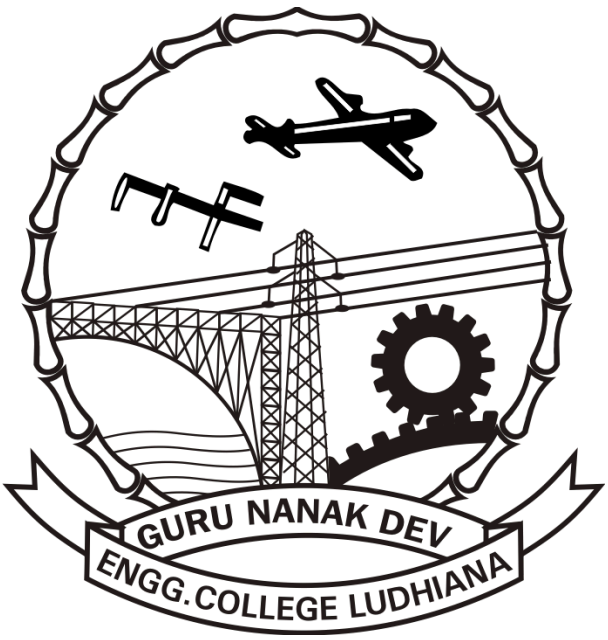
# INSTRUCTION MANUAL

# Free/ Open Source Software Lab

# (BTCS-605)




**GURU NANAK DEV ENGINEERING COLLEGE,**
**LUDHIANA (141006)**

# INDEX


| Sr. No. | Title of Programming Assignment | Page No. |
|---|---|---|
| 1. | Design a Website in HTML | 1 |
| 2. | To study Eclipse tool | 2 |
| 3. | Program to determine sum of following harmonic series for a given number n. $1 + \frac{1}{2} + \frac{1}{3} + \cdots\cdots + \frac{1}{n}$ | 16 |
| 4. | Program to print Floyd's triangle | 17 |
| 5. | Find Volume of Sphere, Cone and Cylinder (using switch...case..default). | 18 |
| 6. | Program to reverse the digits of a number using while loop. | 19 |
| 7. | Program to print following output using for loop .<br>1<br>2 2<br>3 3 3 | 20 |
| 8. | Program to Make use of Inheritance. | 21 |
| 9. | Program to implement multiple inheritance. | 22 |
| 10. | Program to implement runtime polymorphism. | 23 |
| 11. | Javascript code to greet user with a message welcome when html page completes loading.When a user leaves this page the ggdbye alert dialog box is displayed . | 24 |
| 12. | Javascript code to add two integers . | 25 |
| 13. | Write a function distance() which calculates the distance between two points (x1,y1) and (x2,y2).Incorporate this function into script that enables the user to enter the coordinates of the point to an html form. | 26 |

| Sr. No. | Title of Programming Assignment | Page No. |
|---|---|---|
| 14. | Write code which makes use of form elements like Text Field, Text Area, radio button, check box etc. | 27 |
| 15. | Write a script to change the contents of P element. | 29 |
| 16. | Write a script to change the value of the src attribute of an <img> element: | 30 |
| 17. | Write a script to change the content of H1 element when user clicks on it | 31 |
| 18. | Creating New HTML Elements (Nodes) | 32 |
| 19 | Removing Existing HTML Elements | 33 |
| 20 | Write a script to use onmouseover and onmouseout events | 34 |
| 21 | Writing XML web Documents which make use of XML Declaration, Element Declaration, Attribute Declaration | 35 |
| 22 | Usage of Internal DTD, External DTD. | 37 |
| 23 | Write a PHP program to add two numbers | 43 |
| 24 | Write a PHP program to print n numbers | 44 |
| 25 | Write a PHP program to store and restore session variables | 45 |

# Practical 1

**Title** **Design a Website in HTML .**

**Pre-requisite** Knowledge of

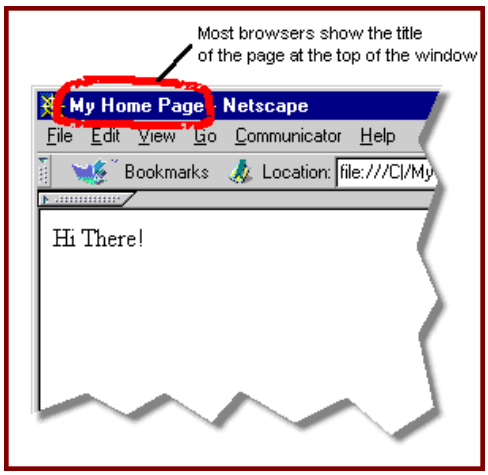
1. HTML.
2. Web Browsers.

**Algorithm**

<HTML>
<HEAD>
**<TITLE>**My Home Page**</TITLE>**
</HEAD>

<BODY>
**<H1>**My Home Page**</H1>**
Hi There!
</BODY>
</HTML>

**Sample Output**



# Practical 2

**Title** **To study Eclipse tool .**

**Objective** To get knowledge how to run java programs on eclipse.

**The Eclipse IDE**

Most people know Eclipse as an integrated development environment (IDE) for Java. The Eclipse IDE is definitely the most known product of the Eclipse Open Source project. Today it is the leading development environment for Java with a market share of approximately 65%.

The Eclipse IDE can be extended with additional software components. Eclipse calls these software components *plug-ins*. Several Open Source projects and companies have extended the Eclipse IDE or created standalone applications (Eclipse RCP) on top of the Eclipse framework.

# Java requirements of eclipse

Eclipse requires an installed Java runtime. Eclipse 4.4 requires at least Java 6 to run.

For the examples in this book, you should use Java in version 8.

Java can be downloaded in two flavors: a *JRE* (Java Runtime Environment) and a *JDK* (Java Development Kit) version.

The Eclipse IDE contains its own Java compiler hence a JRE is sufficient for most tasks with Eclipse.

The *JDK* version of Java is required if you compile Java source code on the command line and for advanced development scenarios, for example, if you use automatic builds or if you develop Java web applications.

