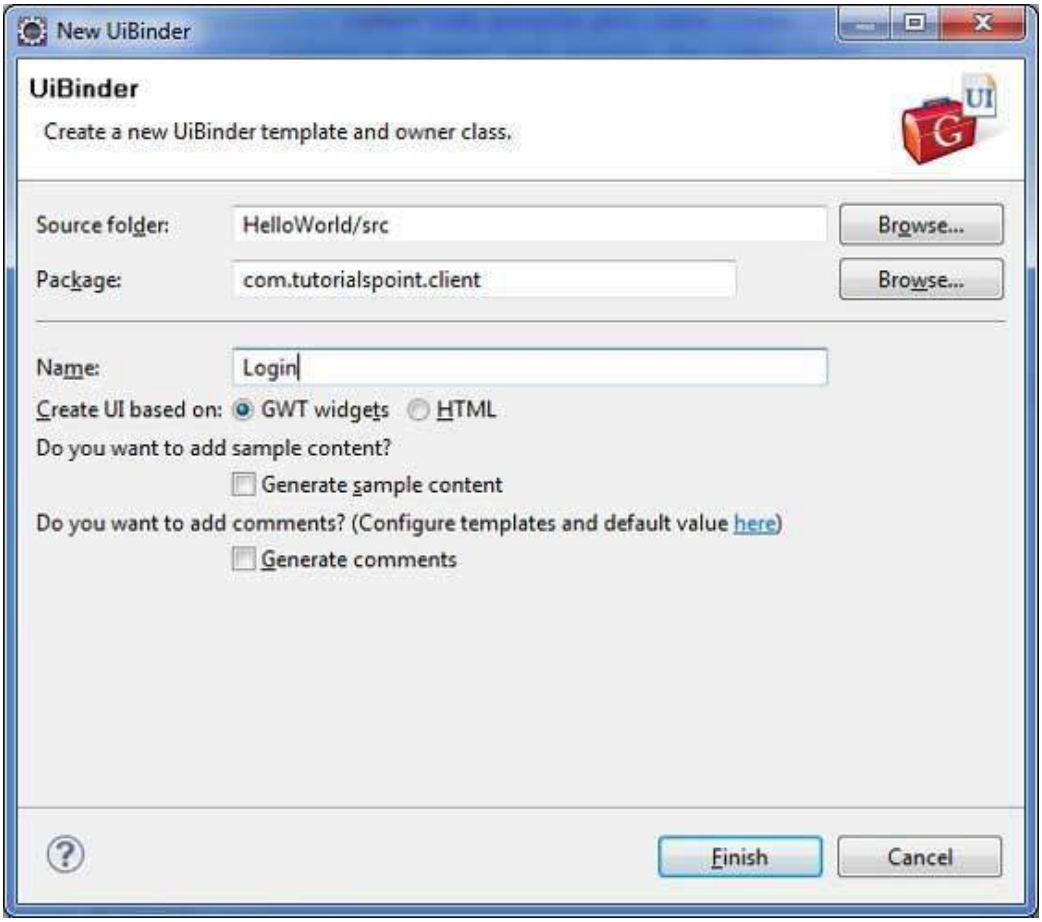
```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>UiBinder Demonstration</h1>
<div ></div>

</body>
</html>
```

Now create a new UiBinder template and owner class (File -> New -> UiBinder).





Choose the client package for the project and then name it Login. Leave all of the other defaults.Click Finish button and the plugin will create a new UiBinder template and owner class.



Now create Login.css file in the **src/com.tutorialspoint/client** package and place the following contents in it

```
.blackText {
   font-family: Arial, Sans-serif;
   color: #000000;
   font-size: 11px;
   text-align: left;
}

.redText {
   font-family: Arial, Sans-serif;
   color: #ff0000;
   font-size: 11px;
   text-align: left;
}

.loginButton {
   border: 1px solid #3399DD;
   color: #FFFFFF;
   background: #555555;
   font-size: 11px;
   font-weight: bold;
   margin: 0 5px 0 0;
   padding: 4px 10px 5px;
```

```
   text-shadow: 0 -1px 0 #3399DD;
}

.box {
   border: 1px solid #AACCEE;
   display: block;
   font-size: 12px;
   margin: 0 0 5px;
   padding: 3px;
   width: 203px;
}

.background {
   background-color: #999999;
   border: 1px none transparent;
   color: #000000;
   font-size: 11px;
   margin-left: -8px;
   margin-top: 5px;
   padding: 6px;
}
```

Now create LoginResources.java file in the **src/com.tutorialspoint/client** package and place the following contents in it

```
package com.tutorialspoint.client;

import com.google.gwt.resources.client.ClientBundle;
import com.google.gwt.resources.client.CssResource;

public interface LoginResources extends ClientBundle {
   /**
   * Sample CssResource.
   */
   public interface MyCss extends CssResource {
      String blackText();

      String redText();

      String loginButton();

      String box();

      String background();
   }

   @Source("Login.css")
   MyCss style();
}
```

Replace the contents of Login.ui.xml in **src/com.tutorialspoint/client** package with the following

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
   xmlns:gwt='urn:import:com.google.gwt.user.client.ui'
   xmlns:res='urn:with:com.tutorialspoint.client.LoginResources'>
   <ui:with type="com.tutorialspoint.client.LoginResources" field="res">
   </ui:with>
   <gwt:HTMLPanel>
      <div align="center">
         <gwt:VerticalPanel res:styleName="style.background">
            <gwt:Label text="Login" res:styleName="style.greyText" />
            <gwt:TextBox ui:field="loginBox" res:styleName="style.box" />
            <gwt:Label text="Password" res:styleName="style.greyText" />
            <gwt:PasswordTextBox ui:field="passwordBox"
            res:styleName="style.box" />
            <gwt:HorizontalPanel verticalAlignment="middle">
               <gwt:Button ui:field="buttonSubmit" text="Submit"
               res:styleName="style.loginButton" />
               <gwt:CheckBox ui:field="myCheckBox" />
               <gwt:Label ui:field="myLabel" text="Remember me"
               res:styleName="style.greyText" />
            </gwt:HorizontalPanel>
            <gwt:Label ui:field="completionLabel1"
            res:styleName="style.greyText" />
```

```
            <gwt:Label ui:field="completionLabel2"
            res:styleName="style.greyText" />
         </gwt:VerticalPanel>
      </div>
   </gwt:HTMLPanel>
</ui:UiBinder>
```

Replace the contents of Login.java in **src/com.tutorialspoint/client** package with the following

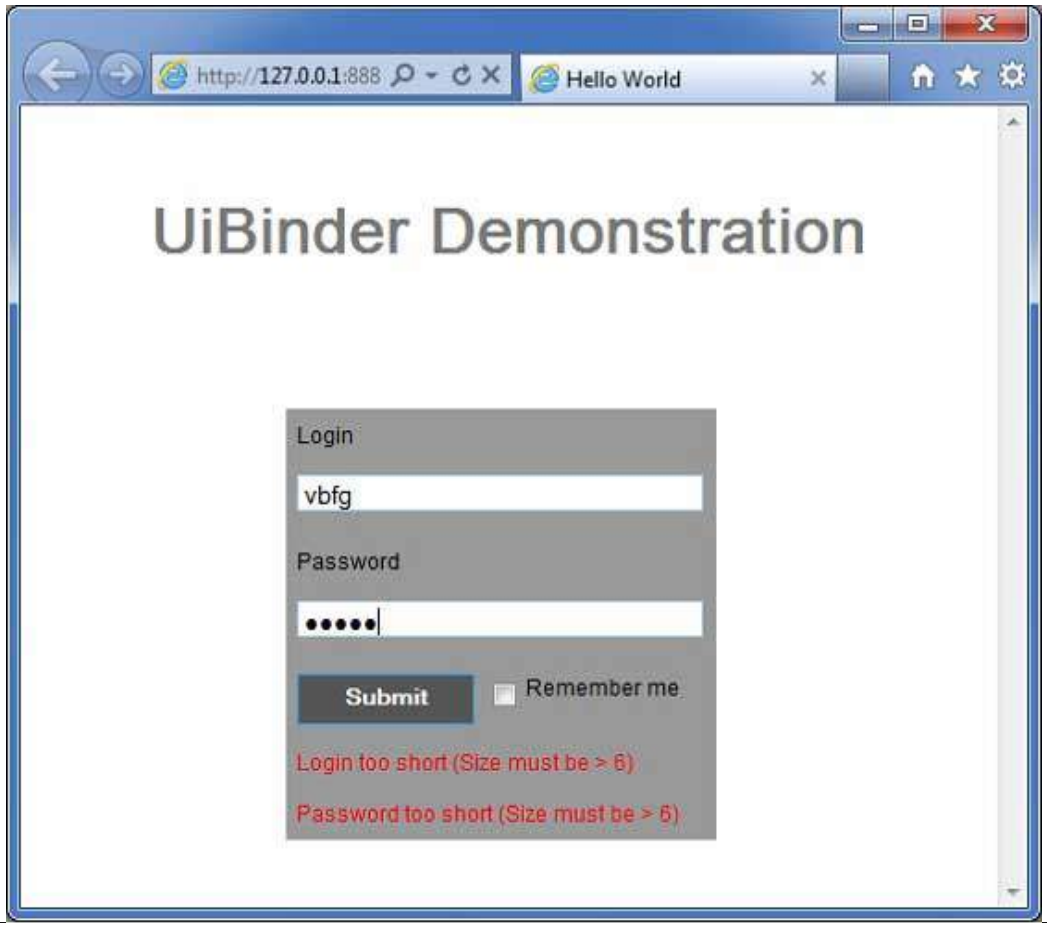
```
package com.tutorialspoint.client;

import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.logical.shared.ValueChangeEvent;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiField;
import com.google.gwt.uibinder.client.UiHandler;
import com.google.gwt.uibinder.client.UiTemplate;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.Widget;

public class Login extends Composite {

   private static LoginUiBinder uiBinder = GWT.create(LoginUiBinder.class);

   /*
   * @UiTemplate is not mandatory but allows multiple XML templates
   * to be used for the same widget.
   * Default file loaded will be <class-name>.ui.xml
   */
   @UiTemplate("Login.ui.xml")
   interface LoginUiBinder extends UiBinder<Widget, Login> {
   }

   @UiField(provided = true)
   final LoginResources res;

   public Login() {
      this.res = GWT.create(LoginResources.class);
      res.style().ensureInjected();
      initWidget(uiBinder.createAndBindUi(this));
   }

   @UiField
   TextBox loginBox;

   @UiField
   TextBox passwordBox;

   @UiField
   Label completionLabel1;

   @UiField
   Label completionLabel2;

   private Boolean tooShort = false;

   /*
   * Method name is not relevant, the binding is done according to the class
   * of the parameter.
   */
   @UiHandler("buttonSubmit")
   void doClickSubmit(ClickEvent event) {
      if (tooShort) {
         Window.alert("Login Successful!");
      } else {
         Window.alert("Login or Password is too short!");
      }
   }

   @UiHandler("loginBox")
```

```
   void handleLoginChange(ValueChangeEvent<String> event) {
      if (event.getValue().length() < 6) {
         completionLabel1.setText("Login too short (Size must be > 6)");
         tooShort = true;
      } else {
         tooShort = false;
         completionLabel1.setText("");
      }
   }

   @UiHandler("passwordBox")
   void handlePasswordChange(ValueChangeEvent<String> event) {
      if (event.getValue().length() < 6) {
         tooShort = true;
         completionLabel2.setText("Password too short (Size must be > 6)");
      } else {
         tooShort = false;
         completionLabel2.setText("");
      }
   }
}
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** which will demonstrate use of UiBinder.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.RootPanel;

public class HelloWorld implements EntryPoint {
   public void onModuleLoad() {
      RootPanel.get().add(new Login());
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:



GWT - RPC COMMUNICATION

A GWT based application is generally consists of a client side module and server side module. Client side code runs in browser and server side code runs in web server. Client side code has to make an HTTP request accross the network to access server side data.