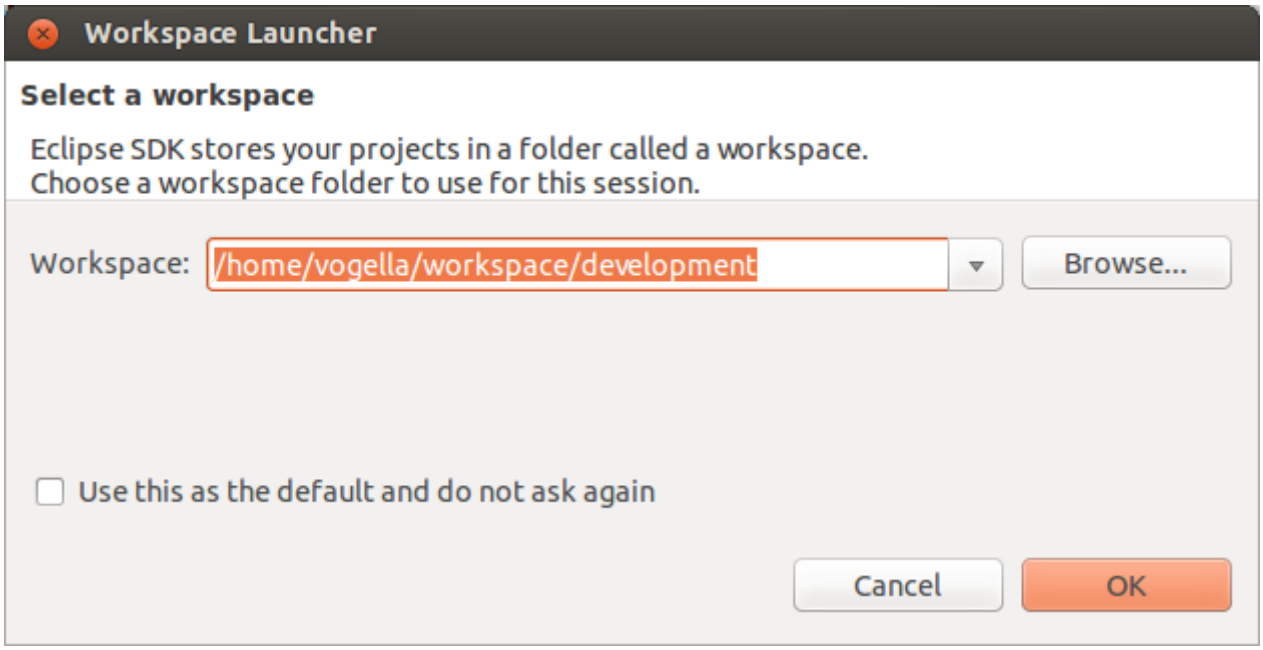
# Getting started

**Starting Eclipse**

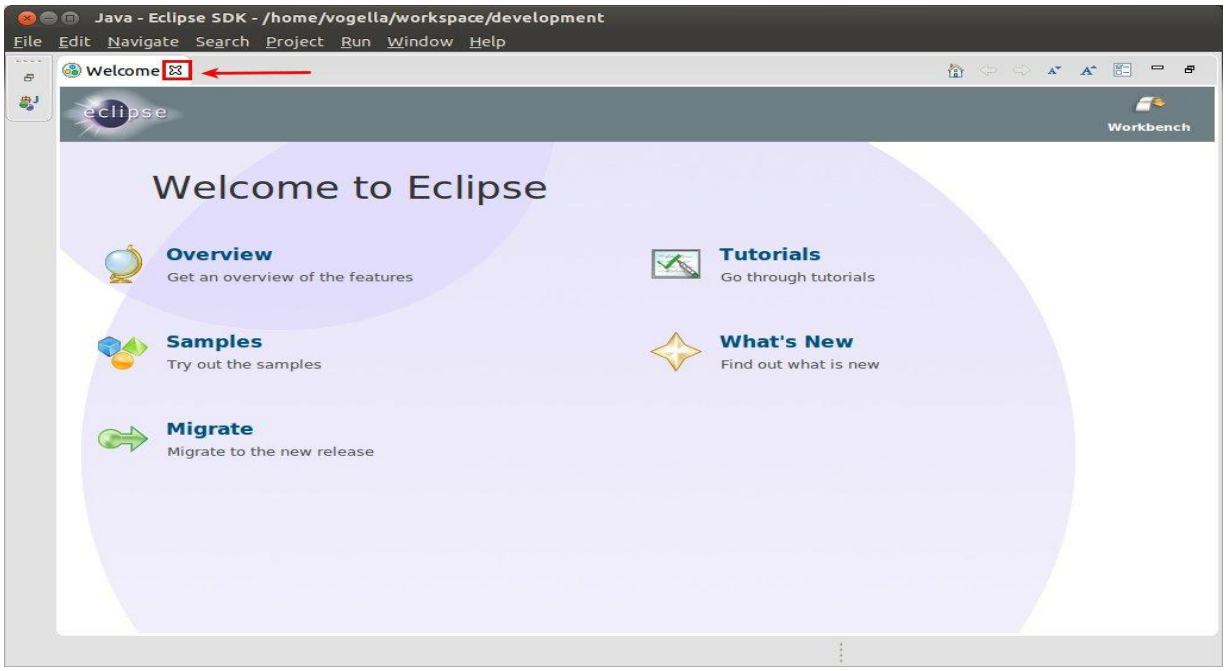
To start Eclipse, double-click on the eclipse.exe (Microsoft Windows) or eclipse (Linux / Mac) file in the directory where you unpacked Eclipse.

The system will prompt you for a *workspace*. The *workspace* is the location in your file system in which Eclipse stores its configuration and potentially other resources, like projects.

Select an empty directory and click the OK button.




Eclipse starts and shows the Welcome page. Close this page by clicking the x beside Welcome.



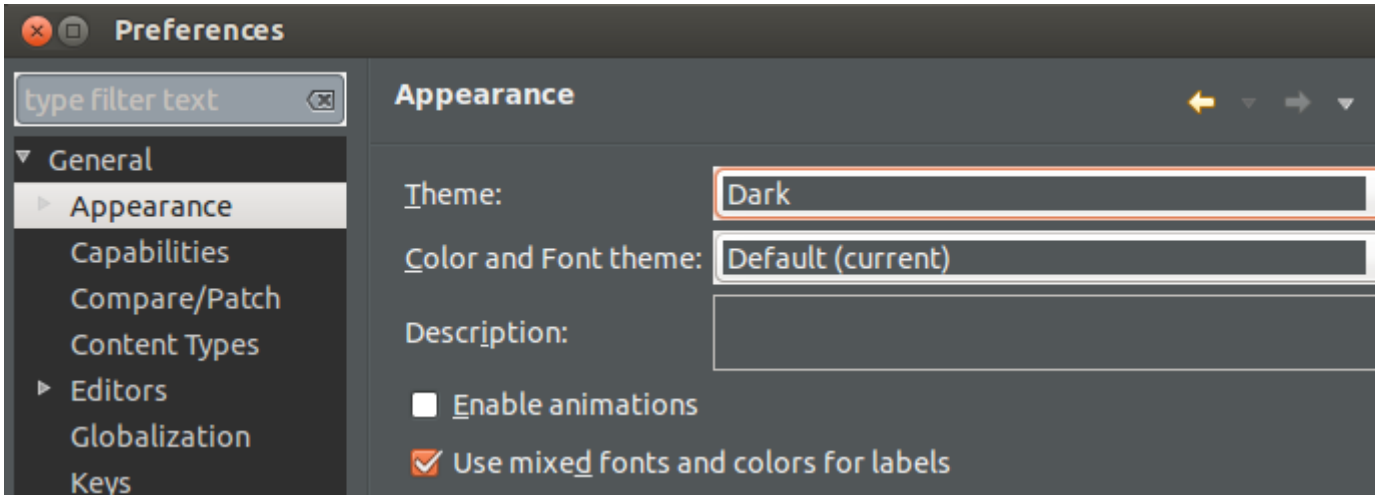

After closing the welcome screen, the application should look similar to the following screenshot.

## Appearance

The appearance of Eclipse can be configured. By default, Eclipse ships with a few themes but you can also extend Eclipse with new themes.

To change the appearance, select from the menu Window → Preferences → General → Appearance.

The Theme selection allows you to change the appearance of your Eclipse IDE. For example you can switch to the *Dark* theme of Eclipse.




# Important Eclipse terminology

Eclipse provides *Perspectives*, *Views* and *Editors*. *Views* and *Editors* are grouped into *Perspectives*.

**Workspace**
The *workspace* is the physical location (file path) you are working in. Your projects, source files, images and other artifacts can be stored and saved in your workspace. The *workspace* also contains preferences settings, plug-in specific meta data, logs etc.

You typically use different *workspaces* if you require different settings for your project or if you want to divide your projects into separate directories.

You can choose the workspace during startup of Eclipse or via the menu (File → Switch Workspace → Others) .

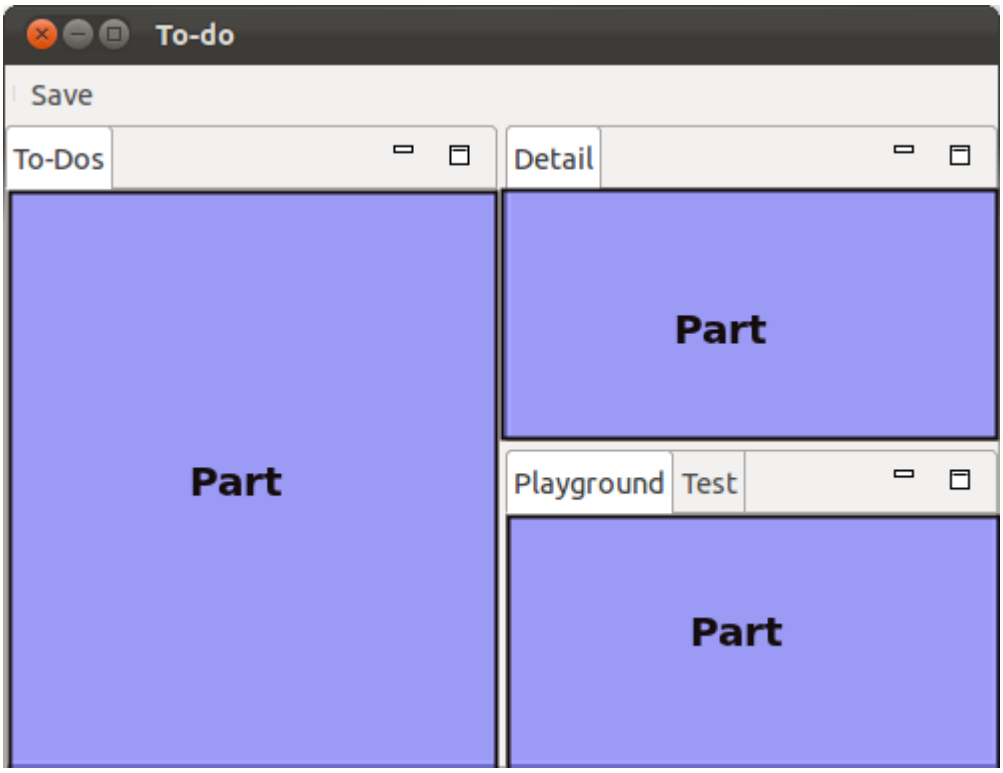
**Eclipse projects**
An Eclipse project contains source, configuration and binary files related to a certain task and groups them into buildable and reusable units. An Eclipse project can have *natures* assigned to it which describe the purpose of this project. For example, the Java *nature* defines a project as Java project. Projects can have multiple natures combined to model different technical aspects.

*Natures* for a project are defined via the .project file in the project directory.

Projects in Eclipse cannot contain other projects. Views and editors - parts

*Parts* are user interface components which allow you to navigate and modify data. A part can have a dropdown menu, context menus and a toolbar.

Parts can be freely positioned in the user interface.




Parts are typically classified into *views* and *editors*. The distinction into views and editors is not based on technical differences, but on a different concept of using and arranging these parts.

A view is typically used to work on a set of data, which might be a hierarchical structure. If data is changed via the view, this change is typically directly applied to the underlying data structure. A view sometimes allows us to open an editor for a selected set of data.

An example for a view is the *Package Explorer*, which allows you to browse the files of Eclipse projects. If you change data in the *Package Explorer*, e.g., renaming a file, the file name is directly changed on the file system.

Editors are typically used to modify a single data element, e.g., the content of a file or a data object. To apply the changes made in an editor to the data structure, the user has to explicitly save the editor content.

For example, the Java editor is used to modify Java source files. Changes to the source file are applied once the user selects the *Save* command. A dirty editor tab is marked with an asterisk to the left of the modified name of the file.

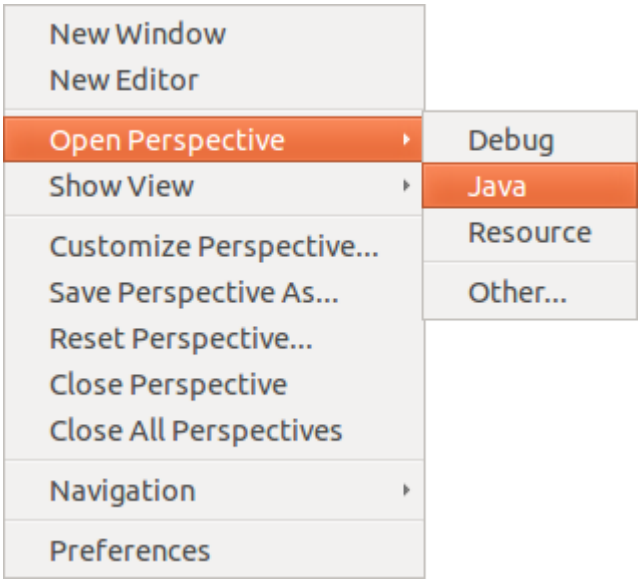

**Perspective**

A *perspective* is a visual container for a set of parts. Perspectives can be used to store different arrangements of parts. For example, the Eclipse IDE uses them to layout the views appropriate to the task (development, debugging, review, ...) the developer wants to perform.

Open editors are typically shared between perspectives, i.e., if you have an editor open in the Java perspective for a certain class and switch to the Debug perspective, this *editor* stays open.

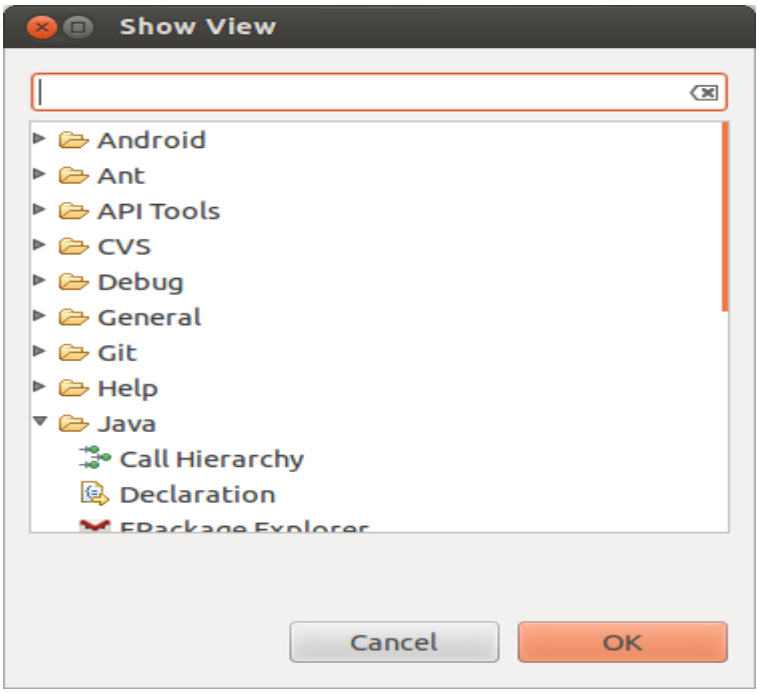
You can switch *Perspectives* via the Window → Open Perspective → Other... menu entry.

The main perspectives used for Java development are the Java perspective and the Debug perspective .



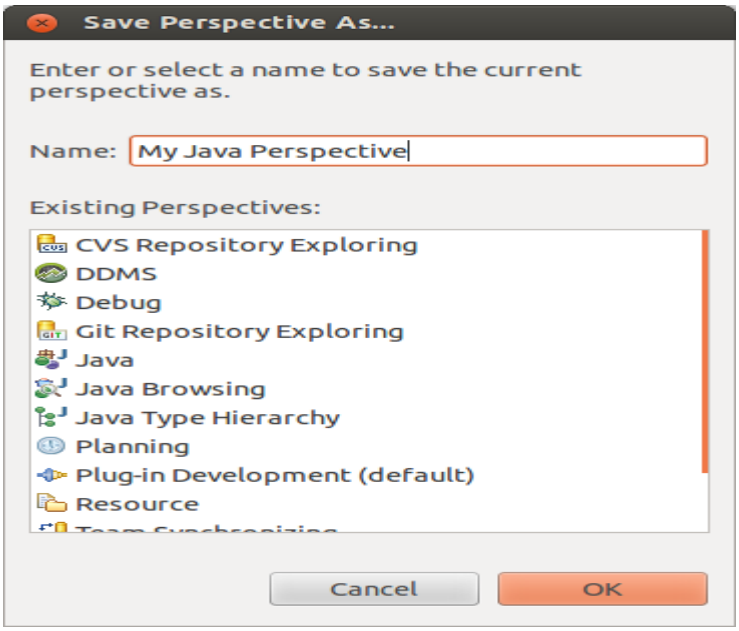

You can change the layout and content within a perspective by opening or closing parts and by re-arranging them.

To open a new part in your current perspective, use the Window → Show View → Other... menu entry. The following Show View dialog allows you to search for certain parts.

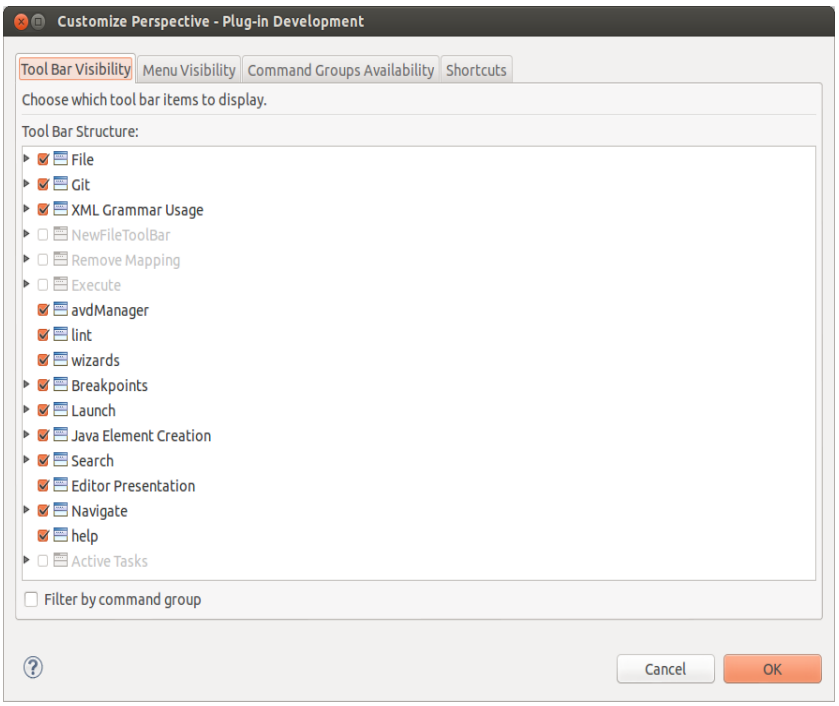



If you want to reset your current perspective to its default, use the Window → Reset Perspective menu entry.

You can save the currently selected perspective via Window → Save Perspective As....




The Window → Customize Perspective... menu entry allows you to adjust the selected perspective . For example, you can hide or show toolbar and menu entries.




# Eclipse Java development user interface

### Perspectives in Eclipse

Eclipse provides different *perspectives* for different tasks. The available *perspectives* depend on your installation.

For Java development you usually use the Java Perspective, but Eclipse has much more predefined *perspectives*, e.g., the Debug *perspective*.

Eclipse allows you to switch to another *perspective* via the Window → Open Perspective → Other... menu entry.

**Resetting a perspective**

A common problem is that you changed the arrangement of views and editors in your *perspective* and you want to restore Eclipse to its original state. For example, you might have closed a view .

You can reset a *perspective* to its original state via the Window → Reset Perspective menu entry.