RPC, Remote Procedure Call is the mechansim used by GWT in which client code can directly executes the server side methods.

- GWT RPC is servlet based.
- GWT RPC is asynchronous and client is never blocked during communication.
- Using GWT RPC Java objects can be sent directly between the client and the server (which are automatically serialized by the GWT framework).
- Server-side servlet is termed as **service**.
- Remote procedure call that is calling methods of server side servlets from client side code is referred to as **invoking a service**.

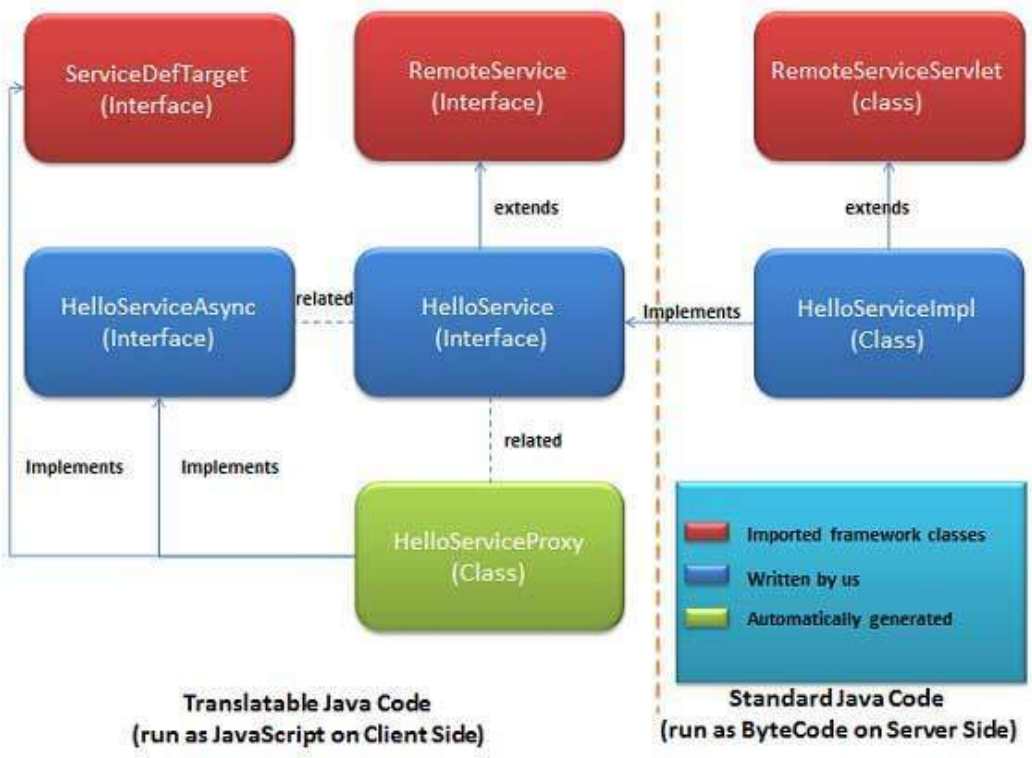
GWT RPC Components

Following are the three components used in GWT RPC communication mechanism

- A remote service (server-side servlet) that runs on the server.
- Client code to invoke that service.
- Java data objects which will be passed between client and server.

GWT client and server both serialize and deserialize data automatically so developers are not required to serialize/deserialize objects and data objects can travel over HTTP.

Following diagram is showing the RPC Architecture.



To start using RPC, we're required to follow the GWT conventions.

RPC Communication workflow

Step 1: Create a Serializable Model Class

Define a java model object at client side which should be serializable.

```
public class Message implements Serializable {
   ...
   private String message;
   public Message(){};

   public void setMessage(String message) {
      this.message = message;
   }
   ...
}
```

Step 2: Create a Service Interface

Define an interface for service on client side that extends RemoteService listing all service methods.

Use annotation @RemoteServiceRelativePath to map the service with a default path of remote servlet relative to the module base URL.

```
@RemoteServiceRelativePath("message")
public interface MessageService extends RemoteService {
   Message getMessage(String input);
}
```

Step 2: Create a Async Service Interface

Define an asynchronous interface to service on client side (at same location as service mentioned above) which will be used in the GWT client code.

```
public interface MessageServiceAsync {
   void getMessage(String input, AsyncCallback<Message> callback);
}
```

Step 3: Create a Service Implementation Servlet class

Implement the interface at server side and that class should extends RemoteServiceServlet class.

```
public class MessageServiceImpl extends RemoteServiceServlet
   implements MessageService{
   ...
   public Message getMessage(String input) {
      String messageString = "Hello " + input + "!";
      Message message = new Message();
      message.setMessage(messageString);
      return message;
   }
}
```

Step 4: Update Web.xml to include Servlet declaration

Edit the web application deployment descriptor (web.xml) to include MessageServiceImpl Servlet declaration.

```
<web-app>
   ...
   <servlet>
      <servlet-name>messageServiceImpl</servlet-name>
      <servlet-class>com.tutorialspoint.server.MessageServiceImpl
      </servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>messageServiceImpl</servlet-name>
      <url-pattern>/helloworld/message</url-pattern>
   </servlet-mapping>
</web-app>
```

Step 5: Make the remote procedure call in Application Code

Create the service proxy class.

```
MessageServiceAsync messageService = GWT.create(MessageService.class);
```

Create the AsyncCallback Handler to handle RPC callback in which server returns the Message back to client

```
class MessageCallBack implements AsyncCallback<Message> {

   @Override
   public void onFailure(Throwable caught) {
      Window.alert("Unable to obtain server response: "
      + caught.getMessage());
   }

   @Override
   public void onSuccess(Message result) {
      Window.alert(result.getMessage());
   }
}
```

Call Remote service when user interacts with UI

```
public class HelloWorld implements EntryPoint {
   ...
   public void onModuleLoad() {
   ...
      buttonMessage.addClickHandler(new ClickHandler() {
         @Override
         public void onClick(ClickEvent event) {
            messageService.getMessage(txtName.getValue(),
            new MessageCallBack());
         }});
   ...
   }
}
```

RPC Communication Complete Example

This example will take you through simple steps to show example of a RPC Communication in GWT. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>
  <!-- Inherit the UiBinder module.                                -->
  <inherits name="com.google.gwt.uibinder.UiBinder"/>
  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>
```

```
</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**.

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>RPC Communication Demonstration</h1>
<div ></div>

</body>
</html>
```

Now create Message.java file in the **src/com.tutorialspoint/client** package and place the following contents in it

```
package com.tutorialspoint.client;

import java.io.Serializable;

public class Message implements Serializable {

   private static final long serialVersionUID = 1L;
   private String message;
   public Message(){};

   public void setMessage(String message) {
      this.message = message;
   }

   public String getMessage() {
      return message;
   }
}
```

Now create MessageService.java file in the **src/com.tutorialspoint/client** package and place the following contents in it

```
package com.tutorialspoint.client;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

@RemoteServiceRelativePath("message")
public interface MessageService extends RemoteService {
   Message getMessage(String input);
}
```

Now create MessageServiceAsync.java file in the **src/com.tutorialspoint/client** package and place the following contents in it

```
package com.tutorialspoint.client;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface MessageServiceAsync {
   void getMessage(String input, AsyncCallback<Message> callback);
}
```

Now create MessageServiceImpl.java file in the **src/com.tutorialspoint/server** package and place the following contents in it

```
package com.tutorialspoint.server;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.tutorialspoint.client.Message;
import com.tutorialspoint.client.MessageService;

public class MessageServiceImpl extends RemoteServiceServlet
   implements MessageService{

   private static final long serialVersionUID = 1L;

   public Message getMessage(String input) {
      String messageString = "Hello " + input + "!";
      Message message = new Message();
      message.setMessage(messageString);
      return message;
   }
}
```

Update the content of the modified web application deployment descriptor **war/WEB-INF/web.xml** to include MessageServiceImpl Servlet declaration .

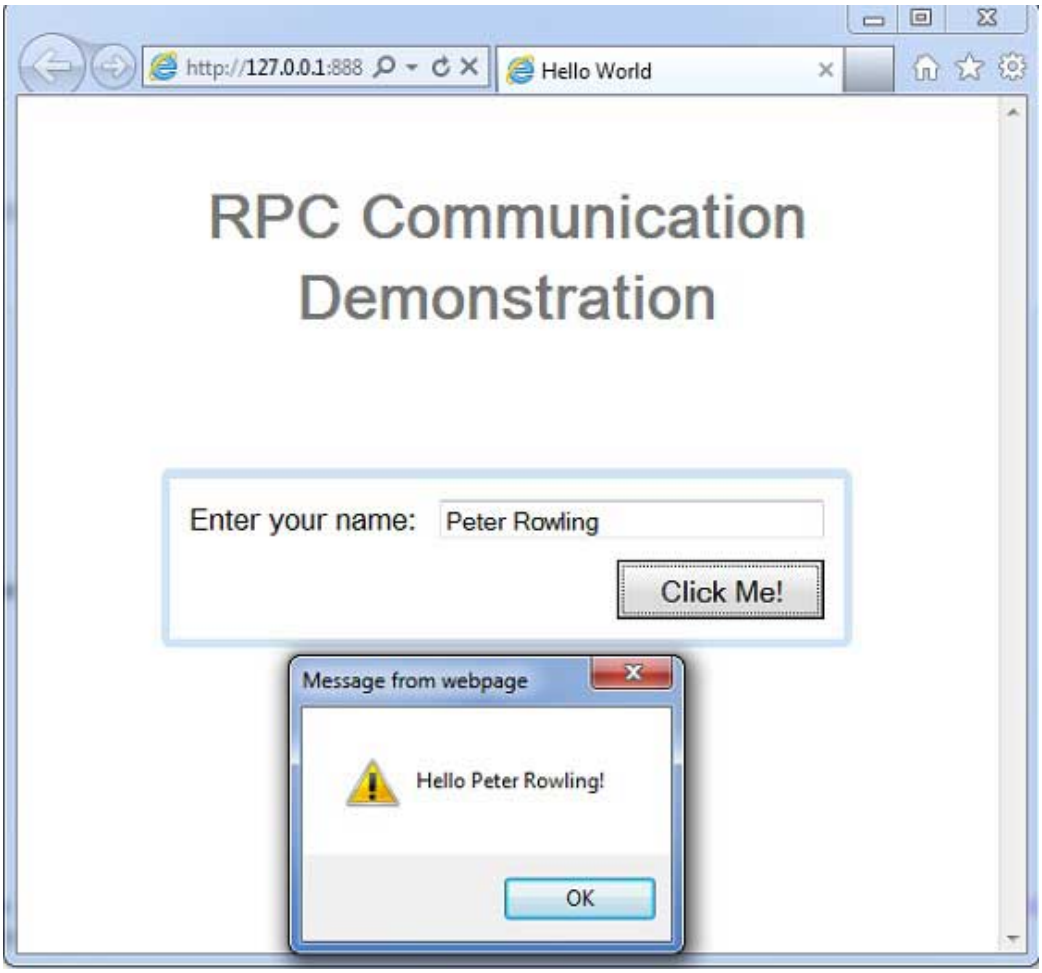
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
   <!-- Default page to serve -->
   <welcome-file-list>
      <welcome-file>HelloWorld.html</welcome-file>
   </welcome-file-list>
   <servlet>
      <servlet-name>messageServiceImpl</servlet-name>
      <servlet-class>com.tutorialspoint.server.MessageServiceImpl
      </servlet-class>
   </servlet>

   <servlet-mapping>
      <servlet-name>messageServiceImpl</servlet-name>
      <url-pattern>/helloworld/message</url-pattern>
   </servlet-mapping>
</web-app>
```

Replace the contents of HelloWorld.java in **src/com.tutorialspoint/client** package with the following

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DecoratorPanel;
import com.google.gwt.user.client.ui.HasHorizontalAlignment;
```