**Java perspective and Package Explorer**

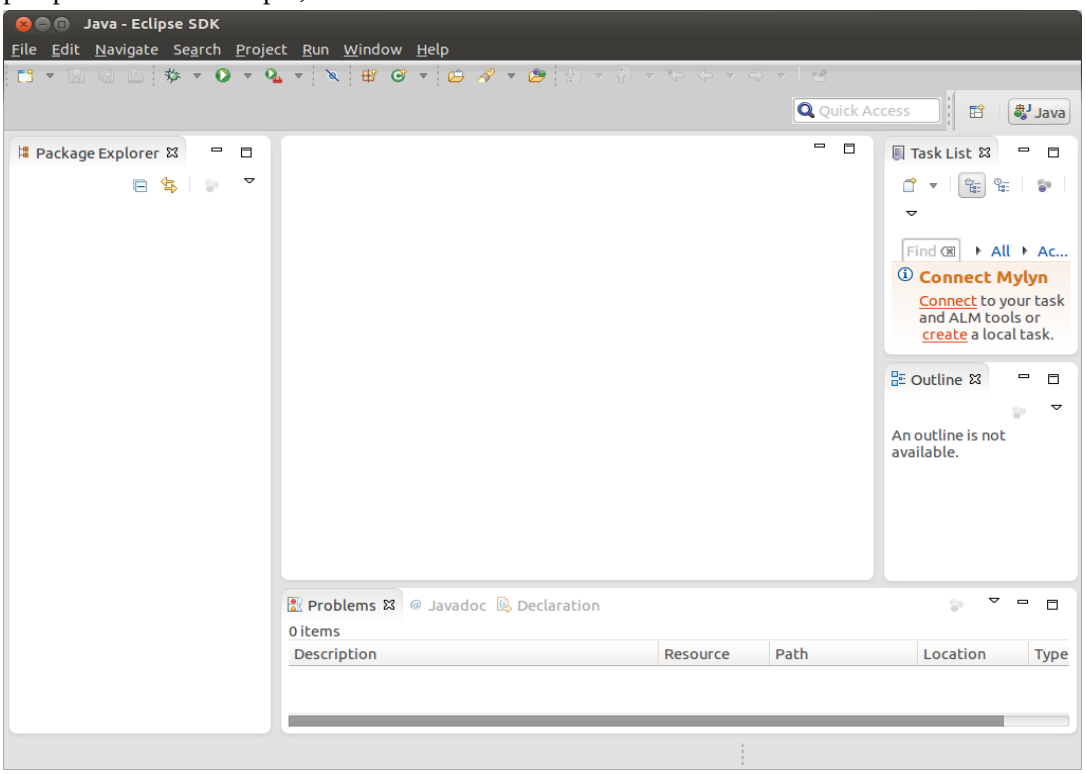
The default *perspective* for Java development can be opened via Window → Open Perspective → Java.

On the left hand side, this perspective shows the Package Explorer view, which allows you to browse your *projects* and to select the components you want to open in an editor via a double-click.

For example, to open a Java source file, open the tree under src, select the corresponding .java file and double-click it. This will open the file in the default Java *editor*.

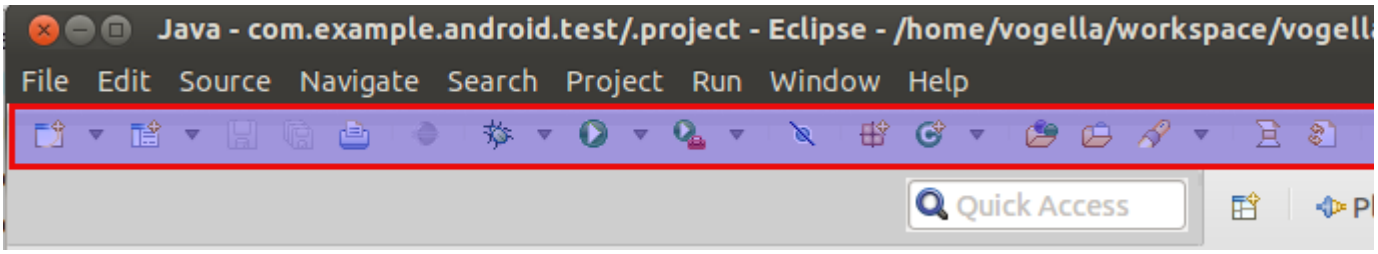
The following picture shows the Eclipse IDE in its standard Java *perspective*. The Package Explorer view is on the left. In the middle you see the open *editors*. Several *editors* are stacked in the same container and you can switch between them by clicking on the corresponding tab. Via drag and drop you can move an editor to a new position in the Eclipse IDE.

To the right and below the editor area you find more *views* which were considered useful by the developer of the perspective. For example, the Javadoc view shows the Javadoc of the selected class or method.




# Eclipse Java perspective

**Toolbar**
The application toolbar contains actions which you typically perform, e.g., creating Java resources or running Java projects. It also allows you to switch between perspectives.

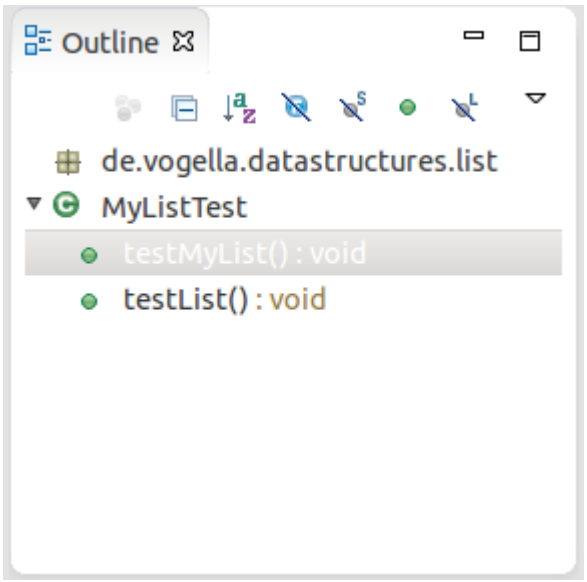



**Useful views**
The Java *perspective* contains useful *views* for working with your Java project. The following description explains the most important ones.

**Package Explorer view**
The Package Explorer view allows you to browse the structure of your projects and to open files in an *editor* via a double-click on the file.

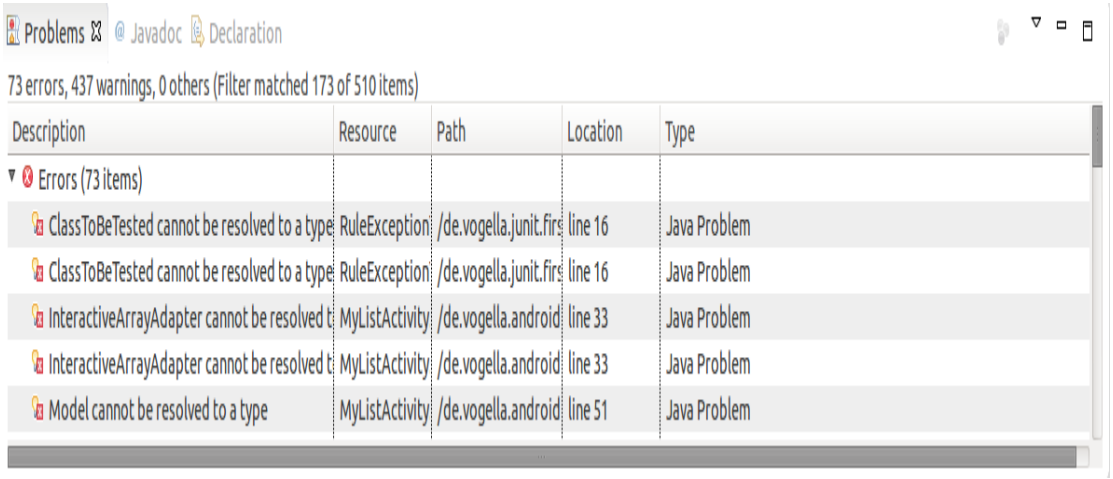
It is also used to change the structure of your project. For example, you can rename files or move files and folders via drag and drop. A right-click on a file or folder shows you the available options.