```
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

public class HelloWorld implements EntryPoint {

   private MessageServiceAsync messageService =
   GWT.create(MessageService.class);

   private class MessageCallBack implements AsyncCallback<Message> {
      @Override
      public void onFailure(Throwable caught) {
         /* server side error occured */
         Window.alert("Unable to obtain server response: "
         + caught.getMessage());
      }
      @Override
      public void onSuccess(Message result) {
          /* server returned result, show user the message */
         Window.alert(result.getMessage());
      }
   }

   public void onModuleLoad() {
      /*create UI */
      final TextBox txtName = new TextBox();
      txtName.setWidth("200");
      txtName.addKeyUpHandler(new KeyUpHandler() {
         @Override
         public void onKeyUp(KeyUpEvent event) {
            if(event.getNativeKeyCode() == KeyCodes.KEY_ENTER){
               /* make remote call to server to get the message */
               messageService.getMessage(txtName.getValue(),
               new MessageCallBack());
            }
         }
      });
      Label lblName = new Label("Enter your name: ");

      Button buttonMessage = new Button("Click Me!");

      buttonMessage.addClickHandler(new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
         /* make remote call to server to get the message */
         messageService.getMessage(txtName.getValue(),
         new MessageCallBack());
      }});

      HorizontalPanel hPanel = new HorizontalPanel();
      hPanel.add(lblName);
      hPanel.add(txtName);
      hPanel.setCellWidth(lblName, "130");

      VerticalPanel vPanel = new VerticalPanel();
      vPanel.setSpacing(10);
      vPanel.add(hPanel);
      vPanel.add(buttonMessage);
      vPanel.setCellHorizontalAlignment(buttonMessage,
      HasHorizontalAlignment.ALIGN_RIGHT);

      DecoratorPanel panel = new DecoratorPanel();
      panel.add(vPanel);

      // Add widgets to the root panel.
      RootPanel.get("gwtContainer").add(panel);
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:



GWT - JUNIT INTEGRATION

GWT provides execellent support for automated testing of client side code using JUnit testing framework. In this article we'll demonstrate GWT and JUNIT integration.

Download Junit archive

JUnit Official Site: http://www.junit.org

Download Junit-4.10.jar

OS	Archive name
Windows	junit4.10.jar
Linux	junit4.10.jar
Mac	junit4.10.jar

Store the downloaded jar file to some location on your computer. We've stored it at **C:/ > JUNIT**

Locate GWT installation folder

OS	GWT installation folder
Windows	C:\GWT\gwt-2.1.0
Linux	/usr/local/GWT/gwt-2.1.0
Mac	/Library/GWT/gwt-2.1.0

GWTTestCase Class

GWT provides GWTTestCase base class which provides JUnit integration. Running a compiled class which extends GWTTestCase under JUnit launches the HtmlUnit browser which serves to emulate your application behavior during test execution.

GWTTestCase is a derived class from JUnit's TestCase and it can be run using JUnit TestRunner.

Using webAppCreator

GWT provides a special command line tool **webAppCreator** which can generate a starter test case for us, plus ant targets and eclipse launch configs for testing in both development mode and production mode.

Open command prompt and go to **C:\ > GWT_WORKSPACE >** where you want to create a new project with test support.Run the following command

```
C:\GWT_WORKSPACE>C:\GWT\gwt-2.1.0\webAppCreator
                 -out HelloWorld
                 -junit C:\JUNIT\junit-4.10.jar
                 com.tutorialspoint.HelloWorld
```

Noteworthy Points

- We are executing webAppCreator command line utility.
- HelloWorld is the name of the project to be created
- -junit option instructs webAppCreator to add junit suppport to project
- com.tutorialspoint.HelloWorld is the name of the module

Verify the output.

```
Created directory HelloWorld\src
Created directory HelloWorld\war
Created directory HelloWorld\war\WEB-INF
Created directory HelloWorld\war\WEB-INF\lib
Created directory HelloWorld\src\com\tutorialspoint
Created directory HelloWorld\src\com\tutorialspoint\client
Created directory HelloWorld\src\com\tutorialspoint\server
Created directory HelloWorld\src\com\tutorialspoint\shared
Created directory HelloWorld\test\com\tutorialspoint
Created directory HelloWorld\test\com\tutorialspoint\client
Created file HelloWorld\src\com\tutorialspoint\HelloWorld.gwt.xml
Created file HelloWorld\war\HelloWorld.html
Created file HelloWorld\war\HelloWorld.css
Created file HelloWorld\war\WEB-INF\web.xml
Created file HelloWorld\src\com\tutorialspoint\client\HelloWorld.java
Created file
HelloWorld\src\com\tutorialspoint\client\GreetingService.java
Created file
HelloWorld\src\com\tutorialspoint\client\GreetingServiceAsync.java
Created file
HelloWorld\src\com\tutorialspoint\server\GreetingServiceImpl.java
Created file HelloWorld\src\com\tutorialspoint\shared\FieldVerifier.java
Created file HelloWorld\build.xml
Created file HelloWorld\README.txt
Created file HelloWorld\test\com\tutorialspoint\HelloWorldJUnit.gwt.xml
Created file HelloWorld\test\com\tutorialspoint\client\HelloWorldTest.java
Created file HelloWorld\.project
Created file HelloWorld\.classpath
Created file HelloWorld\HelloWorld.launch
Created file HelloWorld\HelloWorldTest-dev.launch
Created file HelloWorld\HelloWorldTest-prod.launch
```

Understanding the test class: HelloWorldTest.java

```
package com.tutorialspoint.client;

import com.tutorialspoint.shared.FieldVerifier;
import com.google.gwt.core.client.GWT;
import com.google.gwt.junit.client.GWTTestCase;
import com.google.gwt.user.client.rpc.AsyncCallback;
```

```
import com.google.gwt.user.client.rpc.ServiceDefTarget;

/**
 * GWT JUnit tests must extend GWTTestCase.
 */
public class HelloWorldTest extends GWTTestCase {

   /**
   * must refer to a valid module that sources this class.
   */
   public String getModuleName() {
      return "com.tutorialspoint.HelloWorldJUnit";
   }

   /**
   * tests the FieldVerifier.
   */
   public void testFieldVerifier() {
      assertFalse(FieldVerifier.isValidName(null));
      assertFalse(FieldVerifier.isValidName(""));
      assertFalse(FieldVerifier.isValidName("a"));
      assertFalse(FieldVerifier.isValidName("ab"));
      assertFalse(FieldVerifier.isValidName("abc"));
      assertTrue(FieldVerifier.isValidName("abcd"));
   }

   /**
   * this test will send a request to the server using the greetServer
   *  method in GreetingService and verify the response.
   */
   public void testGreetingService() {
      /* create the service that we will test. */
      GreetingServiceAsync greetingService =
      GWT.create(GreetingService.class);
      ServiceDefTarget target = (ServiceDefTarget) greetingService;
      target.setServiceEntryPoint(GWT.getModuleBaseURL()
      + "helloworld/greet");

      /* since RPC calls are asynchronous, we will need to wait
       for a response after this test method returns. This line
       tells the test runner to wait up to 10 seconds
       before timing out. */
      delayTestFinish(10000);

      /* send a request to the server. */
      greetingService.greetServer("GWT User",
         new AsyncCallback<String>() {
         public void onFailure(Throwable caught) {
            /* The request resulted in an unexpected error. */
            fail("Request failure: " + caught.getMessage());
         }

         public void onSuccess(String result) {
            /* verify that the response is correct. */
            assertTrue(result.startsWith("Hello, GWT User!"));

            /* now that we have received a response, we need to
             tell the test runner that the test is complete.
             You must call finishTest() after an asynchronous test
             finishes successfully, or the test will time out.*/
            finishTest();
         }
      });
   }
}
```

Noteworthy Points

Sr.	Note
1	HelloWorldTest class was generated in the com.tutorialspoint.client package under the HelloWorld/test directory.

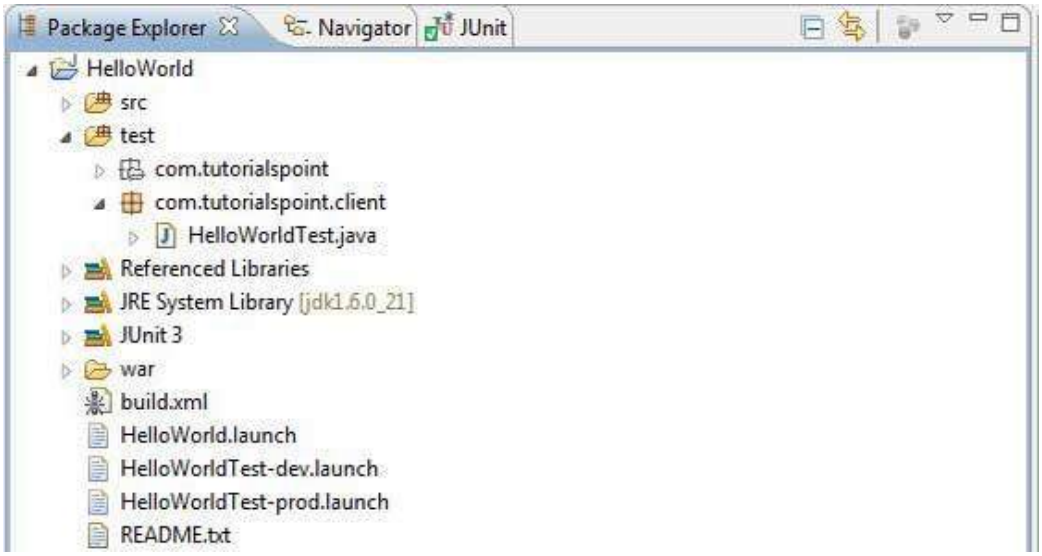
2	HelloWorldTest class will contain unit test cases for HelloWorld.
3	HelloWorldTest class extends the GWTTestCase class in the com.google.gwt.junit.client package.
4	HelloWorldTest class has an abstract method (getModuleName) that must return the name of the GWT module. For HelloWorld, this is com.tutorialspoint.HelloWorldJUnit.
5	HelloWorldTest class is generated with two sample test cases testFieldVerifier, testSimple. We've added testGreetingService.
6	These methods use one of the many assert* functions that it inherits from the JUnit Assert class, which is an ancestor of GWTTestCase.
7	The assertTrue(boolean) function asserts that the boolean argument passed in evaluates to true. If not, the test will fail when run in JUnit.

GWT - JUnit Integration Complete Example

This example will take you through simple steps to show example of JUnit Integration in GWT. Follow the following steps to update the GWT application we created above

Step	Description
1	Import the project with a name *HelloWorld* in eclipse using import existing project wizard (**File > Import > General > Existing Projects into workspace**).
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following will be the project structure in eclipse.



Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>
  <!-- Inherit the UiBinder module.                                -->
  <inherits name="com.google.gwt.uibinder.UiBinder"/>
  <!-- Specify the app entry point class.                          -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>
```

```
  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**.