**Outline view**
The Outline view shows the structure of the currently selected source file.

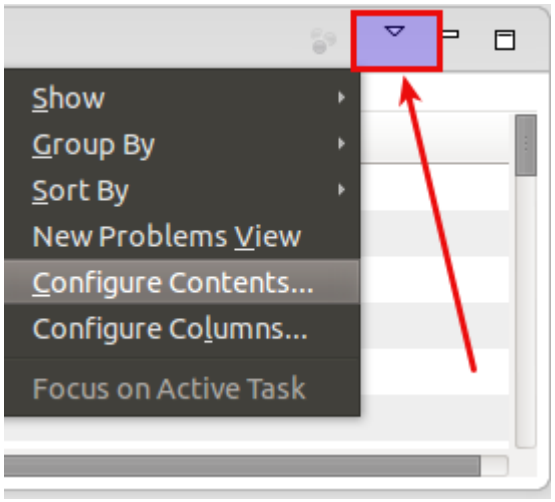


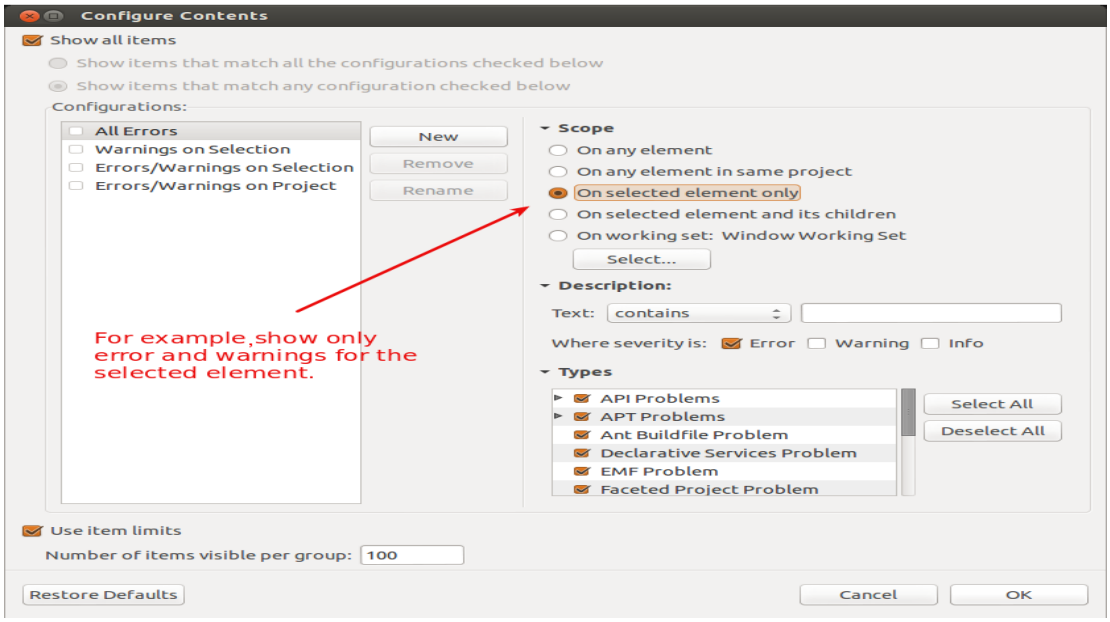

**Problems view**

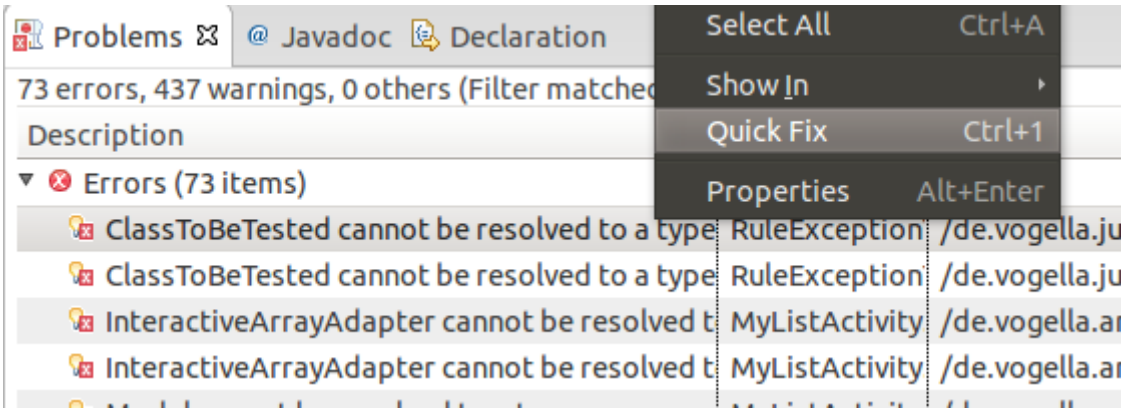
The Problems view shows errors and warning messages. Sooner or later you will run into problems with your code or your project setup. To view the problems in your project, you can use the Problems view which is part of the standard Java *perspective*. If this view is closed, you can open it via Window → Show View → Problems.




The messages which are displayed in the Problems view can be configured via the drop-down menu of the view . For example, to display the problems from the currently selected project, select Configure Contents and set the Scope to On any element in the same project.






**Javadoc view**
The Javadoc view shows the documentation of the selected element in the Java *editor*.

The Java *editor* is used to modify the Java source code. Each Java source file is opened in a separate *editor*.

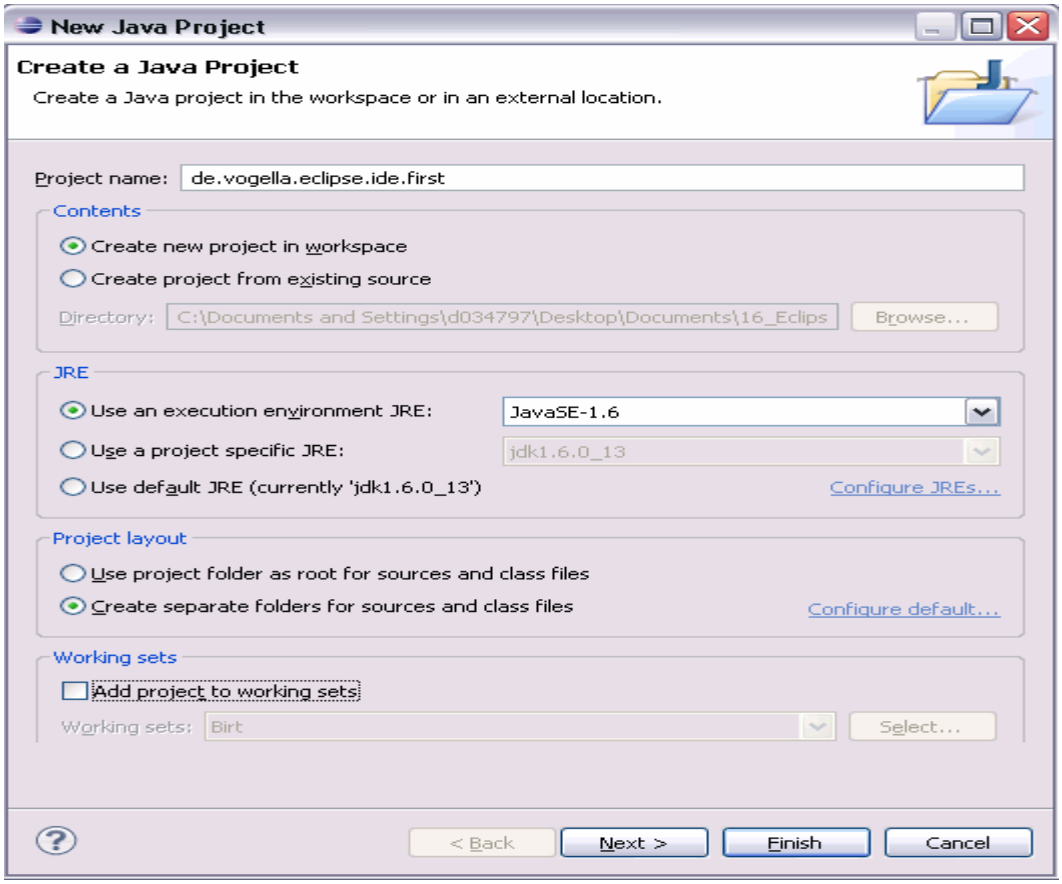
If you click in the left column of the editor, you can configure its properties, for example, that line number should be displayed.

# Create your first Java program

**Target of this exercise**
The following section describes how to create a minimal Java application using Eclipse. It is tradition in the programming world to create a small program which writes "Hello World" to the console. We will adapt this tradition and will write "Hello Eclipse!" to the console.

**Create project**
This book uses the naming convention that the project is named the same as the top-level package in the project.

Select File → New → Java project from the menu. Enter de.vogella.eclipse.ide.first as the project name. Select the Create separate folders for sources and class files flag.

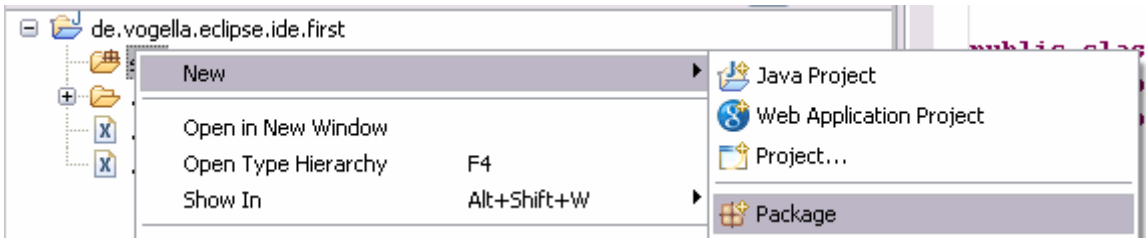



Press the Finish button to create the project. A new project is created and displayed as a folder. Open the de.vogella.eclipse.ide.first folder and explore the content of this folder.

**Create package**

In the following step you create a new package. A good convention for the project and package name is to use the same name for the top level package and the project. For example, if you name your project com.example.javaproject you should also use com.example.javaproject as the top-level package name.