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>JUnit Integration Demonstration</h1>
<div ></div>

</body>
</html>
```

Replace the contents of HelloWorld.java in **src/com.tutorialspoint/client** package with the following

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DecoratorPanel;
import com.google.gwt.user.client.ui.HasHorizontalAlignment;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

public class HelloWorld implements EntryPoint {

   public void onModuleLoad() {
      /*create UI */
      final TextBox txtName = new TextBox();
      txtName.setWidth("200");
      txtName.addKeyUpHandler(new KeyUpHandler() {
         @Override
         public void onKeyUp(KeyUpEvent event) {
            if(event.getNativeKeyCode() == KeyCodes.KEY_ENTER){
               Window.alert(getGreeting(txtName.getValue()));
            }
```

```
            }
         });
         Label lblName = new Label("Enter your name: ");

         Button buttonMessage = new Button("Click Me!");

         buttonMessage.addClickHandler(new ClickHandler() {
         @Override
         public void onClick(ClickEvent event) {
            Window.alert(getGreeting(txtName.getValue()));
         }});

         HorizontalPanel hPanel = new HorizontalPanel();
         hPanel.add(lblName);
         hPanel.add(txtName);
         hPanel.setCellWidth(lblName, "130");

         VerticalPanel vPanel = new VerticalPanel();
         vPanel.setSpacing(10);
         vPanel.add(hPanel);
         vPanel.add(buttonMessage);
         vPanel.setCellHorizontalAlignment(buttonMessage,
         HasHorizontalAlignment.ALIGN_RIGHT);

         DecoratorPanel panel = new DecoratorPanel();
         panel.add(vPanel);

         // Add widgets to the root panel.
         RootPanel.get("gwtContainer").add(panel);
      }

      public String getGreeting(String name){
         return "Hello "+name+"!";
      }
}
```

Replace the contents of HelloWorldTest.java in **test/com.tutorialspoint/client** package with the following

```
package com.tutorialspoint.client;

import com.tutorialspoint.shared.FieldVerifier;
import com.google.gwt.core.client.GWT;
import com.google.gwt.junit.client.GWTTestCase;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;

/**
 * GWT JUnit tests must extend GWTTestCase.
 */
public class HelloWorldTest extends GWTTestCase {

   /**
   * must refer to a valid module that sources this class.
   */
   public String getModuleName() {
      return "com.tutorialspoint.HelloWorldJUnit";
   }

   /**
   * tests the FieldVerifier.
   */
   public void testFieldVerifier() {
      assertFalse(FieldVerifier.isValidName(null));
      assertFalse(FieldVerifier.isValidName(""));
      assertFalse(FieldVerifier.isValidName("a"));
      assertFalse(FieldVerifier.isValidName("ab"));
      assertFalse(FieldVerifier.isValidName("abc"));
      assertTrue(FieldVerifier.isValidName("abcd"));
   }

   /**
   * this test will send a request to the server using the greetServer
   *   method in GreetingService and verify the response.
```

```
   */
   public void testGreetingService() {
      /* create the service that we will test. */
      GreetingServiceAsync greetingService =
      GWT.create(GreetingService.class);
      ServiceDefTarget target = (ServiceDefTarget) greetingService;
      target.setServiceEntryPoint(GWT.getModuleBaseURL()
      + "helloworld/greet");

      /* since RPC calls are asynchronous, we will need to wait
       for a response after this test method returns. This line
       tells the test runner to wait up to 10 seconds
       before timing out. */
      delayTestFinish(10000);

      /* send a request to the server. */
      greetingService.greetServer("GWT User",
         new AsyncCallback<String>() {
         public void onFailure(Throwable caught) {
            /* The request resulted in an unexpected error. */
            fail("Request failure: " + caught.getMessage());
         }

         public void onSuccess(String result) {
            /* verify that the response is correct. */
            assertTrue(result.startsWith("Hello, GWT User!"));

            /* now that we have received a response, we need to
             tell the test runner that the test is complete.
             You must call finishTest() after an asynchronous test
             finishes successfully, or the test will time out.*/
            finishTest();
         }
      });

      /**
      * tests the getGreeting method.
      */
      public void testGetGreeting() {
         HelloWorld helloWorld = new HelloWorld();
         String name = "Robert";
         String expectedGreeting = "Hello "+name+"!";
         assertEquals(expectedGreeting,helloWorld.getGreeting(name));
      }
   }
}
```

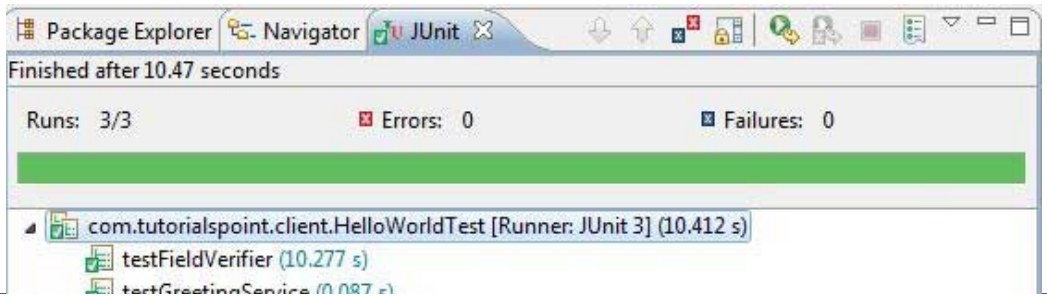
Run test cases in Eclipse using generated launch configurations.

We'll run unit tests in Eclipse using the launch configurations generated by webAppCreator for both development mode and production mode.

Run the JUnit test in development mode.

- From the Eclipse menu bar, select Run > Run Configurations...
- Under JUnit section, select HelloWorldTest-dev
- To save the changes to the Arguments, press Apply
- To run the test, press Run

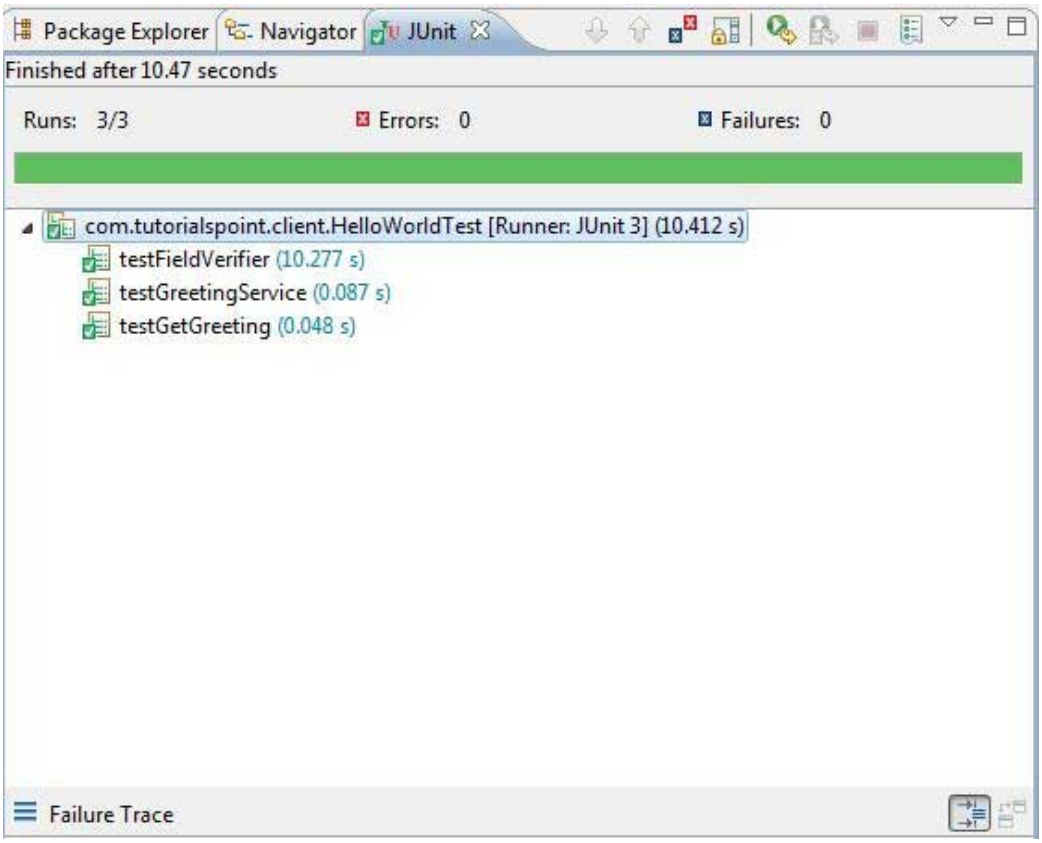
If everything is fine with your application, this will produce following result:



Run the JUnit test in production mode.

- From the Eclipse menu bar, select Run > Run Configurations...
- Under JUnit section, select HelloWorldTest-prod
- To save the changes to the Arguments, press Apply
- To run the test, press Run

If everything is fine with your application, this will produce following result:



GWT - DEBUGGING APPLICATION

GWT provides execellent capability of debugging client side as well as server side code.

- During development mode, GWT Application is in Java code based and is not translated to JavaScript.
- When an application is running in development mode, the Java Virtual Machine (JVM) is actually executing the application code as compiled Java bytecode, using GWT capability to connect to a browser window.

- GWT uses browser based plugin to connect to JVM.
- So developers are free to use any Java based IDE to debug both client-side GWT Code as well as server-side code.

In this article we'll demonstrate usage of debugging GWT Client code using Eclipse. We'll do the following tasks

- Set break points in the code and see them in BreakPoint Explorer.
- Step through the code line by line during debugging.
- View the values of variable.
- Inspect the values of all the variables.
- Inspect the value of an expression.
- Display the stack frame for suspended threads.

Debugging Example

This example will take you through simple steps to demonstrate debugging a GWT application. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>

  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
.gwt-Label{
```

```
   font-size: 150%;
   font-weight: bold;
   color:red;
   padding:5px;
   margin:5px;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html** to accomodate two buttons.

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1>Debugging Application Demonstration</h1>
<div ></div>

</body>
</html>
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** using which we will demonstrate debugging capability of GWT Code.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DecoratorPanel;
import com.google.gwt.user.client.ui.HasHorizontalAlignment;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

public class HelloWorld implements EntryPoint {

   public void onModuleLoad() {
      /*create UI */
      final TextBox txtName = new TextBox();
      txtName.setWidth("200");
      txtName.addKeyUpHandler(new KeyUpHandler() {
         @Override
         public void onKeyUp(KeyUpEvent event) {
            if(event.getNativeKeyCode() == KeyCodes.KEY_ENTER){
               Window.alert(getGreeting(txtName.getValue()));
            }
         }
      });
      Label lblName = new Label("Enter your name: ");

      Button buttonMessage = new Button("Click Me!");

      buttonMessage.addClickHandler(new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
         Window.alert(getGreeting(txtName.getValue()));
      }});

      HorizontalPanel hPanel = new HorizontalPanel();
      hPanel.add(lblName);
      hPanel.add(txtName);
      hPanel.setCellWidth(lblName, "130");
```

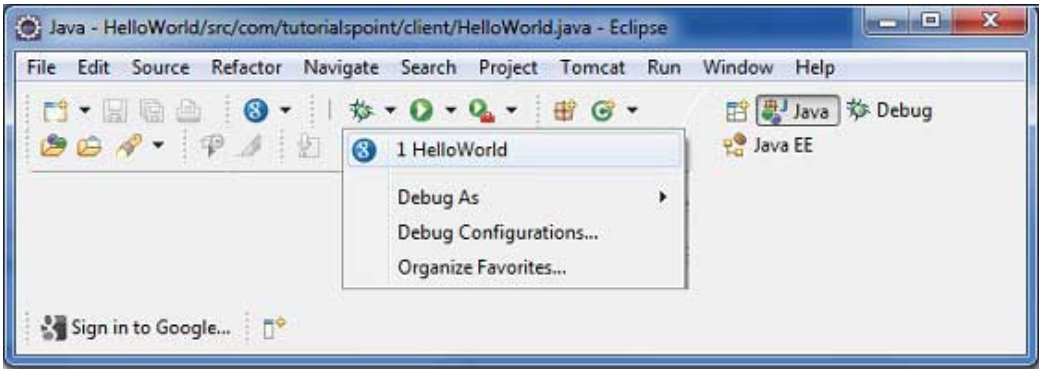
```
      VerticalPanel vPanel = new VerticalPanel();
      vPanel.setSpacing(10);
      vPanel.add(hPanel);
      vPanel.add(buttonMessage);
      vPanel.setCellHorizontalAlignment(buttonMessage,
      HasHorizontalAlignment.ALIGN_RIGHT);

      DecoratorPanel panel = new DecoratorPanel();
      panel.add(vPanel);