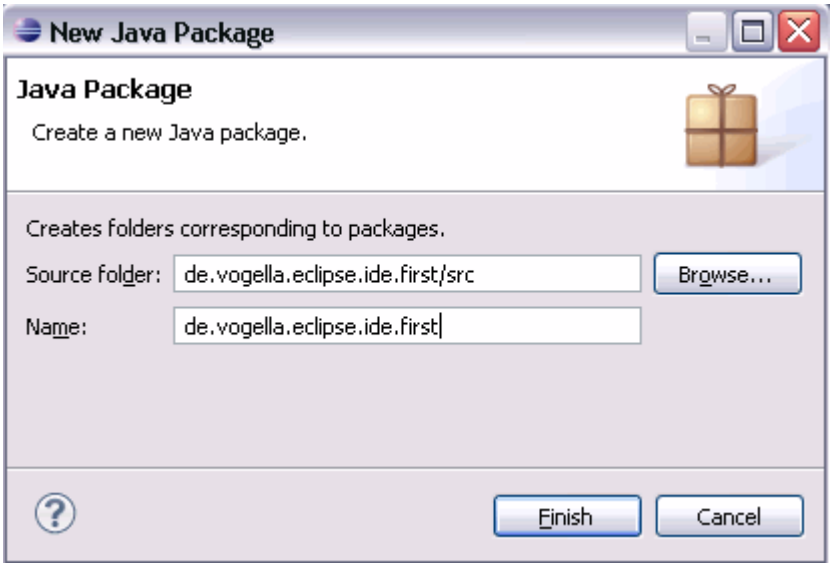
To create the de.vogella.eclipse.ide.first package, select the src folder , right-click on it and select New → Package.




**Tip**

Reverse domain names should be used for packages to prevent name clashes. It is relatively unlikely that another company defines a class called test in the com.vogella package because this is the reverse URL of the vogella GmbH company.

Enter the name of your new package in the dialog and press the Finish button.



Create Java class

Create a Java class. Right-click on your package and select New → Class.

Enter MyFirstClass as the class name and select the public static void main (String[] args) checkbox.

Press the Finish button.

This creates a new file and opens the Java *editor*. Change the class based on the following listing.

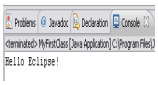
```
package de.vogella.eclipse.ide.first;

public class MyFirstClass {

  public static void main(String[] args) {
    System.out.println("Hello Eclipse!");
  }

}
```

You could also directly create new packages via this dialog. If you enter a new package in this dialog, it is created automatically.

**Run your project in Eclipse**

Now run your code. Either right-click on your Java class in the Package Explorer or right-click in the Java class and select Run-as → Java application.

Eclipse will run your Java program. You should see the output in the Console view .



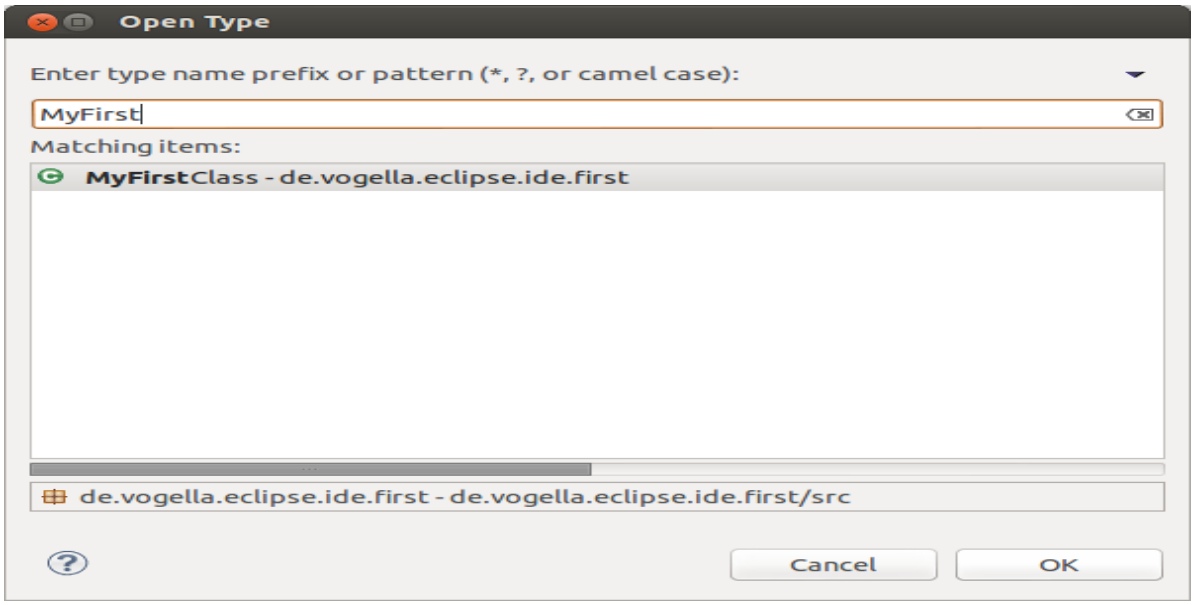
Congratulations! You created your first Java project, a package, a Java class and you ran this program inside Eclipse.

# Navigate in the Java source code

You can also use other means than the Package Explorer to navigate your source code. The following description lists the most important ones.

**Opening a class**
You can navigate between the classes in your project via the Package Explorer view as described before. You can navigate the tree and open a file via a double-click.

In addition, you can open any class by positioning the cursor on the class in an editor and pressing **F3**. Alternatively, you can press **Ctrl**+**Shift**+**T**. This shows the following dialog in which you can enter the class name to open it.



# Practical 3

**Title** **Program to determine sum of following harmonic series for a given number n.**

$$\mathbf{1} + \frac{\mathbf{1}}{\mathbf{2}} + \frac{\mathbf{1}}{\mathbf{3}} + \cdots\cdots + \frac{\mathbf{1}}{\boldsymbol{n}}$$

**Pre-requisite** Knowledge of loops

**Algorithm**

1. Start
2. Import header files
3. Declare variables sum , i, term
4. Initialize sum=1
5. Print "enter the no. whose harmonic series is to be calculated"
6. Read term
7. Repeat from i=1 to term
8. i+1,Calc: sum=sum + $1/i$
9. Goto step (10)
10. Print "harmonic series of the number=" ; sum
11. Stop

# Practical 4

**Title** **Program to print floyd's triangle .**

**Pre-requisite** Knowledge of loops and steps to print floyd's triangles

**Algorithm**

1. Start
2. Declare required variables for controlling loop, inputting number of rows and printing numbers.
3. Initialize number=1.
4. Enter the number of rows to be printed.
5. Print the number in standard format utilizing the application of loop as follows
6. do for x=1 to rows
7. do for y=1 to x
8. print number
9. increase the number by 1
10. go to next line
    [End of loop]
11. Print triangle
12. Stop

# Practical 5

**Title** **Find Volume of Sphere,Cone and Cylinder(using switch...case..default).**

**Pre-requisite** Knowledge of if and switch.

**Algorithm**

step 1: Start
step 2: Define pi<-3.14
step 3: Intilize 1<-1
step 4: check whether i=1 then go to step5
else go to step 24
step 5: Read x
step 6: check whether x=1,then go to step 7
else go to step 10
step 7: Read r
step 8: print 4/3*pi*r*r*r
step 9: break
step 10: check whether x=2,then go to step 11
else go to step 14
step 11: read r and h
step 12: print 1/3*pi*r*r*h
step 13: break
step 14: check whether x=3,then go to step 15
else go to step 18
step 15: read r and h
step 16: print 4/3*pi*r*r*r
step 17: break
step 18: check whether x=4,then go to step 19
else go to step 21
step 19: i=0
step 20: break
step 21: print invalid
step 22: break
step 23: go to step 4
step 24: Stop -

# Practical 6

**Title** **Program to reverse the digits of a number using while loop.**

**Pre-requisite** Knowledge of while loops and steps to find reverse of a number.

**Algorithm**

1. Start
2. Declare required variables for controlling loop
3. Enter the number.
4. Initialize reverse=0 and a = number.
5. Repeat while a!=0
6. a=num%10
7. reverse=reverse*10+a
8. a=a/10
9. print reverse
10. Stop

# Practical 7

**Title** **Program to print following output using for loop .**

**1**

**2 2**

**3 3 3**

**Pre-requisite** Knowledge of loops and steps to print floyd's triangles

**Algorithm**

1. Start
2. Declare required variables for controlling loop, inputting number of rows and printing numbers.
3. Enter the number of rows to be printed.
4. do for x=1 to rows
5. do for y=1 to x
6. print x
7. go to next line
   [End of loop]
8. Print triangle
9. Stop

# Practical 8

**Title** **Program to Make use of Inheritance.**

**Pre-requisite** Knowledge of Inheritance

**Algorithm**

1. Start
2. Create a class Staff and define methods code and name.
3. Create a class teacher derived from Staff and define methods subjects and publication.
4. Create a class typist derived from Staff and define methods speed.
5. Create a class officer derived from Staff and define methods grade.
6. Create a class regular derived from teacher
7. Create a class casual derived from teacher and define methods daily wages.