      // Add widgets to the root panel.
      RootPanel.get("gwtContainer").add(panel);
   }

   public String getGreeting(String name){
      return "Hello "+name+"!";
   }
}
```

Step 1 - Place BreakPoints

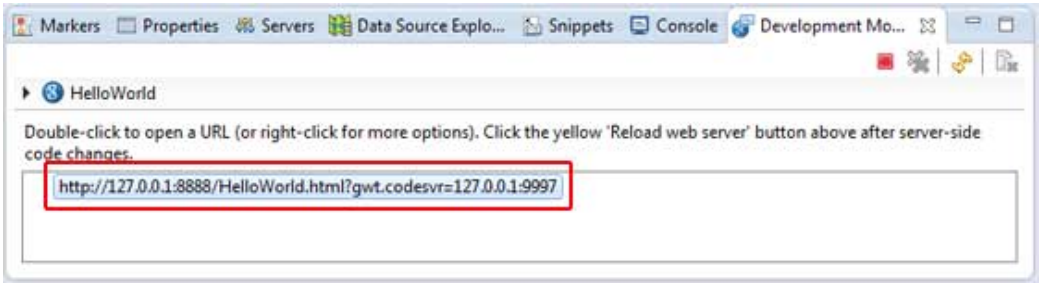
Place a breakpoint on the first line of onModuleLoad() of HelloWorld.java

Step 2 - Debug Application

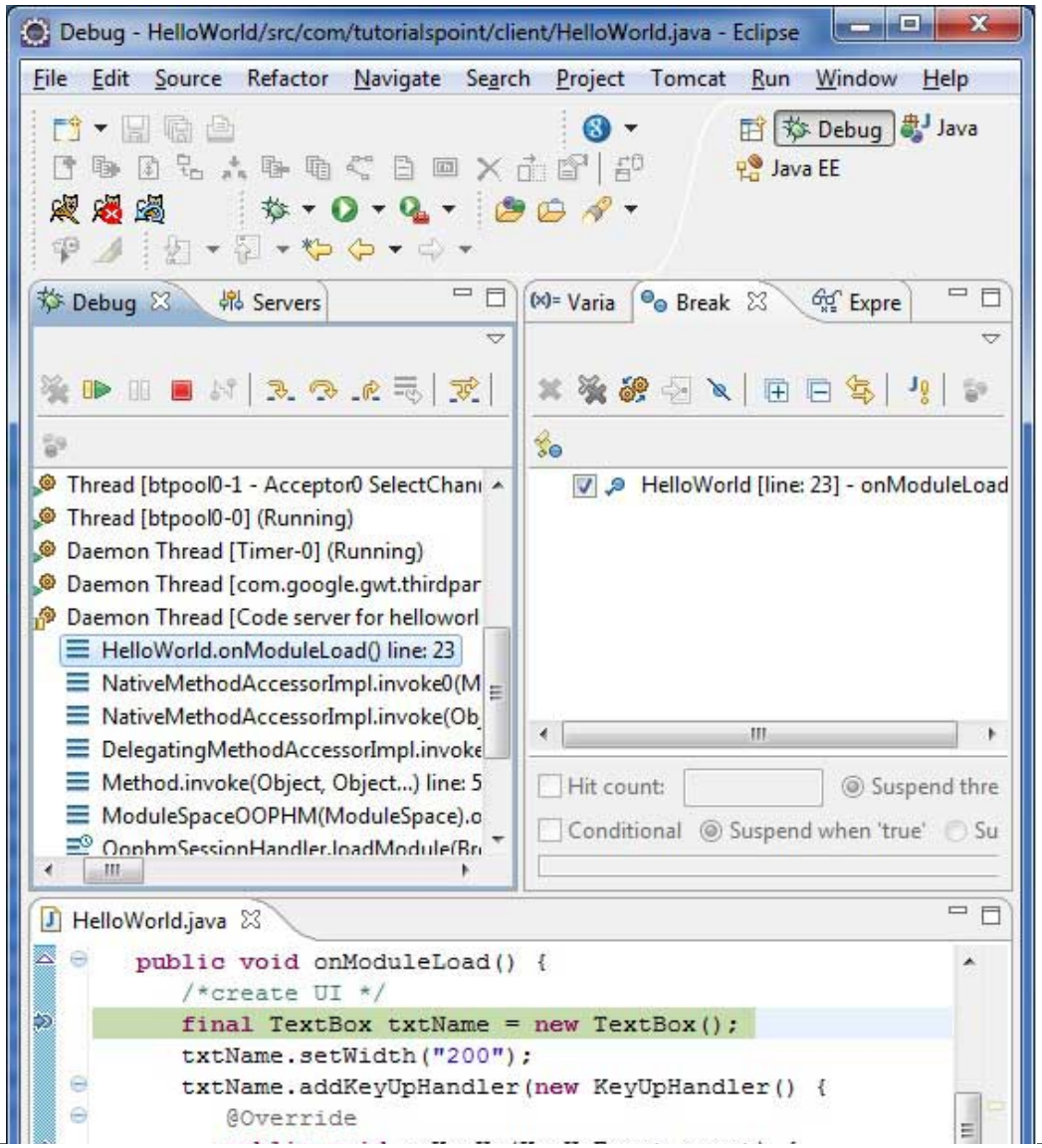
Now click on
Debug application menu and select **HelloWorld** application to debug the application.

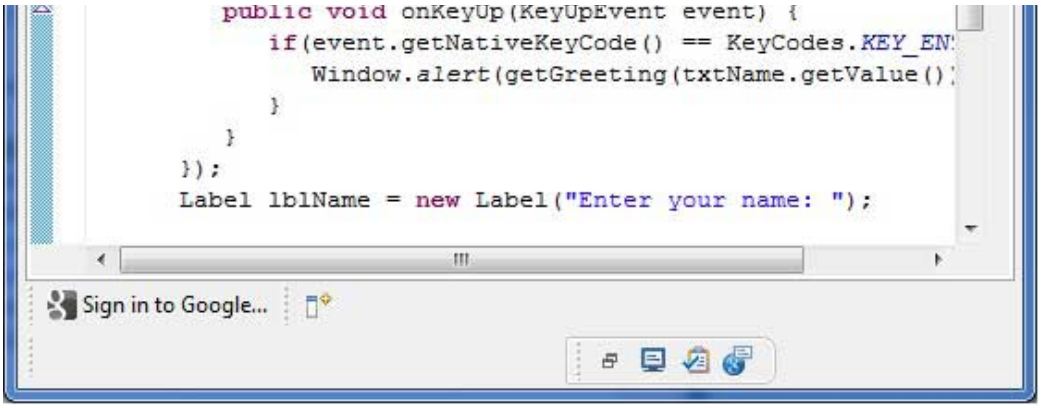


If everything is fine, you must see GWT Development Mode active in Eclipse containing a URL as shown below. Double click the URL to open the GWT application.

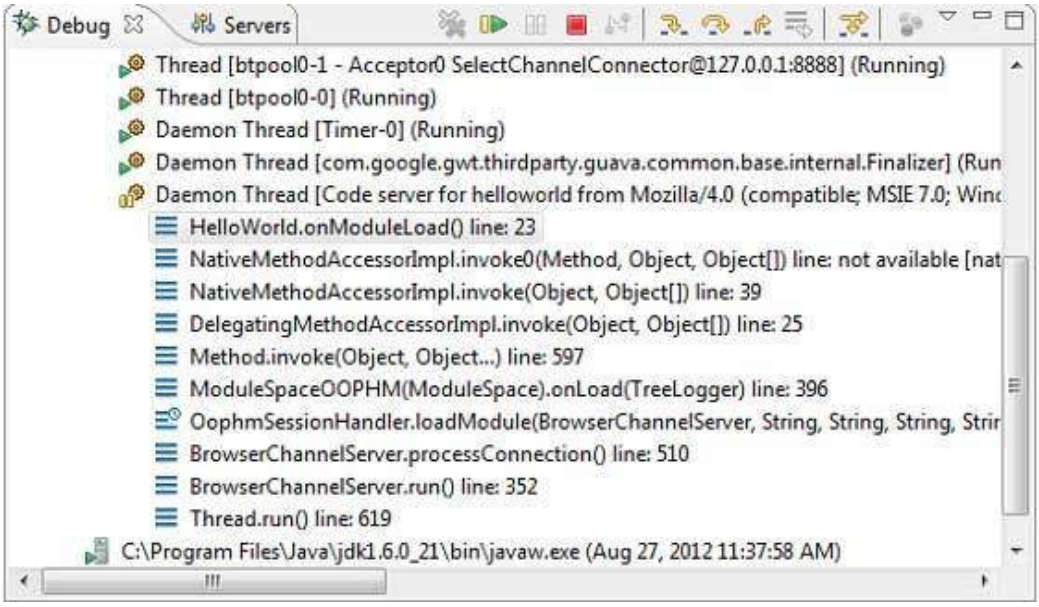


As soon as Application launches, you will see the focus on Eclipse breakpoint as we've placed the breakpoint on first line of entry point method.

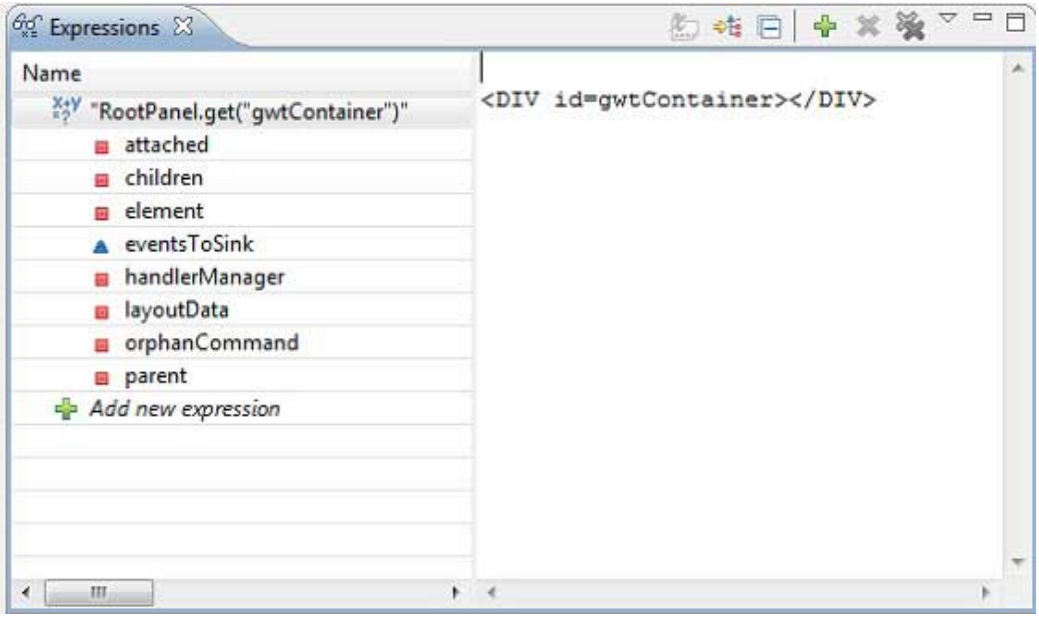




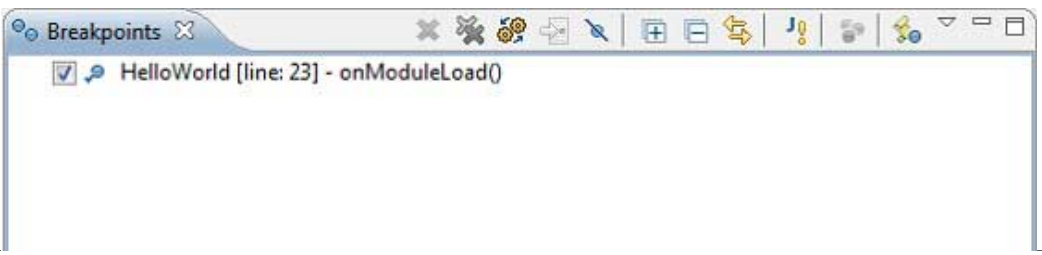
You can see the stacktrace for suspended threads.



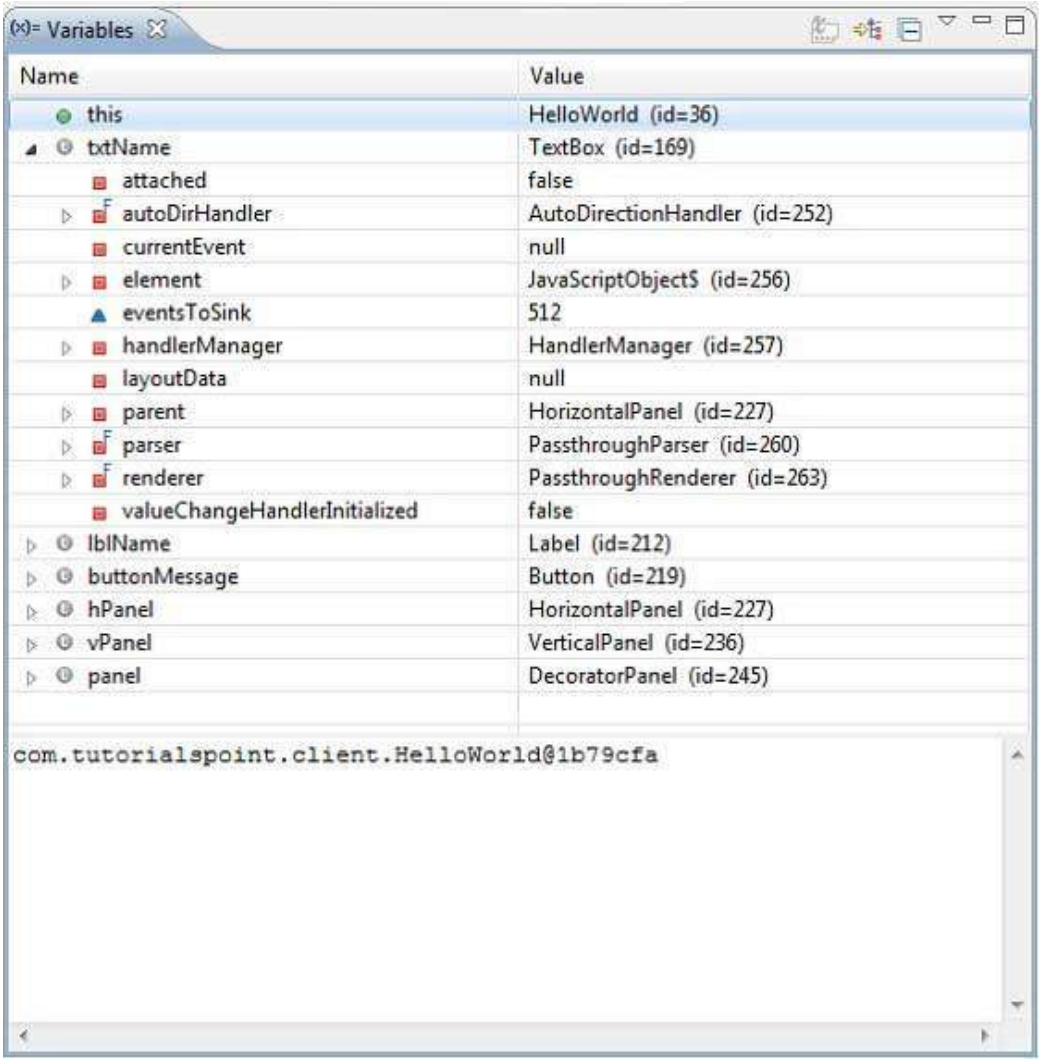
You can see the values for expressions.



You can see the list of breakpoints placed.



Now keep pressing F6 until you reach the last line of onModuleLoad() method. As reference for function keys, F6 inspects code line by line, F5 steps inside further and F8 will resume the application. Now you can see the list of values of all variables of onModuleLoad() method.



Now you can see the GWT client code can be debugged in the same way as a Java Application can be debugged. Place breakpoints to any line and play with debugging capabilities of GWT.

GWT - INTERNATIONALIZATION

GWT provides three ways to internationalize a GWT application, We'll demonstrate use of Static String Internationalization being most commonly used among projects.

Technique	Description
Static String Internationalization	This technique is most prevalent and requires very little overhead at runtime; is a very efficient technique for translating both constant and parameterized strings;simplest to implement. Static string internationalization uses standard Java properties files to store translated strings and parameterized messages, and strongly-typed Java interfaces are created to retrieve their values.

Dynamic String Internationalization	This technique is very flexible but slower than static string internationalization. Host page contains the localized strings therefore, applications are not required to be recompiled when we add a new locale. If GWT application is to be integrated with an existing server-side localization system, then this technique is to be used.
Localizable Interface	This technique is the most powerful among the three techniques. Implementing Localizable allows us to create localized versions of custom types. It's an advanced internationalization technique.

Workflow of internationalizing a GWT Application

Step 1: Create properties files

Create properties file containing the messages to be used in the application. We've created a **HelloWorldMessages.properties** file in our example.

```
enterName=Enter your name
clickMe=Click Me
applicationTitle=Application Internationalization Demonstration
greeting=Hello {0}
```

Create properties files containing translated values specific to locale. We've created a **HelloWorldMessages_de.properties** file in our example. This file contains translations in german language. _de specifies the german locale and we're going to support german language in our application.

If you are creating properties file using Eclipse then change the encoding of the file to UTF-8.Select the file and then right-click in it to open its properties window.Select Text file encoding as **Other UTF-8**. Apply and Save the change.

```
enterName=Geben Sie Ihren Namen
clickMe=Klick mich
applicationTitle=Anwendung Internationalisierung Demonstration
greeting=Hallo {0}
```

Step 2: Add i18n module to Module Descriptor XML File

Update module file **HelloWorld.gwt.xml** to include support for german locale

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
...
<extend-property name="locale" values="de" />
...
</module>
```

Step 3: Create Interface equivalent to properties file

Create HelloWorldMessages.java interface by extending Messages interface of GWT to include support for internalization. It should contain same method names as keys in properties file. Place holder would be replaced with String argument.

```
public interface HelloWorldMessages extends Messages {

   @DefaultMessage("Enter your name")
   String enterName();

   @DefaultMessage("Click Me")
   String clickMe();

   @DefaultMessage("Application Internalization Demonstration")
   String applicationTitle();

   @DefaultMessage("Hello {0}")
   String greeting(String name);
}
```

Step 4: Use Message Interface in UI component.

Use object of HelloWorldMessages in HelloWorld to get the messages.

```
public class HelloWorld implements EntryPoint {

   /* create an object of HelloWorldMessages interface
      using GWT.create() method */
   private HelloWorldMessages messages =
   GWT.create(HelloWorldMessages.class);

   public void onModuleLoad() {
   ...
      Label titleLabel = new Label(messages.applicationTitle());
      //Add title to the application
      RootPanel.get("gwtAppTitle").add(titleLabel);
 ...
   }
}
```

Internationalization - Complete Example

This example will take you through simple steps to demonstrate Internationalization capability of a GWT application. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>
  <extend-property name="locale" values="de" />
  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**.

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<h1 ></h1>
<div ></div>

</body>
</html>
```

Now create HelloWorldMessages.properties file in the **src/com.tutorialspoint/client** package and place the following contents in it

```
enterName=Enter your name
clickMe=Click Me
applicationTitle=Application Internationalization Demonstration
greeting=Hello {0}
```

Now create HelloWorldMessages_de.properties file in the **src/com.tutorialspoint/client** package and place the following contents in it

```
enterName=Geben Sie Ihren Namen
clickMe=Klick mich
applicationTitle=Anwendung Internationalisierung Demonstration
greeting=Hallo {0}
```

Now create HelloWorldMessages.java class in the **src/com.tutorialspoint/client** package and place the following contents in it

```
package com.tutorialspoint.client;
import com.google.gwt.i18n.client.Messages;

public interface HelloWorldMessages extends Messages {
   @DefaultMessage("Enter your name")
   String enterName();

   @DefaultMessage("Click Me")
   String clickMe();

   @DefaultMessage("Application Internationalization Demonstration")
   String applicationTitle();

   @DefaultMessage("Hello {0}")
   String greeting(String name);
}
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** using which we will demonstrate Internationalization capability of GWT Code.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DecoratorPanel;
import com.google.gwt.user.client.ui.HasHorizontalAlignment;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Label;
```

```
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

public class HelloWorld implements EntryPoint {

   /* create an object of HelloWorldMessages interface
      using GWT.create() method */
   private HelloWorldMessages messages =
   GWT.create(HelloWorldMessages.class);

   public void onModuleLoad() {
      /*create UI */
      final TextBox txtName = new TextBox();
      txtName.setWidth("200");
      txtName.addKeyUpHandler(new KeyUpHandler() {
         @Override
         public void onKeyUp(KeyUpEvent event) {
            if(event.getNativeKeyCode() == KeyCodes.KEY_ENTER){
               Window.alert(getGreeting(txtName.getValue()));
            }
         }
      });
      Label lblName = new Label(messages.enterName() + ": ");

      Button buttonMessage = new Button(messages.clickMe() + "!");

      buttonMessage.addClickHandler(new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
         Window.alert(getGreeting(txtName.getValue()));
      }});

      HorizontalPanel hPanel = new HorizontalPanel();
      hPanel.add(lblName);
      hPanel.add(txtName);

      VerticalPanel vPanel = new VerticalPanel();
      vPanel.setSpacing(10);
      vPanel.add(hPanel);
      vPanel.add(buttonMessage);
      vPanel.setCellHorizontalAlignment(buttonMessage,
      HasHorizontalAlignment.ALIGN_RIGHT);

      DecoratorPanel panel = new DecoratorPanel();
      panel.add(vPanel);
      Label titleLabel = new Label(messages.applicationTitle());
      //Add title to the application
      RootPanel.get("gwtAppTitle").add(titleLabel);
      // Add widgets to the root panel.
      RootPanel.get("gwtContainer").add(panel);
   }

   public String getGreeting(String name){
      return messages.greeting(name + "!");
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:

Now update the URL to contain the locale=de.Set URL: *http://127.0.0.1:8888/HelloWorld.html?gwt.codesvr=127.0.0.1:9997**&locale=de***. If everything is fine with your application, this will produce following result:

GWT - HISTORY CLASS

GWT applications are normally single page application running JavaScripts and do not contains lot of pages thus browser do not keep track of user interaction with Application. To use browser's history functionality, application should generate a unique URL fragment for each navigable page.

GWT provides **History Mechanism** to handle this situation.

GWT uses a term **token** which is simply a string that the application can parse to return to a particular state. Application will save this token in browser's history as URL fragment.

For example, a history token named "pageIndex1" would be added to a URL as follows:

```
http://www.tutorialspoint.com/HelloWorld.html#pageIndex0
```

History Management Workflow

Step 1: Enable History support

In order to use GWT History support, we must first embed following iframe into our host HTML page.

```
<iframe src="javascript:''"
```

```
   style="width:0;height:0;border:0"></iframe>
```

Step 2: Add token to History

Following example stats how to add token to browser history

```
int index = 0;
History.newItem("pageIndex" + index);
```

Step 3: Retrive token from History

When user uses back/forward button of browser, we'll retrive the token and update our application state accordingly.

```
History.addValueChangeHandler(new ValueChangeHandler<String>() {
   @Override
   public void onValueChange(ValueChangeEvent<String> event) {
      String historyToken = event.getValue();
      /* parse the history token */
      try {
         if (historyToken.substring(0, 9).equals("pageIndex")) {
            String tabIndexToken = historyToken.substring(9, 10);
            int tabIndex = Integer.parseInt(tabIndexToken);
            /* select the specified tab panel */
            tabPanel.selectTab(tabIndex);
         } else {
            tabPanel.selectTab(0);
         }
      } catch (IndexOutOfBoundsException e) {
         tabPanel.selectTab(0);
      }
   }
});
```

Now let's see the History Class in Action.