# Practical 9

**Title** **Program to implement multiple inheritance .**

**Pre-requisite** Knowledge of interfaces and inheritance

**Algorithm**

1. Start
2. Create a interface exam and declare methods percent_cal().
3. Create a class student and define methods display and constructor.
4. Create class result derived from student and implement interface exam.
5. Stop.

# Practical 10

**Title** **Program to implement runtime polymorphism .**

**Pre-requisite** Knowledge of pointers to objects

**Algorithm**

1. Start
2. Create a class figure and define constructor figure and method area.
3. Create a class rectangle derived from figure and define constructor and method area.
4. Create a class triangle derived from figure and define constructor and method area.
5. Create objects of figure ,rectangle and area class.
6. Stop

# Practical 11

**Title** **Javascript code to greet user with a message welcome when html page completes loading.When a user leaves this page the ggdbye alert dialog box is displayed .**

**Algorithm**

```
<html>
<head>
<title>practical-11</title>
<script>
var name;
function hello()
{
name=prompt("Enter your name");
alert("hello "+name+". Your's welcome to javascript world");
}
function bye()
{
alert("good bye "+name+". Come again");
}</script>
</head>
<body onload="hello()" onunload="bye()" >
<!-- onunload is not working -->
<p>Hello again to all my friends, together we can play some rock n roll! rock n roll
</p>
</body>
</html>
```

# Practical 12

**Title** **Javascript code to add two integers .**

**Pre-requisite** Knowledge of loops and steps to print floyd's triangles

**Algorithm**

```
<html>
<head>
<title>Practical-12</title>
<script>
var n1,n2,n3,n4,n5;
function setter()
{
n1=prompt("Enter first number");
parseInt(n1);
n2=prompt("Enter second number");
parseInt(n2);
n3=n1+n2;
}
function getter()
{
alert(n3);
}

</script>
</head>

<body>
<form>
<input type="button" value="enter" onclick="setter()" />
<input type="button" value="add" onclick="getter()" />

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

# Practical 13

**Title** **Write a function distance() which calculates the distance between two points (x1,y1) and (x2,y2).Incorporate this function into script that enables the user to enter the coordinates of the point to an html form.**

**Algorithm**

```
<html>
<head>
<title>practical-13</title>
<script>
var x1,y1,x2,y2,dis;
function dist()
{
x1=parseInt(document.myform.x1.value);
y1=parseInt(document.myform.y1.value);
x2=parseInt(document.myform.x2.value);
y2=parseInt(document.myform.y2.value);
dis=parseFloat(Math.sqrt(Math.pow((Math.abs(y2-y1)),2)+Math.pow((Math.abs(x2-
x1)),2)));
document.myform.res.value=dis;
}

</script>
</head>

<body>
<form name="myform">
x1<input type="text" name="x1" /><br />
y1<input type="text" name="y1" /><br />
x2<input type="text" name="x2" /><br />
y2<input type="text" name="y2" /><br />
result<input type="text" name="res" /><br />

<input type="button" value="calculate" onclick="dist()" />
</form>
</body>
</html>
```

# Practical 14

**Title** **Write code which makes use of form elements like Text Field, Text Area, radio button, check box etc.**

**Pre-requisite** Knowledge of form elements

**Algorithm**

```
<script type="text/javascript">
function validate() {
var combo1 = document.getElementById("country")
if(combo1.value == null || combo1.value == "") {
alert("Please select a country");
document.getElementById("cty").style.border = "2px solid red";
return false;
} else {
document.getElementById("cty").style.border = "";
}
}
</script>
<form action="" method="post" onsubmit="return validate()">
<span id="cty">
Country: <select name="country" id="country">
<option value="">Choose a country</option>
<option value="aus">Australia</option>
<option value="gb">Great Britain</option>
```

```
<option value="ind">India</option>

</select>

</span>

<input type="submit" value="Submit" />

</form>
```

# Practical 15

**Title** **Write a script to change the contents of P element.**

**Algorithm**

```
<html>
<body>

<p id="p1">Hello World!</p>

<script>
document.getElementById("p1").innerHTML="New text!";
</script>

</body>
</html>
```

# Practical 16

**Title** **Write a script to change the value of the src attribute of an <img> element:**

**Algorithm**

```
<!DOCTYPE html>
<html>
<body>

<img id="image" src="smiley.gif">

<script>
document.getElementById("image").src="landscape.jpg";
</script>

</body>
</html>
```

# Practical 17

**Title** **Write a script to change the content of H1 element when user clicks on it**

**Algorithm**

```
<!DOCTYPE html>
<html>
<head>
<script>
function changetext(id)
{
id.innerHTML="Ooops!";
}
</script>
</head>
<body>
<h1 onclick="changetext(this)">Click on this text!</h1>
</body>
</html>
```

# Practical 18

**Title** **Creating New HTML Elements (Nodes)**

**Algorithm**

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var para=document.createElement("p");
var node=document.createTextNode("This is new.");
para.appendChild(node);

var element=document.getElementById("div1");
element.appendChild(para);
</script>
```

# Practical 19

**Title** Removing Existing HTML Elements

**Algorithm**

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
var parent=document.getElementById("div1");
var child=document.getElementById("p1");
parent.removeChild(child);
</script>
```

# Practical 20

**Title** Write a script to use onmouseover and onmouseout events

**Algorithm**

```
<!DOCTYPE html>
<html>
<body>
<img onmouseover="bigImg(this)" onmouseout="normalImg(this)" border="0" src="smiley.gif"
alt="Smiley" width="32" height="32">

<p>The function bigImg() is triggered when the user moves the mouse pointer over the image.</p>
<p>The function normalImg() is triggered when the mouse pointer is moved out of the image.</p>

<script>
function bigImg(x) {
   x.style.height = "64px";
   x.style.width = "64px";
}
function normalImg(x) {
   x.style.height = "32px";
   x.style.width = "32px";
}
</script>

</body></html>
```

# Practical 21

**Title** **Writing XML web Documents which make use of XML Declaration, Element Declaration, Attribute Declaration.**

**Objective** To study and Implement XML.

**Pre-requisite** Knowledge of

1. XML

**Algorithm** **XML documents use a self-describing and simple syntax declaration.**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

**XML documents use a self-describing and simple element declaration.**

Imagine that the author of the XML document added some extra information to it:

```
<note>
<date>2002-08-01</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

**XML documents use a self-describing and simple Attribute declaration.**

Data can be stored in child elements or in attributes.

Take a look at these examples:

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
```

```
</person>
```

In the first example sex is an attribute. In the last, sex is a child element. Both examples provide the same information.

There are no rules about when to use attributes, and when to use child elements. My experience is that attributes are handy in HTML, but in XML you should try to avoid them. Use child elements if the information feels like data.

# Practical 22

**Title** **Usage of Internal DTD, External DTD.**

**Objective** To study and Implement Internal DTD, External DTD

**Pre-requisite** Knowledge of DTD.

## Why we need a DTD

XML is a language specification. Based on this specification, individuals and organizations develop their own markup languages which they then use to communicate information with. When this information is transferred from source to destination, the destination:

* Needs to know how the document is structured and

* Needs to check if the content is indeed compliant with the structure

The Document Type Definition also known as DTD holds information about the structure of an XML document. In this chapter we will understand the important aspects of DTDs. The concept of a DTD is not new. It actually finds its origins with SGML (remember the good old SGML?!) and has of course evolved since.

Any human or computer reader can "read" the DTD and understand how the document content will be made available. As a corollary, if the document content is not present in the way that the DTD has specified, the human or computer reader can reasonably assume that the content is not properly structured and throw an error or request a resend.

*DTDs **specify the structure** of XML Documents*
*DTDs **provide a basis for validating** XML Content*

## Placing the DTD in an XML document

Before we understand how to write a Document Type Definition, let us see where it's content is placed in order to provide the structure of the document.

This complete DTD content can have two placements:

*** Internal** – Refers to the placement of the DTD content directly within the XML document itself.

Every internal DTD is enclosed within the following two statements to differentiate it from the rest of the XML document:

"<!DOCTYPE root_element [" and "]>"

where *root_element* is the name of the root element of the XML document (remember every XML document must have a root element since XML is a tree structure).

For instance, consider the following XML file with an internal DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ARTICLES [
<!ELEMENT ARTICLES (ARTICLE*)>
<!ELEMENT ARTICLE (ARTICLEDATA*)>
<!ELEMENT ARTICLEDATA (TITLE, AUTHOR)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT AUTHOR (#PCDATA)>
]>
<ARTICLES>
    <ARTICLE>
        <ARTICLEDATA>
            <TITLE>XML Demystified</TITLE>
            <AUTHOR>Jaidev</AUTHOR>
        </ARTICLEDATA>
    </ARTICLE>
</ARTICLES>
```

Note that the Document Type Definition (shown in dark blue and boldface above) is placed within the XML document itself.

Also, the DTD specification is always placed after the first line which is always <?xml version …….?> and before the actual document content starts. This holds good even for External References, as we shall see in a moment.