History Class - Complete Example

This example will take you through simple steps to demonstrate History Management of a GWT application. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>
  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>
```

```
</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**

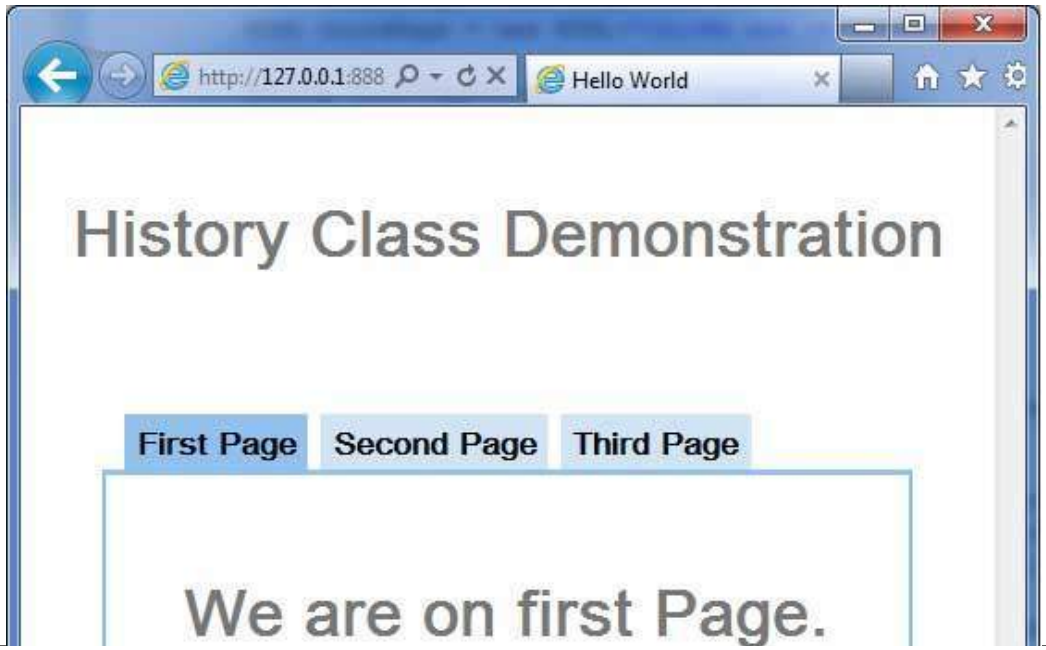
```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<iframe src="javascript:''"

   style="width:0;height:0;border:0"></iframe>
<h1> History Class Demonstration</h1>
<div ></div>

</body>
</html>
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** using which we will demonstrate History Management in GWT Code.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.logical.shared.SelectionEvent;
import com.google.gwt.event.logical.shared.SelectionHandler;
import com.google.gwt.event.logical.shared.ValueChangeEvent;
import com.google.gwt.event.logical.shared.ValueChangeHandler;
import com.google.gwt.user.client.History;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TabPanel;

public class HelloWorld implements EntryPoint {

   /**
   * This is the entry point method.
   */
   public void onModuleLoad() {
      /* create a tab panel to carry multiple pages */
      final TabPanel tabPanel = new TabPanel();

      /* create pages */
      HTML firstPage = new HTML("<h1>We are on first Page.</h1>");
      HTML secondPage = new HTML("<h1>We are on second Page.</h1>");
      HTML thirdPage = new HTML("<h1>We are on third Page.</h1>");

      String firstPageTitle = "First Page";
      String secondPageTitle = "Second Page";
      String thirdPageTitle = "Third Page";
      tabPanel.setWidth("400");

   /* add pages to tabPanel*/
      tabPanel.add(firstPage, firstPageTitle);
```

```
      tabPanel.add(secondPage,secondPageTitle);
      tabPanel.add(thirdPage, thirdPageTitle);

      /* add tab selection handler */
      tabPanel.addSelectionHandler(new SelectionHandler<Integer>() {
         @Override
         public void onSelection(SelectionEvent<Integer> event) {
            /* add a token to history containing pageIndex
             History class will change the URL of application
             by appending the token to it.
            */
            History.newItem("pageIndex" + event.getSelectedItem());
         }
      });

      /* add value change handler to History
       this method will be called, when browser's
       Back button or Forward button are clicked
       and URL of application changes.
       */
      History.addValueChangeHandler(new ValueChangeHandler<String>() {
         @Override
         public void onValueChange(ValueChangeEvent<String> event) {
            String historyToken = event.getValue();
            /* parse the history token */
            try {
               if (historyToken.substring(0, 9).equals("pageIndex")) {
                  String tabIndexToken = historyToken.substring(9, 10);
                  int tabIndex = Integer.parseInt(tabIndexToken);
                  /* select the specified tab panel */
                  tabPanel.selectTab(tabIndex);
               } else {
                  tabPanel.selectTab(0);
               }
            } catch (IndexOutOfBoundsException e) {
               tabPanel.selectTab(0);
            }
         }
      });

      /* select the first tab by default */
      tabPanel.selectTab(0);

      /* add controls to RootPanel */
      RootPanel.get().add(tabPanel);
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:



- Now click on each tab to select different pages.
- You should notice, when each tab is selected ,application url is changed and #pageIndex is added to the url.
- You can also see that browser's back and forward buttons are enabled now.
- Use back and forward button of the browser and you will see the different tabs get selected accordingly.

GWT - BOOKMARK SUPPORT

GWT supports browser history management using a History class for which you can reference *GWT - History Class* chapter.

GWT uses a term **token** which is simply a string that the application can parse to return to a particular state. Application will save this token in browser's history as URL fragment.

In *GWT - History Class* chapter, we handle the token creation and setting in the history by writing code.

In this article, we will discuss a special widget Hyperlink which does the token creation and history management for us automatically and gives application capability of bookmarking.

Bookmarking Example

This example will take you through simple steps to demonstrate Bookmarking of a GWT application. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>

  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>
  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>

</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**

```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<iframe src="javascript:''"

   style="width:0;height:0;border:0"></iframe>
<h1> Bookmarking Demonstration</h1>
<div ></div>

</body>
</html>
```

Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** using which we will demonstrate Bookmarking in GWT Code.

```
package com.tutorialspoint.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.logical.shared.ValueChangeEvent;
import com.google.gwt.event.logical.shared.ValueChangeHandler;
import com.google.gwt.user.client.History;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Hyperlink;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TabPanel;
import com.google.gwt.user.client.ui.VerticalPanel;

public class HelloWorld implements EntryPoint {

   private TabPanel tabPanel;

   private void selectTab(String historyToken){
      /*  parse the history token */
      try {
         if (historyToken.substring(0, 9).equals("pageIndex")) {
            String tabIndexToken = historyToken.substring(9, 10);
            int tabIndex = Integer.parseInt(tabIndexToken);
            /* Select the specified tab panel */
            tabPanel.selectTab(tabIndex);
         } else {
            tabPanel.selectTab(0);
         }
      } catch (IndexOutOfBoundsException e) {
         tabPanel.selectTab(0);
      }
   }

   /**
   * This is the entry point method.
   */
   public void onModuleLoad() {
      /* create a tab panel to carry multiple pages */
```

```
      tabPanel = new TabPanel();

      /* create pages */
      HTML firstPage = new HTML("<h1>We are on first Page.</h1>");
      HTML secondPage = new HTML("<h1>We are on second Page.</h1>");
      HTML thirdPage = new HTML("<h1>We are on third Page.</h1>");

      String firstPageTitle = "First Page";
      String secondPageTitle = "Second Page";
      String thirdPageTitle = "Third Page";

      Hyperlink firstPageLink = new Hyperlink("1", "pageIndex0");
      Hyperlink secondPageLink = new Hyperlink("2", "pageIndex1");
      Hyperlink thirdPageLink = new Hyperlink("3", "pageIndex2");

      HorizontalPanel linksHPanel = new HorizontalPanel();
      linksHPanel.setSpacing(10);
      linksHPanel.add(firstPageLink);
      linksHPanel.add(secondPageLink);
      linksHPanel.add(thirdPageLink);

      /* If the application starts with no history token,
         redirect to a pageIndex0 */
      String initToken = History.getToken();

      if (initToken.length() == 0) {
         History.newItem("pageIndex0");
         initToken = "pageIndex0";
      }

      tabPanel.setWidth("400");
      /* add pages to tabPanel*/
      tabPanel.add(firstPage, firstPageTitle);
      tabPanel.add(secondPage,secondPageTitle);
      tabPanel.add(thirdPage, thirdPageTitle);

      /* add value change handler to History
      * this method will be called, when browser's Back button
      * or Forward button are clicked.
      * and URL of application changes.
      * */
      History.addValueChangeHandler(new ValueChangeHandler<String>() {
         @Override
         public void onValueChange(ValueChangeEvent<String> event) {
            selectTab(event.getValue());
         }
      });

      selectTab(initToken);

      VerticalPanel vPanel = new VerticalPanel();

      vPanel.setSpacing(10);
      vPanel.add(tabPanel);
      vPanel.add(linksHPanel);

      /* add controls to RootPanel */
      RootPanel.get().add(vPanel);
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:

- Now click on 1, 2 or 3. You can notice that the tab changes with indexes.
- You should notice, when you click on 1,2 or 3 ,application url is changed and #pageIndex is added to the url
- You can also see that browser's back and forward buttons are enabled now.
- Use back and forward button of the browser and you will see the different tabs get selected accordingly.

- Right Click on 1, 2 or 3. You can see options like open, open in new window, open in new tab, add to favourites etc.
- ight Click on 3. Choose add to favourites. Save bookmark as page 3.
- Open favourites and choose page 3. You will see the third tab selected.

GWT - LOGGING FRAMEWORK

The logging framework emulates java.util.logging, so it uses the same syntax and has the same behavior as server side logging code

GWT logging is configured using .gwt.xml files.

We can configure logging to be enabled/disabled; we can enable/disable particular handlers, and change the default logging level.

Types of Logger

- Loggers are organized in a tree structure, with the Root Logger at the root of the tree.
- Name of the logger determine the Parent/Child relationships using **.** to separate sections of the name.
- As an example if we have two loggers Hospital.room1 and Hospital.room2, then they are siblings, with their parent being the logger named Hospital. The Hospital logger (and any logger with a name which does not contain a dot ".") has the Root Logger as a parent.

```
private static Logger room1Logger = Logger.getLogger("Hospital.room1");
private static Logger room2Logger = Logger.getLogger("Hospital.room2");
private static Logger hospitalLogger = Logger.getLogger("Hospital");
private static Logger rootLogger = Logger.getLogger("");
```

Log Handlers

GWT provides default handlers which will show the log entries made using loggers.

Handler	Logs to	Description
SystemLogHandler	stdout	These messages can only be seen in Development Mode in the DevMode window.
DevelopmentModeLogHandler	DevMode Window	Logs by calling method GWT.log. These messages can only be seen in Development Mode in the DevMode window.
ConsoleLogHandler	javascript console	Logs to the javascript console, which is used by Firebug Lite (for IE), Safari and Chrome.
FirebugLogHandler	Firebug	Logs to the firebug console.
PopupLogHandler	popup	Logs to the popup which resides in the upper left hand corner of application when this handler is enabled.
SimpleRemoteLogHandler	server	This handler sends log messages to the server, where they will be logged using the server side logging mechanism.

Configure Logging in GWT Application

.gwt.xml file is to be configured to enable GWT logging as follows:

HelloWorld.gwt.xml

```
# add logging module
<inherits name="com.google.gwt.logging.Logging"/>
# To change the default logLevel
```

```
   <set-property name="gwt.logging.logLevel" value="SEVERE"/>
   # To enable logging
   <set-property name="gwt.logging.enabled" value="TRUE"/>
   # To disable a popup Handler
   <set-property name="gwt.logging.popupHandler" value="DISABLED" />
```

Use logger to log user actions

```
    /* Create Root Logger */
    private static Logger rootLogger = Logger.getLogger("");
 ...
    rootLogger.log(Level.SEVERE, "pageIndex selected: "
    + event.getValue());
    ...
```

Logging Framework Example

This example will take you through simple steps to demonstrate Logging Capability of a GWT application. Follow the following steps to update the GWT application we created in *GWT - Create Application* chapter:

Step	Description
1	Create a project with a name *HelloWorld* under a package *com.tutorialspoint* as explained in the *GWT - Create Application* chapter.
2	Modify *HelloWorld.gwt.xml*, *HelloWorld.css*, *HelloWorld.html* and *HelloWorld.java* as explained below. Keep rest of the files unchanged.
3	Compile and run the application to verify the result of the implemented logic.

Following is the content of the modified module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>
  <inherits name="com.google.gwt.logging.Logging"/>
  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>
  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>
  <set-property name="gwt.logging.logLevel" value="SEVERE"/>
  <set-property name="gwt.logging.enabled" value="TRUE"/>
  <set-property name="gwt.logging.popupHandler" value="DISABLED" />
</module>
```

Following is the content of the modified Style Sheet file **war/HelloWorld.css**.

```
body{
   text-align: center;
   font-family: verdana, sans-serif;
}
h1{
   font-size: 2em;
   font-weight: bold;
   color: #777777;
   margin: 40px 0px 70px;
   text-align: center;
}
```

Following is the content of the modified HTML host file **war/HelloWorld.html**

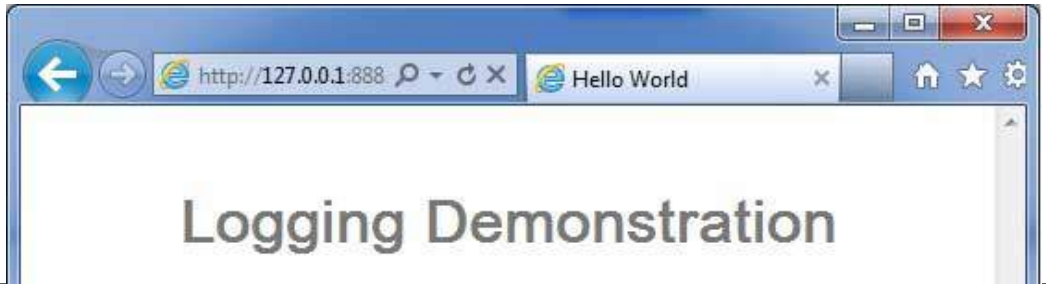
```
<html>
<head>
<title>Hello World</title>
   <link rel="stylesheet" href="HelloWorld.css"/>
   <script language="javascript" src="helloworld/helloworld.nocache.js">
   </script>
</head>
<body>

<iframe src="javascript:''"

   style="width:0;height:0;border:0"></iframe>
<h1> Logging Demonstration</h1>
<div ></div>

</body>
</html>
```

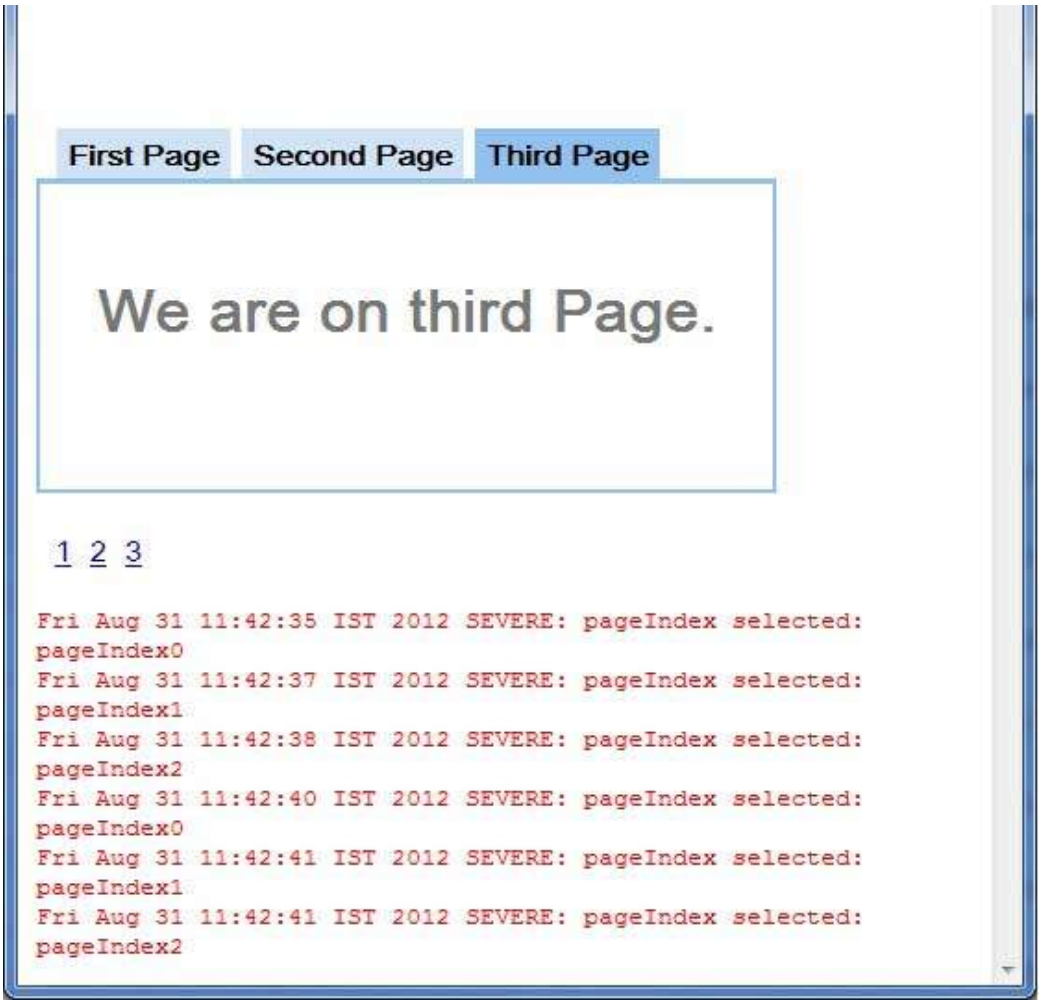
Let us have following content of Java file **src/com.tutorialspoint/HelloWorld.java** using which we will demonstrate Bookmarking in GWT Code.

```
package com.tutorialspoint.client;

import java.util.logging.Level;
import java.util.logging.Logger;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.logical.shared.ValueChangeEvent;
import com.google.gwt.event.logical.shared.ValueChangeHandler;
import com.google.gwt.logging.client.HasWidgetsLogHandler;
import com.google.gwt.user.client.History;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.HorizontalPanel;
import com.google.gwt.user.client.ui.Hyperlink;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TabPanel;
import com.google.gwt.user.client.ui.VerticalPanel;

public class HelloWorld implements EntryPoint {

   private TabPanel tabPanel;
   /* Create Root Logger */
   private static Logger rootLogger = Logger.getLogger("");
   private VerticalPanel customLogArea;

   private void selectTab(String historyToken){
      /*  parse the history token */
      try {
         if (historyToken.substring(0, 9).equals("pageIndex")) {
            String tabIndexToken = historyToken.substring(9, 10);
            int tabIndex = Integer.parseInt(tabIndexToken);
            /* Select the specified tab panel */
            tabPanel.selectTab(tabIndex);
         } else {
            tabPanel.selectTab(0);
         }
      } catch (IndexOutOfBoundsException e) {
         tabPanel.selectTab(0);
      }
   }

   /**
   * This is the entry point method.
   */
   public void onModuleLoad() {
      /* create a tab panel to carry multiple pages */
      tabPanel = new TabPanel();

      /* create pages */
      HTML firstPage = new HTML("<h1>We are on first Page.</h1>");
      HTML secondPage = new HTML("<h1>We are on second Page.</h1>");
      HTML thirdPage = new HTML("<h1>We are on third Page.</h1>");

      String firstPageTitle = "First Page";
```

```
      String secondPageTitle = "Second Page";
      String thirdPageTitle = "Third Page";

      Hyperlink firstPageLink = new Hyperlink("1", "pageIndex0");
      Hyperlink secondPageLink = new Hyperlink("2", "pageIndex1");
      Hyperlink thirdPageLink = new Hyperlink("3", "pageIndex2");

      HorizontalPanel linksHPanel = new HorizontalPanel();
      linksHPanel.setSpacing(10);
      linksHPanel.add(firstPageLink);
      linksHPanel.add(secondPageLink);
      linksHPanel.add(thirdPageLink);

      /* If the application starts with no history token,
         redirect to a pageIndex0 */
      String initToken = History.getToken();

      if (initToken.length() == 0) {
         History.newItem("pageIndex0");
         initToken = "pageIndex0";
      }

      tabPanel.setWidth("400");
      /* add pages to tabPanel*/
      tabPanel.add(firstPage, firstPageTitle);
      tabPanel.add(secondPage,secondPageTitle);
      tabPanel.add(thirdPage, thirdPageTitle);

      /* add value change handler to History
      * this method will be called, when browser's Back button
      * or Forward button are clicked.
      * and URL of application changes.
      * */
      History.addValueChangeHandler(new ValueChangeHandler<String>() {
         @Override
         public void onValueChange(ValueChangeEvent<String> event) {
            selectTab(event.getValue());
            rootLogger.log(Level.SEVERE, "pageIndex selected: "
            + event.getValue());
         }
      });

      selectTab(initToken);

      VerticalPanel vPanel = new VerticalPanel();

      vPanel.setSpacing(10);
      vPanel.add(tabPanel);
      vPanel.add(linksHPanel);

      customLogArea = new VerticalPanel();
      vPanel.add(customLogArea);

      /* an example of using own custom logging area. */
      rootLogger.addHandler(new HasWidgetsLogHandler(customLogArea));

      /* add controls to RootPanel */
      RootPanel.get().add(vPanel);
   }
}
```

Once you are ready with all the changes done, let us compile and run the application in development mode as we did in GWT - Create Application chapter. If everything is fine with your application, this will produce following result:





- Now click on 1, 2 or 3. You can notice, when you click on 1,2 or 3 ,you can see the log is getting printed displaying the pageIndex.

Check the Console output in Eclipse. You can see the log is getting printed in Eclipse console as well.

```
Fri Aug 31 11:42:35 IST 2012
SEVERE: pageIndex selected: pageIndex0
Fri Aug 31 11:42:37 IST 2012
SEVERE: pageIndex selected: pageIndex1
Fri Aug 31 11:42:38 IST 2012
SEVERE: pageIndex selected: pageIndex2
Fri Aug 31 11:42:40 IST 2012
SEVERE: pageIndex selected: pageIndex0
Fri Aug 31 11:42:41 IST 2012
SEVERE: pageIndex selected: pageIndex1
Fri Aug 31 11:42:41 IST 2012
SEVERE: pageIndex selected: pageIndex2
```

Now update module descriptor **src/com.tutorialspoint/HelloWorld.gwt.xml** to enable popupHandler.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='helloworld'>
  <!-- Inherit the core Web Toolkit stuff.                        -->
  <inherits name='com.google.gwt.user.User'/>

  <!-- Inherit the default GWT style sheet.                       -->
  <inherits name='com.google.gwt.user.theme.clean.Clean'/>
  <inherits name="com.google.gwt.logging.Logging"/>
  <!-- Specify the app entry point class.                         -->
  <entry-point class='com.tutorialspoint.client.HelloWorld'/>
  <!-- Specify the paths for translatable code                    -->
  <source path='client'/>
  <source path='shared'/>
  <set-property name="gwt.logging.logLevel" value="SEVERE"/>
  <set-property name="gwt.logging.enabled" value="TRUE"/>
```

```
  <set-property name="gwt.logging.popupHandler" value="ENABLED" />
</module>
```

Once you are ready with all the changes done, reload the application by refreshing the browser window (press F5/reload button of the browser).

Notice a popup window is present now in upper left corner of the application.

Now click on 1, 2 or 3. You can notice, when you click on 1,2 or 3 ,you can see the log is getting printed displaying the pageIndex in the popup window.