*** External Reference** – Refers to saving the DTD as a file with extension .dtd and then referencing the DTD file within the XML document. For instance consider the two files shown below – the standalone DTD file articles.dtd and the actual XML file articles.xml based on this DTD.

Notice the similarity in positioning. The only difference from Internal DTD is that, in this case the DTD file is external and is only referenced in the XML document -articles.xml (as shown in dark blue and boldface).

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT ARTICLES (ARTICLE*)>
<!ELEMENT ARTICLE (ARTICLEDATA*)>
<!ELEMENT ARTICLEDATA (TITLE,
AUTHOR)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT AUTHOR (#PCDATA)>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ARTICLES SYSTEM
"D:\articles.dtd">
<ARTICLES>
    <ARTICLE>
        <ARTICLEDATA>
            <TITLE>XML Demystified</TITLE>
            <AUTHOR>Jaidev</AUTHOR>
        </ARTICLEDATA>
    </ARTICLE>
</ARTICLES>
```

An External DTD can feature as a file on the local file system or as a URL reference and each case is

specified slightly differently.

For instance, in the above example, the DTD file was on my file system as “D:\articles.dtd”. Since this was the case, the keyword SYSTEM was used to indicate that it is a personal external reference.

<!DOCTYPE ARTICLES SYSTEM "D:\articles.dtd">

Had the same DTD been available off my website (it is not, so don’t bother looking for it!!), it might have been specified with the keyword PUBLIC as:

<!DOCTYPE ARTICLES PUBLIC "-//JAIDEV//DTD ARTICLES XML V1.0//EN"
“http://www.mydomain.com/dtd/articles.dtd">

Here the formal name of the DTD is specified followed by the URL of the DTD location.

### Writing a DTD

By now we know the syntax of an XML document (from the previous chapter). To recall, every XML document markup is made up of

* Elements and

* Attribute-Values

The structure of these elements and attributes is exactly what a DTD seeks to formally provide. So, let us take a look at each.

## Elements

An element in a DTD is defined as:

<!ELEMENT element_name (child_content)>

Here element_name is the name of the element and the *child_content* is the content of the child (or children) of this element.

A *child_element* can be any of the following:

* A single element name: Implies only one child

* More than one element names separated by commas: Implies a sequence of children

* The keyword EMPTY: Implies no children

* The keyword ANY: Implies any combination within

* The keyword (#PCDATA) including the round brackets: Implies any text that will be parsed. Since this data will be parsed be careful not to use any markup text and special symbols.

* Choices: Implies that the children can be selected from among choices separated by the "|" symbol.

* Instance Quantities: Implies that the element can appear as many times as specified. Can be one of three types:

  o "*" implying "any number of times",

  o "+" implying "at least once" and

  o "?" implying "at most once".

Let us consider examples for each of these cases:

| Child_Element | Example |
|---|---|
| Single Element | <!ELEMENT ARTICLEDATA (TITLE)> |
| Multiple Elements | <!ELEMENT ARTICLEDATA (TITLE, AUTHOR)> |
| EMPTY | <!ELEMENT BR EMPTY> |
| ANY | <!ELEMENT UNIVERSAL_SET ANY> |
| (#PCDATA) | <!ELEMENT TITLE (#PCDATA)> |
| Choices | <!ELEMENT COMPANY_INFO (NAME \| (LOGO, ADDRESS))> |
| Instance Quantities | <!ELEMENT ARTICLES (ARTICLE*)><br><!ELEMENT ARTICLEDATA (TITLE, AUTHOR+)><br><!ELEMENT CHILD (MOTHER?, FATHER?)> |

### Attributes

Attributes as we have seen in the last chapter are additional pieces of information about an element. There has been a long standing debate about when to use attributes and when to breakdown an element into child elements. I shall not attempt to provide my own theory or rule-of-thumb. Please see the following site (one of many many) to guide you if you are lucky or spark your own imagination if you are not:

http://www-106.ibm.com/developerworks/xml/library/x-eleatt.html

Let us instead proceed to see how attributes are specified in a DTD. The following syntax holds:

**<!ATTLIST element attribute_name attribute_type additional_characteristic>**

where:

  o element is the name of the tag for which this attribute is being specified

  o attribute_name is the name of the attribute

  o attribute_type can be either of

    * "CDATA" for character data but no markup

* “ID” implying that the value cannot be repeated anywhere in the document i.e. it is unique

* “NMTOKEN” implying that the value must conform to XML identifier name specifications.

* choice list as (choice1 | choice2 |……| choiceN) o additional_characteristic can be one of

* “default_value” where default_value is the default value of the attribute

* #FIXED “default_value” where default_value is the default value of the attribute and this is the only value you can specify

* #REQUIRED implying that a value is mandatory

* #IMPLIED implying that the value (or default) is optional

Once again let us see some examples:

| Attribute Specification | Example |
|---|---|
| *attribute_type* is CDATA and it has an implied value | <!ELEMENT ARTICLE (#PCDATA)><br><!ATTLIST ARTICLE TYPE CDATA #IMPLIED> |
| *attribute_type* is a choice list and a value is required | <!ELEMENT COMPANY (#PCDATA)><br><!ATTLIST COMPANY TYPE (COMPANY \| BUSINESS) #REQUIRED> |
| Specifying a fixed default value as 2004 | <!ELEMENT SURVEYDATA (#PCDATA)><br><!ATTLIST SURVEYDATA BASELINE CDATA #FIXED '2004'> |
| Specifying that a value must be unique | <!ELEMENT PERSON (#PCDATA)><br><!ATTLIST PERSON EMAIL ID #REQUIRED> |
| Specifying that a value must be a NMTOKEN | <!ELEMENT PERSON (#PCDATA)><br><!ATTLIST PERSON FIRSTNAME NMTOKEN #REQUIRED> |

### A Frequently Asked Question on Binary Data

Before we close this chapter on DTDs, let us relook at a FAQ (that we did look at in the last chapter).

Q. Can XML files hold binary data as part of an element?

A. No. XML files can only carry text data

Q. Is that not restrictive? What do I do for binary data then? All my image files, sound files etc?

A. Well, yes it is restrictive but that is what gives XML its power. XML being plain text can be transported and understood so easily. Instead of transporting binary data as part of an XML document one can simply reference the binary file as an attribute of an element or another element itself. For instance, consider the following element and attribute definition in the DTD: <!ELEMENT MYPHOTO EMPTY> <!ATTLIST MYPHOTO FILENAME ENTITY #REQUIRED> The corresponding XML element would be: <MYPHOTO FILENAME=”jaidev.jpg”/> Now, any program using this XML data could easily acquire the data separately using native mechanisms. For instance a

browser encountering an image reference tag could easily make a request over HTTP, obtain the binary information and display the JPEG file.

### DTDs rock but….

… they do have some drawbacks. Perhaps the two most significant ones are: * There is no formal datatyping scheme * All definitions have a global scope To solve some of these drawbacks, the concept of XML Schemas was introduced. In the next chapter we will consider the basics of Schemas and we will also discuss a tad bit more on validation of XML documents that are based on DTDs or Schemas. I do hope you try out some of the examples given above using an XML editor. You can also practice writing your own DTD for any common type of document that you deal with – perhaps for personal data, or account information or maybe ticketing data. Take your pick but do it. Practice makes perfect, at least from this chapter onwards!

## Introduction to DTD

---

The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. A DTD can be declared inline in your XML document, or as an external reference.

---

**Internal DTD**

This is an XML document with a Document Type Definition: (Open it in IE5, and select view source)

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note    (to,from,heading,body)>
  <!ELEMENT to      (#PCDATA)>
  <!ELEMENT from    (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body    (#PCDATA)>
]>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The DTD is interpreted like this:
**!ELEMENT note** (in line 2) defines the element "note" as having four elements: "to,from,heading,body".
**!ELEMENT to** (in line 3) defines the "to" element  to be of the type "CDATA".
**!ELEMENT from** (in line 4) defines the "from" element to be of the type "CDATA"
and so on.....

---

**External DTD**

This is the same XML document with an external DTD:  (Open it in IE5, and select view source)

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

This is a copy of the file "note.dtd" containing the Document Type Definition:

```
<?xml version="1.0"?>
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

---

**Why use a DTD?**

XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

A lot of forums are emerging to define standard DTDs for almost everything in the areas of data exchange.

# Practical 23

**Title** Php program to add two numbers

**Algorithm**

```
<?php

$first_number = 10;
$second_number = 20;
$sum_total = $first_number + $second_number;

$direct_text = 'The two variables added together = ';

print ($direct_text . $sum_total);

?>
```

# Practical 24

**Title**

Php program to print n numbers

**Algorithm**

```
<html>
<body>
<?php
$i = 0;
$num = 0;
do
{
  $i++;
echo ($i );


}
while( $i < 10 );
?>
</body>
</html>
```

# Practical 25

**Title** Php program to store and restore session variables

**Algorithm**

```
<?php
session_start();
// store session data
$_SESSION['views']=1;
?>

<html>
<body>

<?php
//retrieve session data
echo "Pageviews=". $_SESSION['views'];
?>

</body>
</html>
```